A Proof Planning Framework For Isabelle

Lucas Dixon

Doctor of Philosophy

Centre for Intelligent Systems and their Applications

School of Informatics

University of Edinburgh

2005

Abstract

Proof planning is a paradigm for the automation of proof that focuses on encoding intelligence to guide the proof process. The idea is to capture common patterns of reasoning which can be used to derive abstract descriptions of proofs known as *proof plans*. These can then be executed to provide fully formal proofs.

This thesis concerns the development and analysis of a novel approach to proof planning that focuses on an explicit representation of choices during search. We embody our approach as a proof planner for the generic proof assistant *Isabelle* and use the *Isar* language, which is human-readable and machine-checkable, to represent proof plans. Within this framework we develop an inductive theorem prover as a case study of our approach to proof planning.

Our prover uses the difference reduction heuristic known as *rippling* to automate the step cases of the inductive proofs. The development of a flexible approach to rippling that supports its various modifications and extensions is the second major focus of this thesis. Here, our inductive theorem prover provides a context in which to evaluate rippling experimentally.

This work results in an efficient and powerful inductive theorem prover for Isabelle as well as proposals for further improving the efficiency of rippling. We also draw observations in order to direct further work on proof planning. Overall, we aim to make it easier for mathematical techniques, and those specific to mechanical theorem proving, to be encoded and applied to problems.

Acknowledgements

This research was funded by EPSRC grant GR/N37414/01 and the work on this thesis has also been supported by many people. Jacques Fleuriot, my supervisor, has helped me throughout, especially with his bard-like advice on weaving stories. Technically as well as socially the members of the DReaM group have also been constant torrent of support and inspiration. This research group has exemplified clarity of mind, humility and honesty of science, and I cannot say how much I appreciate you all. I would also like to thank my examiners who gave helpful and insightful feedback.

The flexibility to enjoy working on this thesis has come from those who opened my mind, those who showed me how to break down walls, and how to indulge feelings with a smile. My amigos from tango provided such delightful and essential distraction. Aikido and the friends who practice it have also played an essential role in making me feel balanced. This sense has given me stability to work from.

But my deepest determination, this source of optimism and happiness, has come from the cartoons that my sister sends me, from the lunches with my mum and her ear for dreams, from the conversations with my brother, his deep hopefulness and natural scientific calm. Also from my sister who shares so many quirks of my mind that her adventures and lessons are mine also. To my father I owe my happy madness and to other-father the silicon seeds that guided to me to this work.

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.

(*Lucas Dixon*)

Table of Contents

Chapter 1

Introduction

The formalisation and automation of mathematical proof is one of the central foundations of artificial intelligence. Any problem that can be clearly defined can be phrased in terms of the discovery of a proof. However, we know from the incompleteness result of Gödel [42, 99] and the presentation of the halting problem by Turing [98] that in any practical system we develop there will be problems for which we cannot find a proof.

Despite this negative result, in practice, formalisation has many useful applications. For instance, the characterisation of software and hardware in proof systems has been used to prove properties about their behaviour. Another salient application is the expression, validation and automation of mathematical reasoning, which was started in 1967 by de Bruijn's Automath project [31]. Since then, a large body of mathematics as well as software and hardware has been formalised in a diverse collection of systems including Mizar [91], Isabelle [84], ACL2 [57, 58], HOL [43], PVS [79], and NuPRL [30]. Thus the negative theoretical results of Godel and Turing seem to have had little impact on the practical task of formalisation. However, the process of mechanisation is still largely considered difficult, slow, and cumbersome. An important research direction that attempts to address this issue is further automation of proof. This can be done by either adding new axioms or by automating the use of existing ones. The latter approach avoids the danger of introducing inconsistencies with the new axioms, but it requires more work to develop.

In 1988 Bundy suggested an approach, called *proof planning*, to ease the process of extending proof systems by providing a language with which to express common patterns of reasoning [14, 16, 20]. Applying the encoded reasoning to problems results in abstract descriptions of proofs called *proof plans*. These can then be executed to get a fully formal proof in terms of the basic axioms of an underlying proof system. However, proof planning systems have rarely ensured that the found proof plans can be executed to produce an independently checkable represenrtation of the proofs, and have always separated the execution from the planning. This brings into question the feasibility of a proof planning system that does produce fully formal proofs. One of the contributions of this thesis is to show that the generation of fully formal proofs by interleaving proof planning with the proof plan's execution is feasible and often desirable.

Proof planning has been championed by the development of inductive theorem proving machinery. In particular, it was shown that a difference reduction heuristic, termed *rippling*, can be used to guide the step cases of inductive proofs [17,19]. Many variations and extensions have been proposed for rippling and a number of proof planning techniques have been developed for other domains such as non-standard analysis [67], hardware verification [22], and first-order temporal logic [23].

Traditional approaches to proof planning have represented the encoded patterns of reasoning as well as the generated proof plans in a interpreted and declarative form. The role of the proof planner is to interpret the encoded techniques and use them to derive proof plans. This thesis proposes a novel approach to proof planning based on an explicit representation of choices within an encoded pattern of reasoning. We evaluate our proposal by implementing it in a system, which we name *IsaPlanner*, and then by redeveloping and extending the inductive theorem proving technology introduced in the proof planning literature.

Our system builds on the generic theorem prover Isabelle [84] and the Isar language [102]. In this sense, our work brings together recent developments in structured proof languages, proof checking, and proof planning. Following the methodology of Isabelle's existing proof tools, we show that proof planning techniques can be made generic. Furthermore, we show that it is feasible to ensure the soundness of proof planning by justifying results in terms of the underlying proof system.

The second contribution of this thesis is the development and extension of inductive theorem proving techniques. In particular, we provide a rich framework for expressing different versions of rippling. This results from a novel adaptation of rippling for higher order settings that allows us to capture many of the variations introduced in the literature and compare them. We perform an empirical evaluation of different versions within the context of our inductive theorem prover. This helps clarify the effect of variations to rippling for an inductive theorem prover. The result of these experiments highlight modifications to rippling that could further improve its efficiency.

This results in expanding the applicability of proof planning and extending the automation of inductive proof in Isabelle. Finally, the development these techniques has lead us to remark on how proof planning frameworks can be further improved.

Publications

Different aspects of the work presented in this thesis have been published in [36–39].

1.1 Contributions

In summary, the main contributions of this thesis are:

- The introduction and analysis of a novel approach to proof automation we term *observational proof planning*.
- A representation of proof plans as Isar proof scripts which allows the automatic generation and manipulation of readable and executable proof scripts. However, we found this representation to be somewhat problematic. From our analysis of these problems as well as issues in other systems we provide a brief review to guide further research in the representation of proof plans.
- A generic proof planner that interleaves proof planning and the execution of the generated proof plan, ensuring the soundness of the resulting proof.
- Improved automation in Isabelle which is provided by techniques that implement an inductive theorem prover. We show some of the choices available to an inductive theorem prover and illustrate their importance by experimentation.
- A rich framework for experimenting with rippling and a study of many of the variations available within the context of our inductive prover. This study shows that a novel measure for rippling improves on the traditional one within the examined domains.

1.2 Thesis Outline

We start by presenting the foundations on which this work is based in chapter 2. This involves an introduction to proof planning, interactive tactic driven proof in Isabelle, the readable and machine checkable Isar language, and the rippling technique traditionally used to guide inductive proof. Our observational approach to proof planning is detailed in chapter 3. This provides an abstract view of our approach which we then illustrate, in chapter 4, with machinery to trace the application of encoded techniques. We describe the relationship between techniques and search in chapter 5. In this chapter, we also provide flexible and extensible machinery that allows techniques to specify search strategies which can be applied locally or globally.

The tools described up to this point in the thesis are independent of Isabelle and abstract over the actual representation of proof plans. In chapter 6 we provide a concrete implementation of proof plans as Isar proof scripts for Isabelle. We analyse this representation for proof plans to suggest features for a more flexible representation.

We then examine the rippling technique as a prelude to implementing an inductive theorem prover. In chapter 7, we provide flexible representation of rippling suitable for higher-order settings. We examine issues of efficiency and provide suitable algorithms for working with our representation. This results in several open questions for an implementation of rippling. In particular, we show that there are many variations of rippling which will behave differently. This leads towards the development of an inductive theorem prover which we can use on the one hand to evaluate the variations to rippling, and on the other as a tool in its own right to improve the automation in Isabelle. Before describing the development of our inductive theorem proving technique in chapter 9, we describe, in chapter 8, some basic machinery for Isabelle which provides the necessary support for fine grained control of reasoning with equations. Within the presentation of our inductive theorem prover we note the utility of our approach to proof planning.

In chapter 10 we evaluate our inductive theorem prover and examine the effects of various modifications to rippling. We perform experiments that show the effectiveness of our inductive theorem proving technique and highlight which variations of rippling are most effective in terms of proving power and speed.

Finally, in chapter 11 we summarise our experience with the observational approach to proof planning. We note particular characteristics of the language that are beneficial. We also highlight important areas for further work, largely in the representation of proof plans and in improved rewriting machinery for rippling.

Chapter 2

Background

2.1 Higher Order Abstract Syntax

Higher-order abstract syntax (HOAS) is a technique for the representation of syntax trees for languages with variable binders. The term was introduced by Pfenning and Elliot and used for a HOAS that provided practical representation for programs, formulas, rules and other such syntactic objects [85]. We will use the term HOAS, as has become common in the literature, to refer to the representation of term syntax with binders. Thus we consider the term representations of Isabelle [84] and HOL [43] and the encodings in λProlog [74], Twelf [86], Coq [4], to all be variations of HOAS. The basic idea, which goes back to Church [28], is to provide a uniform treatment of name binding for higher order settings. This can be done by representing terms in the λ-calculus using a datatype. For example, a HOAS for untyped λ-terms could be given by the following datatype:

```
term = Abs(term)
     | App(term * term)
     | Bound(nat)
```

This uses de Bruijn indices to refer to bound variables: *Bound i* refers to the i^{th} abstraction (*Abs*) above it in the term tree. It is common to write the application of "*f*" to "*x*" using a space and "*Abs*" as "λ*x*" where *x* replaces the binding constructors that refer to that abstraction. For example, "$Abs(App(f, Bound\ 0))$" would simply be "$\lambda x.\ (f\ x)$". When convenient, following the literature, we will also write as lambda abstractions with names and use these names instead of the bound indices. For example, the term "$Abs(Abs(App(Bound\ 1, Bound 0)))$" can be written more succinctly as "$\lambda x.\ \lambda y.\ x\ y$". We indulge a final notational convenience which is to group the binding of variables so that "$\lambda x_0.\ \ldots\ \lambda x_n.\ t$" becomes "$\lambda x_0 \ldots x_n.\ t$".

One of the central motivations for using HOAS is that it allows a single implementation of many tools, such as unification, matching, and substitution, to suffice for all objects represented in it. This makes it a practical internal representation for proof systems which are based on unification matching and substitution.

The datatype expressing HOAS can be much more complex than the example given above. For instance, that used by Isabelle also includes typing information, names for abstractions, and a special case for constants, free variables and meta variables[1]. When we write Isabelle terms we will prefix meta variables by a "?". Types for terms can usually be inferred, but we will sometimes provide them explicitly using "::" infix with the type on the right hand side. Constants will always be defined before they are used. Variables will be clear from the context and will usually be single letters. For example, given a polymorphic binary infix constant "+", the term "$?a + ?b = ?b + (?a :: nat)$" expresses the commutativity of "+" for objects of type "*nat*".

2.2 Isabelle

Isabelle is a proof assistant written in ML which supports formal reasoning in a number of object logics [83, 84]. Examples of such object logics include Zermelo-Fraenkel set theory (ZF), first order logic (FOL), higher order logic (HOL), and constructive type theory (CTT). Object logics are formed and manipulated by Isabelle's intuitionistic higher order meta-logic, which supports polymorphic typing and performs type-inference.

Formalisation of mathematics in Isabelle involves defining constants and types about which properties are then proved. Mixfix annotations are used to manage the parsing and printing for the concrete syntax of the underlying lambda calculus. Syntax translations support more complex relationships between the syntax and the underlying terms.

Soundness is treated by following the LCF design principle of having a fixed logical kernel containing the primitive inference rules. Additional *tactics* to perform higher-level proof steps are written in terms of these rules and previously proved theorems. The ML type system ensures that theorems are constructed only in this manner, thus reducing concerns about the soundness of new tools to the consistency of the logical kernel. This provides a disciplined approach to ensuring soundness while providing flexibility for the development of more powerful

[1]For a more detailed account of terms in Isabelle, see the Isabelle reference manual [82].

proof tools.

To ease and speed the proof process, Isabelle provides the user with a number of generic, as well as logic-specific proof tools. These range from simple mechanisms for combining theorems to fully automatic theorem provers. One of these is the generic simplification package which supports higher order conditional rewriting using previously proved theorems. The user can customise its behaviour by temporarily or permanently adding theorems to the simplification set. Other generic automatic tactics provided by Isabelle include a classical reasoner [80, 81] and the automatic tactic which attempts to prove all subgoals by a combination of simplification and classical reasoning.

Another important requirement for practical theory development is the need for tools to support new definitions. In the methodology of conservative extensions, adopted by the higher order logic of Isabelle (Isabelle/HOL), these mechanisms should not assert new axioms. Isabelle/HOL hosts an several such conservative mechanisms for writing definitions. These including support for inductively defined sets, inductive datatypes, types as sets, extensible records and the usual mechanisms for defining functions and types.

Isabelle's higher order logic provides the largest theory platform for further development and is the most widely used logic. It sports a large theory library of formalised mathematics developed as conservative extensions of the object logic. This includes developments within nonstandard analysis [41], a formalisation of Hilbert's axioms for geometry [70], and mechanisations of number theory [89], among many others [53]. Recently, Isabelle has also successfully imported all the theories from the HOL system.

The Representation of Proof

Isabelle can produce proof terms that describe a theorem's derivation in terms of the primitive logical inferences. This supports the validation of proof using a small proof checker independent of Isabelle. However, such proof terms provide far too much detail to be humanly checked, let alone easily readable. Furthermore, as well as failing to capture the 'idea' behind a proof, these do not provide a useful way of storing proofs. This is due to their verbose nature and the lack of support for their modification and maintenance. It is thus normal for users of Isabelle to store *proof scripts* in a file that contains the tactic commands to re-derive the proofs.

Before the development of Isar, proof scripts were typically expressed 'procedurally' as a sequence of ML proof commands. For example, a procedural-style proof that the sum of odd numbers up to n is equal to n^2, is shown Fig. 2.1. In this proof script, the `by` function applies a

```
Goal "$\sum_{i<n} 2 * i + 1 = n^2$";
by (induct_tac "n" 1);
by (Simp_tac 1);
by (Simp_tac 1);
by (simp_tac (simpset() addsimps [power2_eq_square]) 1);
qed "sum_of_odds";
```

Figure 2.1: *An example ML procedural proof for the sum of n odd numbers in Peano arithmetic.*

tactic to the current goal and `qed` stores a proved theorem for later use. The tactic `induct_tac` selects and applies an induction scheme, and `Simp_tac` and `simp_tac` are tactics that simply the goal. The latter simplification tactic is given an explicit simplification set, which in the above proof includes the lemma (`power2_eq_square`: $n * n = n^2$). Although these proofs support reuse of tactics, they are generally not readable off-line, that is without tracing through the goals resulting from each proof step. Isar provides a structured, human-readable, and Isabelle-checkable language for writing proofs. We describe this language further in section 2.3.

Theories, Locales and Type Classes

Within each of Isabelle's logics, developments are organised into *theories*. These are the course-grained basic objects for organising mathematical development as well as storing constants, types, sorts, syntax information, theorems, and contextual information used by proof tools.

Isabelle's theories provide local name spaces and support inheritance. Inheriting from several theories merges their contents, and merges the information used by the corresponding proof tools. Theories are outside the logic, in the sense that Isabelle cannot reason about a theory as an object in its own right. A notion of modularity within the logic is provided by the *locales* mechanism [3, 56]. This supports modularity using Isabelle's meta-logic in terms of parameters, that correspond to abstract constants, which are fixed over a collection of assumptions. For example, a semi-group can be specified as follows:

locale `semigroup =`
  fixes `prod :: "'a ⇒ 'a ⇒ 'a" (infixl "·" 70)`
  assumes `assoc:` $(x \cdot y) \cdot z = x \cdot (y \cdot z)$

Any theorem proved within this locale implicitly assumes the statement `assoc` and each occurrence of the constant "·" is in fact a free variable. For example, the theorem $(w \cdot (x \cdot y)) \cdot z = (w \cdot x) \cdot (y \cdot z)$ proved within the `semigroup` locale, corresponds to the meta-logical theorem:

$$p\ (p\ x\ y)\ z = p\ x\ (p\ yz) \Longrightarrow p\ (p\ w\ (p\ x\ y))\ z = p\ (p\ w\ x)\ (p\ yz)$$

Although Locales provide a powerful tool for modularity and have been extensively used in many formalisations, they are still limited by Isabelle's inherent lack of support for quantifying over types. Locales provide a mechanism for modularity without having to assert new axioms. However, in Hindley/Milner style higher order logics as used in Isabelle [75, 77], proof from an axiom is not equivalent to proof from an assumption. This is due to types implicitly being universally quantified over the whole statement. This means that modularity using locales is not exactly equivalent to the use of axioms. One solution to regain this equivalence is to extend higher order logic with quantification over types, as described by Melham [97].

Modularity is also provided by Isabelle's axiomatic type classes [101]. These allow classes of types to be defined in terms of basic properties that hold for the class. From these, theorems can be proved about the objects within the type-class. Isabelle's unification supports type-classes and thus the proof tools fit naturally. Unfortunately, type classes are limited in their expressivity. For instance, they can only be dependent on a single type variable.

Proof Construction and Notation

Proof construction in Isabelle is essentially by higher order resolution of sequents at the meta level. The meta level connectives are implication ($\Longrightarrow$), universal quantification ($\bigwedge$) and equality ($\equiv$). To prove a goal G an initial theorem of the form $G \Longrightarrow G$ is created by instantiation of the basic meta-logical axiom *trivial*. The premise is then resolved with previously proved results. This theorem represents the proof state. For example, we could resolve our proof state with another theorem of the form $P' \Longrightarrow G'$, where G' unifies with G, to get a new proof state $P \Longrightarrow G$, where P is P' instantiated from the unification of G' with G. Thus we see that the meta level implication is to denote subgoals as well as assumptions. When multiple assumptions exist, we follow Isabelle's syntax and abbreviate terms such as $A_0 \Longrightarrow (A_1 \Longrightarrow \ldots (A_n \Longrightarrow C))$ to $[\![A_0; \ldots A_n]\!] \Longrightarrow C$.

In Isabelle, *meta-variables* correspond to variables that are universally quantified at the meta-level and whose scope is around the whole of the theorem. In order to stop meta variables becoming instantiated, Isabelle can also represent them as what are called *free-variable*. These have the same meaning as meta variables but avoid instantiation during resolution. They are denoted just by their name.

Variables bound by meta-level universal quantifies around a subgoal correspond to arbitrary but fixed parameters. For example, in a proof that there are infinitely many primes, we can reduce this to showing that for any arbitrary but fixed number n we can construct a new prime p larger than it. This would be expressed as the following theorem in Isabelle syntax:

$$\bigwedge n.\ (?f\ n) > n\ \wedge\ \mathit{Prime}\ (?f\ n) \Longrightarrow \mathit{Infinite}\ \{x \mid \mathit{Prime}\ x\}.$$

where the meta-variable $?f$ expresses the existence of some way to construct a prime greater than n. Isabelle manages the parameters to meta variables. When this goal is eventually proved, the assumption will be resolved away leaving the theorem $\mathit{Infinite}\ \{x \mid \mathit{Prime}\ x\}$.

2.3 Isar

Isar aims to provide a language which is both human-readable and machine-checkable [102], following the style used by the Mizar system [91]. It provides a natural deduction style of writing proofs for the Isabelle theorem prover and allows abbreviations using higher order pattern matching. It is independent of the object logic and has been instantiated for Isabelle's HOL, ZF, and FOL, for instance. Furthermore, it has been designed in an extensible fashion which supports defining additional domain specific elements.

A small example Isar script, proving that the sum of odd numbers up to n is equal to n^2, is shown in Fig. 2.2. This script shows a feature of Isabelle that allows "$\lambda\ n.\ \sum_{i<n} 2*i+1$" to be abbreviated to $?\mathit{sumto}$ by unifying the higher order pattern "$?\mathit{sumto}\ n = \_$" with the main goal, where $?\mathit{sumto}$ is a variable and "$\_$" is a wildcard.

Another feature of Isar shown in this proof script is the support for a calculational style of proof, in the sense of iterated chains of transitive reasoning. In Fig. 2.2, this is indicated by the sequence of commands "**have**", "**also have**" and "**finally show**". The ability to name assumptions, for example by the calling the induction hypothesis "`IH`" in Fig. 2.2, further supports calculational and other forward styles of proof.

We remark that although backward proof is supported within the language, if backward steps are too large or numerous the proofs are once more unreadable and procedural in style.

```
theorem sum_of_odds: $\sum_{i<n} 2 * i + 1 = n^2$ (is ?sumto n = _)
proof (induct n)
  show ?sumto 0 = $0^2$ by simp
next
  fix n
  assume IH: ?sumto n = $n^2$
  have ?sumto (Suc n) = ?sumto n + Suc(2 * n) by simp
  also have ... = $n^2$ + Suc(2 * n) using IH by (simp)
  also have ... = $(Suc\ n)^2$ by (simp add: power2_eq_square)
  finally show ?sumto (Suc n) = $(Suc\ n)^2$ .
qed
```

Figure 2.2: *An example Isar proof for the sum of n odd numbers in Peano arithmetic.*

```
theorem sum_of_odds: $\sum_{i<n} 2 * i + 1 = n^2$ (is ?sumto n = _)
proof (induct n, simp, simp)
  fix n
  show Suc (2 * n + n) = (Suc n) by (simp add: power2_eq_square)
qed
```

Figure 2.3: *An example Isar proof where the backward proof step is so large that it obscures the proof.*

For example, the proof shown in Fig. 2.2 can be expressed in a briefer, but more procedural form, as shown in Fig. 2.3. In this example, we show a proof script in which the backward proof step includes induction, simplification to solve the base case, and a simplification that applies the induction hypothesis to the step case. The resulting goal is then made explicit and proved by adding the lemma `power2_eq_square` to the simplifier. However, because of the large backward proof step, it becomes unclear why showing this subgoal proved the main theorem. Furthermore, the combination of proof steps in the **proof** command, are essentially procedural as they hide the structure of the inductive proof. This also shows that some discipline is needed to write Isar proof script that are readable.

Internally, Isar operates as a state machine with transitions that incrementally parse elements in the proof language. This machinery has two main modes, one of which supports forward proof by allowing the user to express statements and one of which supports backward proof by allowing the user to apply tactics. Fig. 2.4 shows basic elements of the language and how they effect Isar's mode. The transitions take arguments which are omitted for the sake clarity. The basic Isar machinery is designed in terms of Isabelle's meta logic and is thus independent of any object logic. Domain specific additions are be provided by the extensible design of Isar. This allows new transitions and extensions to the notion of context build on top of the basic machinery. Such additions allow new notations for proof, such as the calculational style described earlier, to extend the basic Isar language within the theories that support them.

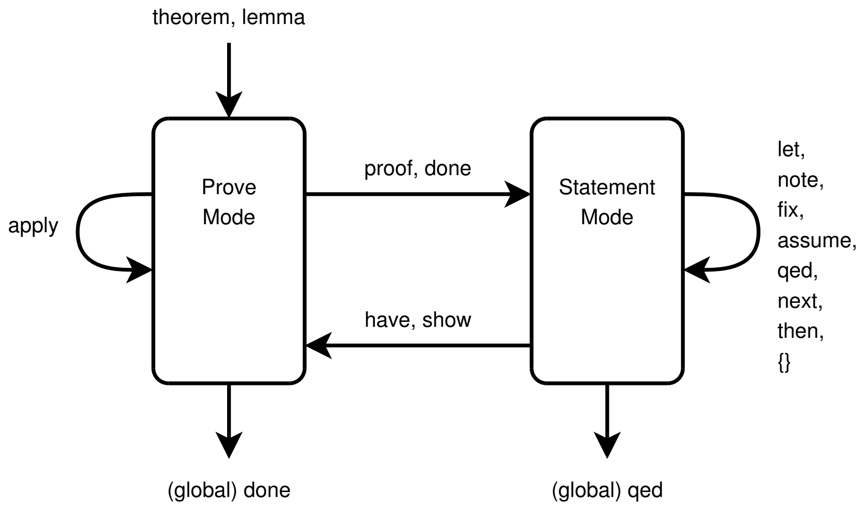


Figure 2.4: *The basic Isar state machine transitions for parsing a proof.*

While the Isar language makes it easier to read proofs, supports abbreviations, and simplifies forward reasoning, the language itself can be difficult to learn. Moreover, the lack of proper support in exploring the application of tactics makes writing proofs slower and more arduous. The reason for this difficulty in exploration is that to examine the effect of applying a tactic, a user must take a backward, procedural proof step. If this solves the goal, then the user can usually replace the backward step with a single tactic justifying the proved statement. However, if the tactic fails to solve the goal, to maintain readability, the user either needs to modify the tactic and try again, or remove the tactic application state and prove an intermediate result in a forward manner. Only if the user is able to second guess the level of automation available can they directly express the intermediate steps. As a result, it is common for users to explore and find a proof using a procedural style, working backward from the goal, and then rewrite the proof in Isar's structured forward style. The lack of proper tools for exploring Isar style proof is one of central issues that we try to address using proof planning.

2.4 Rippling

The term *rippling* was coined by Aubin who made the observation that during many inductive proofs parts the terms introduced into the goal by induction are moved or removed to eventually allow the induction hypothesis to be applied. Bundy turned this notion around by proposing that rippling can be used as a heuristic to guide inductive proofs [17]. Many variations of rippling have since been proposed [19, 49, 87, 96] and the strategy has found several other applications [22, 64, 66] and been implemented in a few different systems [9, 32, 40, 50, 87]. This section gives a brief outline of rippling and introduces the key terminology. We give further details and describe our characterisation in chapter 7.

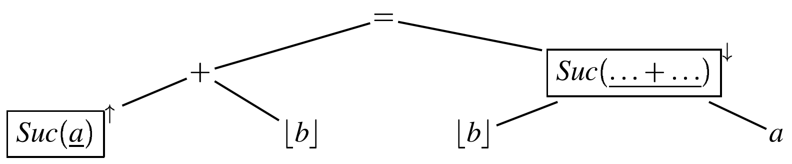
While there are many variations of rippling, the central principle is to remove the differences between all or part of a goal and some defined *skeleton* constructed from a theorem or assumption, typically an inductive hypothesis. Through the removal of this difference, the assumption or theorem that was employed to construct the skeleton can then be used to either solve the goal, a process known as *strong fertilisation*, or failing that it can be used to rewrite a subterm in the goal, which is called *weak fertilisation*. Rippling can thus be seen as guiding rewriting towards fertilisation.

The difference removal is facilitated by specialised annotations on the goal known as *wave fronts*, *wave holes*, and *sinks*. More specifically, wave fronts indicate difference between the skeleton and the goal while wave holes identify subterms inside the wave fronts that are similar to parts of the goal. Sinks, for their part, indicate *positions* in the skeleton that correspond to universally quantified variables. Fertilisation is possible when the wave fronts have been removed from a subterm matching the skeleton and/or placed in sinks appropriately. Thus, there are two directions rippling can pursue:

rippling-out: tries to remove a difference or to move it to the top of the term tree. Eventually this will allow allowing strong fertilisation, or weak fertilisation in a subterm.

rippling-in: tries to move a difference into a sink which would allow it to be matched by the corresponding universally quantified variable.

As an example consider the skeleton $\forall b.\ a+b=b+a$, then the term $Suc(a)+b=Suc(b+a)$ can be annotated as: $\boxed{Suc(\underline{a})}^{\uparrow} + \lfloor b \rfloor = \boxed{Suc(\underline{\lfloor b \rfloor + a})}^{\downarrow}$. The boxes indicate wave fronts, and the underlined subterms are wave holes. The up and down arrows indicate rippling outward and inward respectively, and the annotations $\lfloor b \rfloor$ indicate that the subterm b is at the location of a sink.


=
+
$\boxed{Suc(\underline{a})}^{\uparrow}$
$\lfloor b \rfloor$
$\boxed{Suc(\underline{\ldots + \ldots})}^{\downarrow}$
$\lfloor b \rfloor$
a

Out	In
0	0
0	1
1	0

Figure 2.5: *The annotated term tree, outward measure and inward measure for the goal $Suc(a) + b = Suc(b + a)$ with respect to the skeleton $\forall b.\ a + b = b + a$.*

To provide rippling with a direction and to ensure its termination, a measure is used that decreases each time the goal is rewritten. The measure is a pair of lists of natural numbers that indicates the number of wave fronts (outward and inward) at each depth in the skeleton term. The outward list is obtained by counting the number of outward wave fronts from leaf to root and the inward list by tallying the inward ones from root to leaf. For the above example, this is calculated as shown in Figure 2.5, which results in the measure $([1,0,0],[0,1,0])$.

Such measures are compared lexicographically as if they were a single list starting with the outward elements. For example, the above goal can be rewritten to $\boxed{Suc(\underline{a + \lfloor b \rfloor})}^{\uparrow} = \boxed{Suc(\underline{\lfloor b \rfloor + a})}^{\downarrow}$, using the defining equation for addition $Suc(X) + Y = Suc(X + Y)$. This new goal has a measure of $([0,1,0],[0,1,0])$ which decreases with respect to the previous measure and so the rewrite expresses a valid ripple step. In contrast, it would be invalid to ripple to $\boxed{Suc(\underline{a})}^{\uparrow} + \lfloor b \rfloor = \lfloor b \rfloor + \boxed{Suc(\underline{a})}^{\uparrow}$ as the measure of this new goal increases to $([0,2,0],[0,0,0])$. This restriction allows wave fronts to move from *out* to *in* but not visa-versa.

We now present a simple example of rippling in the domain of Peano arithmetic to solve the step case for an inductive proof of the commutativity of addition ($a + b = b + a$). We will use the following annotated rewrite rules:

$$\boxed{Suc(\underline{X})}^{\uparrow} + Y \Rightarrow \boxed{Suc(\underline{X + Y})}^{\uparrow} \tag{2.1}$$

$$X + \boxed{Suc(\underline{Y})}^{\uparrow} \Rightarrow \boxed{Suc(\underline{X + Y})}^{\uparrow} \tag{2.2}$$

The step case assumes the inductive hypothesis "$\forall x.\ a + x = x + a$", and from this proves "$Suc(a) + b = b + Suc(a)$" for arbitrary but fixed a and b. The hypothesis is used as the skeleton with which to annotate the goal and the following proof can then be carried out as shown in Figure 2.6.

$$
\begin{array}{rcll}
\boxed{Suc(\underline{a})}^{\uparrow} + \lfloor b \rfloor & = & \lfloor b \rfloor + \boxed{Suc(\underline{a})}^{\uparrow} & Measure: \ ([2,0,0],[0,0,0]) \\
 & \downarrow & \text{Ripple using: 2.1} & \\
\boxed{Suc(\underline{a + \lfloor b \rfloor})}^{\uparrow} & = & \lfloor b \rfloor + \boxed{Suc(\underline{a})}^{\uparrow} & Measure: \ ([1,1,0],[0,0,0]) \\
 & \downarrow & \text{Ripple using: 2.2} & \\
\boxed{Suc(\underline{a + \lfloor b \rfloor})}^{\uparrow} & = & \boxed{Suc(\underline{\lfloor b \rfloor + a})}^{\uparrow} & Measure: \ ([0,2,0],[0,0,0]) \\
 & \downarrow & \text{Weak fertilise using the inductive hypothesis.} & \\
Suc(b+a) & = & Suc(b+a) & \\
 & \square & \text{Solved by reflexivity} &
\end{array}
$$

Figure 2.6: *The rippling proof of the step case of "$Suc(a)+b = Suc(b)+a$" with the induction hypothesis "$\forall b.\ a+b = b+a$".*

This proof shows how rippling can guide rewriting to move the differences to the top of the term tree to allow weak fertilisation, and finally to prove the goal. While everything works in this proof, it often happens that at some point during rippling, no wave rules apply and the goal cannot be fertilised. In this situation, the goal is said to be *blocked*. This typically indicates that some backtracking is required, or that a lemma is needed.

Further Terminology

The notion of an annotated term introduces some further terminology to rippling. The *erasure* of an annotated term is simply the annotated term without annotations. For example, the erasure of $f(\boxed{g(\underline{x}, \lfloor y \rfloor)}^{\uparrow})$ is $f(g(x,y))$. We sometimes want to consider annotations without a direction. Such terms are said to have *undirected annotations*. For example, the undirected version of the above annotated term is $f(\boxed{g(\underline{x}, \lfloor y \rfloor)})$.

The Inward Wave Front Sink Restriction

One restriction often added to rippling is to disallow inward wave fronts over subterms that do not contain a sink. For example, this would exclude the possibility of rewriting the goal

$\boxed{Suc(\underline{a})}^{\uparrow} + \lfloor b \rfloor = \boxed{Suc(\underline{\lfloor b \rfloor} + a)}^{\uparrow}$ to $\boxed{Suc(\underline{a})}^{\uparrow} + \lfloor b \rfloor = \lfloor b \rfloor + \boxed{Suc(\underline{a})}^{\downarrow}$, which would normally be permissible as it decreases the measure from $([1,1,0],[0,0,0])$ to $([0,1,0],[0,1,0])$.

The motivation for this restriction is that if a sink does not occur in the subterm under the wave front, then there is no way to ripple the wave front inward to the location of a universally quantified variable. Thus rippling is *unlikely* to allow fertilisation. In theory, it is sometimes still possible achieve fertilisation as some rules may simply remove inward wave fonts. However, the argument for the restriction on rippling inward to non-sinks is that it is not worthwhile because it significantly increases the search space while yielding few additional proofs. We justify this heuristic restriction by experimentation in chapter 10.

2.5 Proof Planning

Proof planning was first introduced by Bundy [14,20] and first implemented in the Clam system [15]. It tries to capture common patterns of reasoning for families of similar proofs in what we shall call *reasoning techniques*. Proof planning involves searching through the ways that these encoded techniques can be applied to conjectures. Applying a technique derives an abstract description of the proof known as a *proof plan*. This is typically a tactic tree or compound tactic that can be executed in a theorem prover to derive a fully formal proof.

Reasoning techniques are typically presented using diagrams that show the behaviour using nested boxes and which abstractly describe the search space. For example, a simple inductive proving technique based on rippling is shown in Figure 2.7. This technique tries to solve a given goal by simplification or by applying an induction principle followed by symbolic evaluation on the base cases and rippling on the step cases. When rippling has finished, the technique tries to fertilise the goal and then solve it by simplification.

Both proof planning and the execution of the resulting proof plan can fail. The execution process can fail when the effect of applying a tactic does not give the expected result. For example, if a tactic failed to solve a subgoal that the proof planner expected it to, then the proof plan would not result in a complete proof and part or all of the proof plan would have to be changed. The proof planning process can fail when there is no reasoning technique applicable to the current problem.

Experience in automated inductive theorem proving has shown that failed proof attempts can often help to find a complete proof [12]. This has motivated the development of techniques based on planning critics that try to make productive use of failure at the level of proof plan-

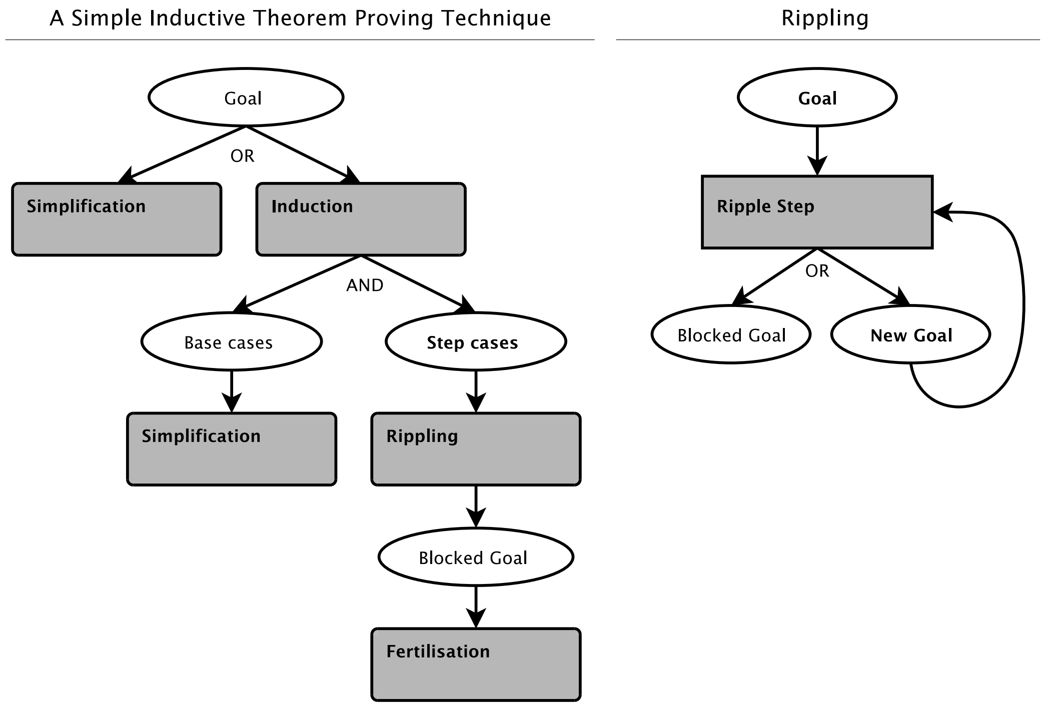


Figure 2.7: *An illustration of an inductive theorem proving technique that uses rippling. The boxes represent techniques and the ovals goals.*

ning [51]. This notion of proof planning critic tries to capture the idea of a patch to a family of failed proof attempts. Such critics are typically triggered by failure in the application of a reasoning technique. They can then apply a transformation to the existing partial proof plan in an attempt to allow the failed reasoning technique to be applied successfully. For example, if a reasoning technique requires the goal to contain $Suc(a+b)$ but the goal contains $a+Suc(b)$ then a critic may suggest to prove the lemma $a+Suc(b) = Suc(a+b)$ first.

In the next section we briefly describe the Clam family of proof planners which possess a critics mechanism. We then give a brief overview of the Omega system and its knowledge based approach to proof planning. We discuss particular issues concerning the approaches that these systems embody in more detail as they come up throughout the rest of this thesis.

The Clam family of proof planners

The Clam family of proof planners refers to the Clam and λClam proof planners as well as to the collection of alternative versions that have existed at various times. This family of proof planners encode mathematical knowledge in *methods*. These are interpreted and declarative

Input The input patterns to which this method can be applied.
Preconditions Other conditions which hold for the method to apply.
Postconditions (or Effects) Conditions which must hold after the method has been applied.
Output Output patterns representing the subgoals generated.
Tactic The name and arguments (if any) of the tactic which constructs the piece of object-level proof corresponding to this method.

Figure 2.8: *The slots that make up a λClam method.*

objects, described in Figure 2.8, that consist of several slots representing components of a mathematical technique.

Clam is the name of the original proof planner although their are now several variations of this system. They are all written in Prolog which allows them to work in first order logic. The contents of the various slots in a method are thus Prolog expressions. Techniques are encoded in a two-level hierarchy of plan specifications which consisted of methods and sub-methods, where sub-methods have to be explicitly invoked by methods.

Ireland's proof critics have been developed for a modified version of Clam. This is because the representation of the proof plans in the main version of Clam was an ad-hoc Prolog expression. Thus, there was no way to write critics that modified the proof plans. However, the modified version of Clam lacked machinery to structure the methods.

The latest in the Clam family is λClam [32], which was designed to support reasoning in higher order logics. This supports both a structuring of methods as well as the application of proof critics. It is written in λProlog and thus the contents of its methods are higher-order expressions. The λClam system also provides *methodical* expressions as a language for expressing a hierarchy of reasoning techniques in a declarative way [90]. Methodical expressions comprise of methods linked by methodicals. Examples of methodicals that link two methods are the `THEN_METH` that applies the second method to the result of the first method, and the

`ORELSE_METH` that only applies the second method if the first one fails. A methodical expression which contains methodicals is often called a *compound method* in contrast to an *atomic method* which refers to the methods with slots. Methodical expressions provide a mechanism by which methods can be selected and applied to subgoals. It is in this way that the λClam proof planner reduces the search space of applicable methods.

Another feature of the Clam systems is the existence of a number of decision procedures which have been developed by Janičić [55]. These are implemented as compound methods which perform some translation on the goal and then call external procedures to apply the decision procedure. This work is still in under development for λClam. The Omega system [8], described in the next section, also makes use of external decision procedures.

The role of the Clam and λClam proof planners is to interpret the reasoning techniques encoded in the methods and apply them to a problem. For its part, the process of applying an atomic method to a goal involves first matching the input slot against the goal, then checking the preconditions. Assuming the input slot matches the goal and the preconditions are true, then the postconditions and output are generated. For methodical expressions in λClam, the proof planner must decompose them to select and apply the contained atomic methods. Only a depth-first and iterative-deepening search have been developed for the λClam system. We remark that changing the search strategy or adding a new methodical in the Clam family of systems involves modifying the proof planner.

The Omega System

In the previous sections we have described a few general notions of the proof planning paradigm and focused on the Clam family of proof planners. In this section, we review of a knowledge based approach to proof planning as presented by Mellis [71] and implemented in the Omega system. This approach has a different interpretation of methods which is illustrated in Figure 2.9. Methods in the Omega proof planner are part of a hierarchical theory knowledge base that also contains constraint solvers, control rules and other domain specific knowledge including axioms, theorems and definitions. The knowledge base is organised into theories such as *group theory* and *dense linear logic*.

The Omega style of proof planning does not have a methodical language. Instead, Omega uses control rules from the knowledge base to reduce the search space of methods that can be applied to open goals. These are separate from the method definitions. This separation of control rules from methods characterises a difference in the notion of what a method represents

Premises & Conclusions These are a set of sequents. Each sequent has some additional information associated with it, in particular it is given a label and a justification. The premise-set and the conclusion-set are each labelled as either an add or delete effect, this defines the direction of proof performed by this operator. If the premises are labelled as an add-effect and the conclusion is labelled as a delete-effect then the operator performs backward proof. If the add and delete effects are the other way around then the method is performing forward proof.
Application conditions These are written in some meta language. Their function is to restrict the applicability of the operator and instantiations of parameters.
Proof Schema These are a set of sequents annotated with a label and a justification. The justification is OPEN if it needs further planning, and is otherwise some specification that defines how the sequent can be derived from the state.

Figure 2.9: *The slots that make up an Omega method.*

between the Omega and Clam proof planners. Another difference between Omega and Clam is that the Omega system does not have a notion of proof critics and does not possess rippling methods.

One of the key characteristics of the Omega system is its integrated use of other tools in its knowledge base to provide the justifications for method applications. For example Omega can use a constraint solver to justify portions of a proof. This has been successfully used to find proof plans for problems in the limit domain [72].

The HOL/Clam and FTL/λClam interfaces

Although proof planning looks like a promising approach to proof automation, it is still to be widely used for interactive theorem proving.

In this section we examine how proof planners have been used in conjunction with external object level theorem provers. When proof planning was first implemented in the Oyster/Clam system, Clam performed proof planning and Oyster executed the proof plans [21]. More recently Boulton et al. [9] developed an interface between Clam and HOL. The motivations for their project included providing more automation in the HOL system and investigating the use of proof planning for software and hardware verification.

The interface is implemented as a tactic called from HOL that first translates the goal to the syntax of Clam, and then calls Clam to try and find a proof plan for the translated goal. The proof plan given back to HOL is then translated into a tactic that HOL can apply to the goal. A number of correspondences had to be maintained for this process to work. In particular rules, definitions and induction schemes as well as the correspondence between terms had to be maintained between the two systems.

Boulton and Slind also developed a more complicated protocol that allows a communication between the two systems using iterative 'dialogues' [94]. They believe that this interprocess communication infrastructure is usable for a variety of other systems. For the HOL and Clam systems, the more complicated protocol allowed them to share more information than their initial implementation. The interface between these two systems has enabled HOL to use Clam to find proof plans that it can execute, resulting in the successful automation of proofs for a number of problems.

In recent work, Castellini has developed an interface between λClam and his FTL proof checker [23,25]. This approach is used for proof planning in the domain of first-order temporal logic (FOTL) and embeds FOTL in λClam methods. This involves providing a method for each inference rule in FTL. Proof plans found by λClam can then be executed to derive fully formal proofs in the proof checker.

As both the FTL theorem prover and λClam are implemented in λ*Prolog*, this made the interface between the two systems relatively easy. The proof plan associated with the application of these methods simply contains a tactic that applies the associated inference rule. More complex methods, like case splitting, were more difficult to translate to FTL tactics. Some automatic tactics exist in FTL, but they have not been used, although they may be in future work.

In contrast to the interface between Clam and HOL, where HOL used Clam as a black box to help prove conjectures, the interaction between FTL and λClam is one where λClam contains the FTL theorem prover as a module which is used to execute proof plans. The theorem prover is called from within λClam which involves translating the goal from the embedded first order temporal logic in λClam to the equivalent in FTL, and also translating the resulting proof plan into an FTL tactic which can then be applied to the goal. Castellini's work gives an example of how a logic can be embedded in a proof planner, enabling proof planning in the embedded logic.

Interestingly both the FTL/λClam and the HOL/Clam interfaces required a significant correspondence between the proof planner and the target proof checking system. This provides support for our suggested implementation of a proof planner within an interactive theorem prover, as this would remove the need for such a correspondence.

2.6 Terminology

In order to avoid confusion, we now clarify the terminology used throughout this thesis:

Technique - the informal notion of a common pattern of reasoning.

Encoded technique - some encoding of a technique within a system.

Rule of inference - a logical rule expressed within a formal system. These typically refer to the axioms used in implemented proof systems.

Tactic - a function that combines a number of rules of inference in an implemented system.

Proof script - a file containing a textual representation of the application of tactics that can be executed by a theorem prover. Typically this is a sequence of commands that apply tactics, although it can also be a structured text such as Mizar or Isar proof scripts.

Interpretable - refers to a characterisation that is interpreted in a programming language. This allows it to be examined and manipulated by programs written in the programming language. Examples include, datatypes in ML, and lists in Prolog.

Declarative - is used in two senses. Firstly, we use it to refer to an expression that has a logical interpretation, as is used in the logic programming literature. Secondly, following the literature on proof languages, it is used to refer to a proof script that is intelligible without having to examine the steps by re-executing the script. This is typically done by explicitly representing the intermediate goals in the proof.

Method - a declarative and interpretable description of a tactic.

Proof plan - an interpretable description of a proof script.

Proof planning - the process of constructing a proof plan.

Chapter 3

Observational Proof Planning

In this chapter we examine approaches to the encoding of reasoning techniques. We give an exposition of our *observational style* for expressing common patterns of reasoning and introduce a basic set of constructs. Our approach is contrasted to both the *functional* and *interpreted* approaches, as embodied by Isabelle tactics and *Clam* methods respectively. We also consider how to encode techniques in a logic-independent fashion, and examine the representation and generation of proof plans.

Our general goal is to develop a framework that makes it 'easy' to encode complex reasoning techniques, such as rippling and its proof critics, and that can produce Isar style proof scripts. While our formulation of proof planning has been implemented for Isabelle, the general mechanism described in this chapter is independent and could be transferred to another system.

3.1 Motivation

Proof planning focuses on providing mechanisms to encode and apply mathematical knowledge. The motivation for such a focus comes from the need for domain specific techniques and the need for an environment in which one can develop and experiment with new machinery for proof automation. These features have been of use to many existing proof planning developments including Maclean's methods for nonstandard analysis [66], those for CCS by Monroy [76], and those for quantified temporal and modal logics by Castellini [23].

For its part, the development of techniques has motivated new proof planning architectures. This has led to the development of different systems which essentially embody different techniques. For example, the proof critics machinery implemented by Ireland et al [51, 52] was in an independent version of the *Clam* proof planner that lacked compound methods. Similarly, there was another version for the afore-mentioned CCS proof methods, and another for coinduction [34]. This trend has continued even recently in a separate version of λ*Clam* which supports the automatic derivation of induction schemes [44].

This divergence of systems and architectures makes it difficult to analyse and compare techniques in a uniform manner. Furthermore, it suggests that the goal of having a single common machinery for implementing different proof planning techniques has not yet been achieved.

Closely related to these problems has been the lack of a generic foundation for proof planning to provide logic-independent machinery while maintaining the ability to execute the generated proof plans. This lack of machinery has contributed to the divergence of proof planners and the absence of a unifying framework for executing the proof plans.

These issues in the development of proof planners provide the motivation for the work in this chapter. In particular, we aim to develop a generic framework for proof planning with techniques that can clearly specify their dependencies on the underlying logic.

3.2 Introduction and Overview

We describe the foundations for a proof planning framework that has been implemented in IsaPlanner, a generic proof planner for the Isabelle proof assistant. We present, in section 3.3, the observational style of encoding techniques, a central theme of our approach and an alternative to both the functional and interpreted styles. The observational style features an extensible language for techniques, lazy evaluation of the reasoning process, and a generic mechanism for techniques to share information.

We present, in section 3.5, a basic language for observational proof planning which provides a platform for writing techniques. These also provide a basis with which the language can be further extended.

While the basic language is independent of the both the meta and object logics of Isabelle, the framework is designed to support the development of domain specific tools. This leads us to describing a methodology to manage the dependency on the object logic and local theories in order to support the generic reasoning techniques, in a style similar to Isabelle's generic tactics. We examine these issues in section 3.6.

The lazy unfolding of techniques generates the search space of possible ways to tackle a problem. Proof planning involves searching this space for a proof plan that solves the goal. We examine the representation of proof plans and their relation to the underlying theorem prover in section 3.7.

3.3 Encoding Techniques

In this section, we consider the question of how to encode problem-solving knowledge in a theorem prover. We analyse two existing approaches categorised into two general styles, namely functional and interpreted. We then examine the characteristics and limitations of these approaches. This forms the basis from which we then develop our observational framework for proof planning.

3.3.1 The Interpreted Style

In this style, techniques are interpretable objects and their application involves an *interpretation function* that is given both an object representing the technique and a proof state. This typically results in a new proof state with techniques to solve the subgoals. This style of encoding techniques is commonly used by proof planners such as the *Clam* systems.

For example, in λ*Clam*, techniques are encoded as *proof methods* which are either:

- compound: they are built from methodicals containing sub-methods. The methodicals are essentially treated as datatype constructors and as such can be analysed to retrieve their constituent methods.
- atomic: they are made up of a number of 'slots' which contain predicates, such as a precondition slot containing predicates that must be satisfied in order for the method to be applicable.

Applying a compound proof method involves its traversal to retrieve and then apply the first atomic sub-method. Applying an atomic method involves examining and satisfying the various higher order predicates. Thus the interpretation function can be seen as defining an operational semantics for the method language. Such a semantics for λ*Clam* has has been described by Richardson et al [90].

The central feature of this approach is that encoded techniques can be analysed. This has been used in the application of proof critics which analyse the failure to apply an atomic method and suggest a patch to the proof attempt [52].

3.3.2 The Functional Style

In this style, techniques are functions on *proof states*. For example, Edinburgh LCF tactics are functions that reduce a goal to a list of subgoals. Applying a technique to a problem simply involves applying the function representing the technique to a proof state that expresses the problem. We will refer to techniques encoded in the functional style as *tactics*.

The crucial features of this style are that tactics cannot be examined - they can only be applied in an opaque way - and that they can be combined using higher order functions called *tacticals*. For example, in Isabelle, the tactical `THEN` can be defined in the following way:

```
THEN(r1,r2) = flatten o (map r2) o r1
```

where `o` is function composition, and `map` and `flatten` are defined in the obvious way for lazy lists. This tactical is given two sub-tactics and applies the second one to each proof state resulting from the application of the first.

The functional style represents an extreme position in the encoding of techniques where all the work is done by the tactic implementing the pattern of reasoning. Initially, it may seem that the technique developer in this approach is at a disadvantage compared with the interpreted style, where the framework interprets methods thus requiring them to express less. However, because the functional style supports development of new tacticals and other functions to perform common operations, it can arbitrarily lessen the load for a tactic developer. For example, Isabelle already includes tacticals that perform various kinds of search over tactic applications, including depth first, breadth first and best first.

3.3.3 Representing Choice

For many techniques, from Ireland's proof critics to resolution based proof procedures, search is a key factor. Additionally, it is useful to be able to express techniques that can be applied in a number of ways. For example, applying the equation $(Suc\ a) + b = Suc(a + b)$ to a goal $P\ ((Suc\ x) + ((Suc\ y) + z))$ could result in $P\ (Suc(x + ((Suc\ y) + z)))$ or $P\ ((Suc\ x) + (Suc(y + z)))$. A language to encode techniques needs to be able to express such choice.

In the functional style, tactics can be represented by functions that result in a 'collection', such as a list, of new proof states. These capture the different ways the tactic can be applied. For example, Isabelle uses lazy lists to express the result of a tactic application. Another way choice occurs in tactic languages is through tacticals. For example, the `ORELSE` tactical is given two tactics and results in a new tactic that tries to apply the first, and if it fails, applies the second.

Similarly, in the interpreted style, there are two analogous ways of expressing choice. Firstly, methods can often have their preconditions instantiated in different ways. Each possible instantiation then corresponds to a possible way the method can be applied. Secondly, there is an `OR` methodical which is treated by the interpretation mechanism and can result in the same behaviour as the equivalent tactical.

3.3.4 Information Sharing

Developing mechanisms to automate proof frequently involves extending existing techniques. Such extensions often need to make use of information that is not part of the object level goal. For example, proof critics added to rippling often need the annotation information. Even within some techniques there is a need to share information between different steps in the proof attempt. Expressing the technique in a modular extensible fashion then requires some approach to management of shared information. In particular, we would like to be able to develop proof critics independently from the technique to which they are applied.

More generally, shared information can be local to a branch of the search space, such as the annotations for rippling, or globally shared across alternative branches, such as a cache of proved lemmas. One way to share information locally, and that is frequently employed in the functional style, is to pass the information as a parameter between techniques. Unfortunately, a problem with this approach is that applying one technique after another makes it impossible for the latter technique to access the information used by the former. This inhibits developers from extending the behaviour of existing techniques.

For example, consider the followings technique encoded in the functional style:

1. An initial technique $f = \lambda a. \ldots$, where a is a parameter, such as an initial annotation.

2. A second technique, $g = \ldots f\, x \ldots$, defined in terms of f.

This makes it impossible express a third technique of the form g `THEN` h that allows h to use the value of the parameter given to f by g. We need a richer mechanism for holding information that results from the application of a technique.

A second way to share information, which solves this problem, is to place it into the object level term structure of the proof state, in the style of Slind and Norrish [95]. This is sufficiently expressive as arbitrary information can be placed in the goal. However, this holds the information in an often inefficient and arduous form that 'pollutes' the proof state. Furthermore, there is a danger that proof tools may inadvertently damage the held information.

A third approach that is closely related but arguably 'cleaner', has been developed by Hutter [48] and involves using a modified logical calculus that can carry strategic information. While this provides a powerful way of maintaining meta-logical knowledge attached to terms, such as the annotations used during rippling, it does not help with storing information that is not specific to parts of the term, such as previously proved conjectures. Additionally, it does not provide a way of holding information across branches in the search space.

A fourth approach, employed by the *λClam* system [32], is to use the notion of context that comes from their underlying Lambda Prolog. This can provide a form of information that can be shared across *and*-branches in the search space. However, upon Prolog backtracking the information is removed from the context and thus this mechanism is unsuitable for sharing information between *or*-branches.

In section 3.4.1, we present a novel solution that extends our notion of proof state in a extra-logical fashion by including an extensible table to hold contextual information. This provides a flexible mechanism to share information across both *and*- and *or*-branches in the search space.

3.3.5 Expressivity

An initial observation with regard to the expressivity of the language for encoding techniques is that if this language is sufficiently expressive (for example, Turing-complete), then any technique can be encoded no matter what style is used. However the language does have a significant effect on the clarity, modularity, and reuse of the encoded techniques. Furthermore, it may be that different constructs are useful within different domains. Thus we believe that the key features of a language for encoding techniques concern the convenience of expressing the common patterns of reasoning.

A significant advantage of the interpreted style is that the interpretation function can 'observe' the proof process in terms of the underlying techniques and in particular it can observe failure in the application of a method. This supports the use of proof critics by providing them with information that they can use to examine failed proof attempts and propose patches that might allow proof planning to continue. Thus we see that the interpreted approach benefits from a richer notion of failure and the ability to introspect on why this failure might have occurred. The functional approach is unable to express proof critics in this way because techniques cannot be examined to ascertain *why* failure occurred. Furthermore, the eager evaluation (of tactics) obscures *where* in the tactic the failure occurs.

One way a proof-critic-like mechanism can be implemented in the functional style is through the raising and handling of exceptions. The information that would normally be provided to the proof critic must now be passed in the exception, and the critic itself is expressed as an exception handler. In this approach critics must be provided to catch all exceptions otherwise proof search will be inadvertently stopped by the raising of an unexpected exception. Similarly, in the interpreted style, failure must be handled by interpretation function. However, the interpreted approach benefits from having a default way to handle the failure of a method, which is to backtrack.

3.3.6 Modularity and Reuse

Providing tools for expressing techniques in a modular way is essential to easing their development. In the functional style, modularity and reuse is facilitated by the programming language in which the techniques are expressed. For example, in chapter 9 we describe a flexible modular implementation of rippling that uses ML functors. Using modular programming facilities of the language in which techniques are written is also available to interpreted style.

One approach available to the interpreted style, but not the functional style, is to define new techniques as manipulations of existing ones. However, in our view it is not clear that this presents a significant benefit when we consider that techniques can also be parameterised. It might be argued that the advantage of such a interpreted reading of techniques requires less to be known in advance. In particular, the developer does not have to guess which aspects of a technique need to be parameterised over. This might then simplify the parameters.

We remark that we were unable to find an encoded technique that uses this approach in a practical manner. Thus, even though this is an interesting approach to defining modularity, it is at present unclear whether it is of practical benefit.

A disadvantage of defining techniques in terms of manipulations of other techniques is that it requires having the ‘right’ language for expressing techniques. Modification of this language will break existing techniques as well as any functions that modify them. In our view, given the history of changes to technique languages in systems such as *Clam* and Omega, it is not clear which is the ‘right’ language. Thus we would argue that such an approach should not be considered until there is sufficient evidence that the language will not be further modified.

3.3.7 Extensibility of the Technique Language

It is hard to know which basic elements will facilitate the convenient expression of a technique, let alone what the specifics of a technique language should be. The various changes to proof planning architectures and modifications to the mechanisms for expressing techniques support this view. Furthermore, it may be that some constructs are useful to only specific domains. This motivates having an extensible language for encoding techniques.

In the interpreted style, the language for encoding techniques is defined by the interpretation mechanism. In particular, there are a fixed number of language constructs (methodicals) that allow new techniques to be created from old ones. While these constructs can be parameterised, the language is fixed by what the interpretation function can comprehend. Thus, the interpreted style limits the extensibility of the reasoning technique language, while the functional style is extensible.

In the functional style, the technique language is simply the programming language used to write tactics and tacticals. This enables any function of the correct type to be used as part of the language. Domain specific constructs can then be defined locally and do not need to be supported where they are inconvenient. This approach also supports experimentation with the basic constructs used to express techniques that may, in future work, lead to a clear specification of a technique language for the interpreted approach.

3.3.8 The Construction of Proof Plans

The functional style of encoding techniques has typically been used to write tactics that are responsible for only the derivation of *calculus level* proofs. In contrast to this, the interpreted style has chiefly been used to construct *proof plans*. However, we believe this distinction between the use of functional and interpreted styles is only historical. In support of this, we observe that calculus level operations, including primitive inference rules, are often expressed in interpreted methods. For example, the `imp_i` method of λ*Clam* simply performs implication introduction. Furthermore, the functional style of encoding techniques can also be used to construct proof plans by simply using 'tactics' that modify a representation of an intermediate proof plan rather than a proof state. Thus both styles can be used to express either calculus level proof operations, or proof planning operators.

Viewing both styles as encoding knowledge within a proof planner, there is still a significant difference in the way these approaches construct proof plans. In the interpreted style, the proof plan is constructed by the interpretation function. For example, consider a compound method in λ*Clam* of the form "$(m_1\ \texttt{OR}_\texttt{M}\ m_2)\ \texttt{THEN}_\texttt{M}\ m_3$", where $\texttt{THEN}_\texttt{M}$ and $\texttt{OR}_\texttt{M}$ are methodicals. If m_1 is not applicable, the interpretation function will then try to apply m_2 followed by m_3. In this situation, the resulting proof plan would be "$m_2\theta$ `then` $m_3\theta$", where "`then`" is an interpreted description of the tactical with the same name, and θ is an instantiation of the variables in the methods. In this way, the proof plan is a 'trace' of the proof planners successful interpretation of the method.

In contrast to this, the functional style of proof planning, as suggested above, uses tactics to express modifications to the proof plan. This captures the editing of the proof plan directly in tactics without the need for an interpretation mechanism. Returning to the above example, a tactic that has the same behaviour as the above method would be expressed as "$(t_1\ \texttt{OR}_\texttt{T}\ t_2)\ \texttt{THEN}_\texttt{T}\ t_3$", where t_i tries to add the interpreted description m_i to the proof plan. The tactical $\texttt{THEN}_\texttt{T}$ simply applies the first tactic followed by the second one, and $\texttt{OR}_\texttt{T}$ applies the first tactic, and if it fails, tries to apply the second tactic. In this way, tactics and tacticals in the functional style capture the behaviour of the interpretation function in the interpreted style.

One of the features of proof planning in λ*Clam* is that the proof plans contain goals with associated methods. This is used to delay a proof attempt but remember the method being used for that subgoal. This allows a proof plans to function as both an agenda of things to be done in order to complete the proof, as well as a high level description of the proof attempt. Because methods have a natural interpreted - and usually declarative - reading, they can easily be textually described. This presents a problem for a purely functional approach. A solution to this problem is to pair each open goal with a tactic and an interpreted component, such as a string, that describes that tactic.

3.3.9 Debugging

Developing techniques for automated theorem proving is an error prone and difficult research task. One of the reasons for this is that it is hard to analyse a technique's application. This is needed when attempting to examine where and why a technique fails. This difficulty motivates the need for machinery to *debug* and explore their application. While debuggers exist for some programming languages and for the underlying machine code which is executed, these do not provide a suitable level of abstraction for debugging techniques.

Having a mechanism to examine the application of techniques, including failure, is of particular interest in the development and investigation of new proof critics. Such a mechanism can provide the user with some initial insight into how a new critic might work.

Another motivation for such machinery comes from the ACL2 user community, where significant time is spent analysing failed proof attempts in order to find suitable lemmas that will help the automatic proof process. Flexible machinery to examine the failed proof attempt would help in this process.

An advantage of the interpreted style is that a step-by-step interpretation function can be defined which allows a user to trace through the application of a method. This facilitates the debugging of encoded reasoning techniques. However, it does require a more complex interpretation mechanism. Furthermore, it may be difficult to make the step-by-step interpreter behave in the same way as an automatic version. Any differences in the behaviour would make debugging difficult.

In the functional style, the problem is more severe, tactics are evaluated in an eager fashion which makes their debugging more difficult. In particular, for large tactics it can be difficult to tell at what point failure occurs. Debuggers built into the execution environment can be used, but these tend to provide much more detail than is useful to the casual user or even the technique developer and, as such, complicate the process of debugging.

3.4 The Observational Style

Our observational style of proof planning is based on the notion of a *reasoning process* which is split into a series of ‘snapshots’ called *reasoning states*. Techniques are encoded as functions from a reasoning state to a set of reasoning states, where the resulting set represents the possible ways of applying the technique. Each reasoning state contains an optional *continuation* technique which is either “*None*” or “*Some r*” where *r* is the next reasoning technique to be applied. This allows each state with a continuation to be *unfolded* by applying its continuation technique to itself. Thus reasoning is performed by unfolding the search space in a lazy fashion. The language of techniques, consisting essentially of functions, remains extensible.

Each reasoning state contains a *proof plan*, which is modified and extended by techniques as the reasoning process unfolds. In Figure 3.1, we illustrate the application of a reasoning technique that uses Isabelle’s induction tactic followed by the simplification tactic. This is encoded as a function that adds the induction tactic to the current partial proof plan and sets the continuation to be a function that adds the simplification tactic to the proof plan. After this, there is no continuation and thus no further states in the search space.

In the observational style, the editing of proof plans is done in a similar way to the functional style, namely by modifying of the proof plan within the technique. To facilitate this,

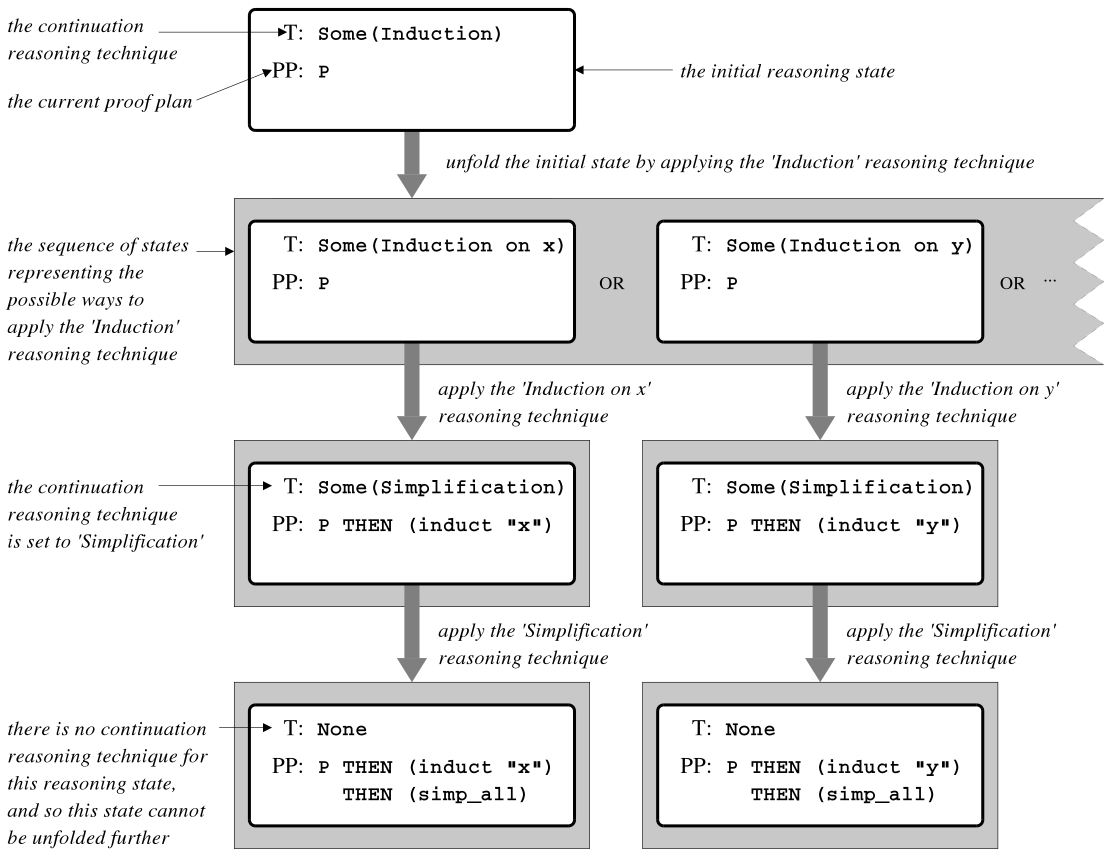

the continuation reasoning technique
the current proof plan
T: Some(Induction)
PP: P
the initial reasoning state
unfold the initial state by applying the 'Induction' reasoning technique
the sequence of states representing the possible ways to apply the 'Induction' reasoning technique
T: Some(Induction on x)
PP: P
OR
T: Some(Induction on y)
PP: P
OR ...
apply the 'Induction on x' reasoning technique
apply the 'Induction on y' reasoning technique
the continuation reasoning technique is set to 'Simplification'
T: Some(Simplification)
PP: P THEN (induct "x")
T: Some(Simplification)
PP: P THEN (induct "y")
apply the 'Simplification' reasoning technique
apply the 'Simplification' reasoning technique
there is no continuation reasoning technique for this reasoning state, and so this state cannot be unfolded further
T: None
PP: P THEN (induct "x") THEN (simp_all)
T: None
PP: P THEN (induct "y") THEN (simp_all)

Figure 3.1: *A reasoning technique that creates a proof plan with Isabelle's induction tactic followed by the simplification tactic.*

common manipulations are provided within the language for writing techniques. However, in contrast to the interpreted style, the elements of the language can be extended. In this way, the observational style can be thought of as providing a dynamic extensible interpretation function.

3.4.1 Contextual Information

To facilitate the sharing of information between techniques, each reasoning state also contains a table of *contextual information* which can be modified during proof planning. This captures any knowledge that might be applicable to the current proof process. Examples of such information include the measure(s) used during rippling and a cache holding the result of proof attempts on conjectured lemmas. One of the main motivations for contextual information is to facilitate analysis of failed proof attempts. For example, Ireland's proof critics attached to rippling make use of the annotations [52].

Contextual information can be *local* to a branch of the search space, such as the annotations and measures which are attached to the current goal term during rippling. It can also be *global* - of importance beyond the local branch of the search space - such as the aforementioned caching of the result of earlier proof attempts.

Contextual information can also be of different *kinds*, which can themselves be theory dependent. For example, information for the conjecture cache would be applicable to any theory and of a different kind to that used for the rippling annotations which may be restricted to theories that support rippling. Making contextual information generic is an issue in the methodology of their development, and is discussed in section 3.6.

To allows new kinds of information to be added is necessary to support modular development of techniques. In particular, it should be possible to develop a techniques using a new kinds of contextual information without having to modify existing techniques. Simply extending the kind of proof state for each new kind of needed information would, for example, fail to allow old technique to be applicable to proof states with the new kinds of information.

In the λ*Clam* proof planner, local contextual information can be expressed in the post-conditions of methods [32]. Some global information can be held by means of uninstantiated meta-variables. However, because search is performed by Lambda Prolog, backtracking removes information from the context, and thus this use of meta variables is not sufficient to express global information that must be held across branches of the search space.

Note that whilst we have presented our information sharing mechanism as part of the observational style, it could also be used in the functional and interpreted styles. In the next section, we describe proof critics, which can make particular use of contextual information.

3.4.2 Proof Critics

The lazy evaluation of techniques supports identification of *where* failure occurs. To establish *why* failure occurred the observational style, like the functional approach, is still unable to examine a reasoning technique's definition. We do not believe that this is necessarily a loss: within a technique there may be many aspects which we do not wish to consider. For example, when applying a proof critics, efficiency measures (such as the use of memoization) may be of no interest.

To provide proof critics with the information that they can use, we express the characteristics of a technique that we want to be able to examine in the contextual information. This provides a clear separation between the interpreted components which are of use to proof critics, and the more functional ones, such as efficiency measures. Such an approach also helps clarify the logical dependency of a proof critic, as described in section 3.6. We note that a similar solution is used in λ*Clam*, where the computational part of a precondition is written within a predicate, and the predicate's arguments provide the interpreted information that can be used by proof critics.

In interpreted proof planning, the key difference between proof methods and critics is the ability of the latter to modify the proof plan. Proof critics can change any part of the proof plan, whereas a method is added to the proof plan in a way defined by the interpretation function. In λ*Clam*, critics are associated with methods and defined by pre-conditions that specify the kind of failure that triggers the critic.

In the observational style, techniques can modify any part, or all, of a proof plan thus enabling proof critics to be expressed within the technique language. This provides a unified framework for encoding mathematical knowledge. The practical result of this unification is that the language for techniques can be reused for writing proof critic like behaviour which provides us with a number of advantages over the classic view of critics:

- Proof critics can be attached to non-atomic methods, including other critics. This is not possible in the *Clam* family of proof planners, as critics are attached to the preconditions of a method, and non-atomic methods do not have preconditions.

- The application of proof critics can involve proof planning search. In contrast to this, λ*Clam* style critics which can manipulate the proof plan and agenda, cannot involve proof planning in the computation of the critic's effect. This is important if the effect of a proof critic is dependent on some further theorem proving.

In our approach, proof critics can be expressed as functions on reasoning techniques. They typically examine the unfolding of the technique they are applied to, and upon failure, by analysing the contextual information, try to suggest a patch to fix the proof. The process of suggesting a patch typically involves modifying the continuation of the reasoning state so that instead of failing the critic is applied. The critic's application then involves some proof planning search (possibly with nested critic applications) and typically manipulates the proof plan and goal agenda.

3.5 A Basic Language for Observational Proof Planning

The language for techniques in observational proof planning is the language of functions from a reasoning state to a collection of reasoning states. In our implementation, we use lazy lists as these allow the direct expression of results from Isabelle's higher order unification. However, in this presentation of the language we will describe the result of techniques in terms of sets. This simplifies the presentation while not straying far from the implementation. The set representing the result of a technique application expresses the possible ways the technique can be applied.

We note that the technique language, being essentially functions, is extensible. In this section, we present a number of basic elements that can be used to define reasoning techniques. These do not require any contextual information, unlike more complex techniques, such as rippling which we describe in later chapters.

In order to clearly understand the basic elements of the technique language, it is important to note that a reasoning state with no continuation (*None*), signifies the successful completion of the reasoning technique. This is in contrast to a technique that has a continuation which results in the empty set and signifies that it is not applicable.

3.5.1 The Basic Elements

We now describe the basic elements of the reasoning technique language using the standard notations for set theory and lambda calculus.

Notation

We will use a triple of the form (p,c,i) to denote a reasoning state with proof plan p, continuation c and contextual information i. We will also use s and t denote reasoning states, and use $s.p$ to represent the current proof plan of a state s, $s.c$ its continuation, and $s.i$ its contextual information. Thus s abbreviates $(s.p, s.c, s.i)$.

Continuations are either "*None*", or "*Some r*" where r is a reasoning technique. Recall that a reasoning technique, typically denoted by r, is a function from a reasoning state to a set of reasoning states, and thus "$r\ s$" is the technique r applied to the reasoning state s, which is a set of reasoning states. A *leaf state* is a reasoning state with no continuation, that is a state of the form $(p, None, i)$. We will let the symbol f denote a function from a reasoning state to a new reasoning state.

We will explicitly describe the types of the elements in the reasoning technique language in order to clarify the arguments they are given. Polymorphic type variables will be written as ‘a, ‘b, ‘c and so on. Given a type α and a type β, the type of a function from α to β is expressed “$\alpha \rightarrow \beta$”, and their pairing is written “$\alpha \times \beta$”. The type for a reasoning state will be written as `rstate`. As reasoning techniques are functions from a reasoning state to a set of reasoning states, they have type “`rstate` $\rightarrow$ `rstate set`” which we will sometime abbreviate to `rtechn` for clarity. Proof plans have type `pplan` and contextual information has type `cinfo`.

Techniques will be often written in the following way:

$\texttt{ELEMENT\_NAME} : \textit{argument\_types} \rightarrow \texttt{rtechn}$

$\texttt{ELEMENT\_NAME}\ \textit{args}\ s = \textit{some\_set\_description}$

with the abbreviations: `ELEMENT_NAME` is the name of the element; *argument_types* will be the types of the arguments to the element if there are any; *args* is the name of the arguments to the element; s is a the reasoning state part of the resulting reasoning technique; and *some_set_description* is the resulting state set, typically defined in terms of s.

NOTHING

We first consider a simple technique that does not affect the proof plan or the contextual information, but results in a single state with no continuation. This can be expressed as:

$\texttt{NOTHING} : \texttt{rtechn}$

$\texttt{NOTHING}\ s = \{(s.p, None, s.i)\}$

FAIL

Another simple technique is the technique that results in no new states. This reflects Prolog’s fail and is expressed as:

$\texttt{FAIL} : \texttt{rtechn}$

$\texttt{FAIL}\ s = \{\ \}$

PPLANOP

Basic techniques are often defined using functions that modify the proof plan. We can define a simple technique `PPLANOP` that uses a function m to modify the proof plan:

$\texttt{PPLANOP} : (\texttt{pplan} \rightarrow \texttt{pplan}) \rightarrow \texttt{rtechn}$

$\texttt{PPLANOP}\ m\ s = \{(m\ s.p, None, s.i)\}$

This results in a single new state that has a modified proof plan and which does not have a continuation. The details pertaining to the modification are dependent on the representation of the proof plan. In chapter 6 we describe a representation of proof plans based on Isar proof scripts. Using this representation, a commmon application of `PPLANOP` is to add elements to the proof plan. In particular, we 'lift' tactics from the underlying prover to the level of techniques by considering the ways that they can be applied and adding them to the proof plan correspondingly.

OR

A simple way to combine technique is using the infix functional `OR` which takes two reasoning techniques r_1 and r_2 and results in a new technique with the union of states from applying r_1 and those from applying r_2:

$$\mathtt{OR} : \mathtt{rtechn} \times \mathtt{rtechn} \to \mathtt{rtechn}$$
$$(r_1\ \mathtt{OR}\ r_2\ s) = (r_1\ s) \bigcup (r_2\ s)$$

THENF

A useful tool for the definition of further reasoning techniques is the functional `THENF`, illustrated in Figure 3.2, which takes a reasoning technique r, a function on reasoning states f and results in a new technique.

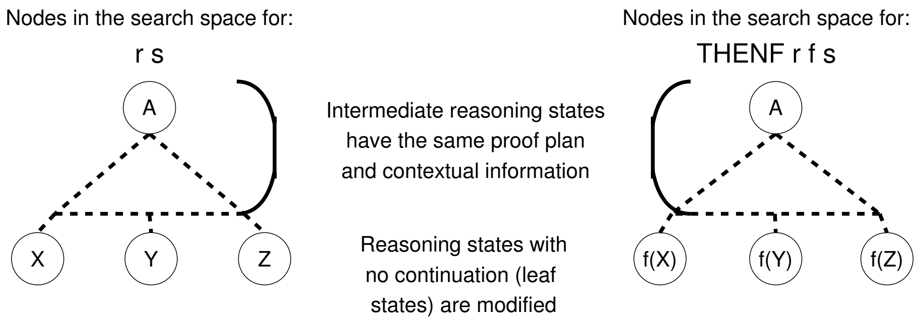

Nodes in the search space for:
r s
A
X
Y
Z
Intermediate reasoning states have the same proof plan and contextual information
Reasoning states with no continuation (leaf states) are modified
Nodes in the search space for:
THENF r f s
A
f(X)
f(Y)
f(Z)

Figure 3.2: *An illustration of the* `THENF` *operation that combines a reasoning technique r with a function on reasoning states f. The states X, Y, and Z are the final state (those with a None continuation) in the application of the technique r to the state s.*

The intuition behind the technique `THENF` $r\ f$ is that is performs r then applies the function

f. More precisely, it has intermediate states that correspond to those the reasoning technique r. In these states the proof plan and contextual information are identical as is their ordering. The difference is at the leaf states in the unfolding of r. In THENF r f, these have the function f applied to them. We define THENF as follows:

THENF r f

$$
\begin{aligned}
&\texttt{THENF} : \texttt{rtechn} \to (\texttt{rstate} \to \texttt{rstate}) \to \texttt{rtechn} \\
&\texttt{THENF}\ r\ f\ s = \\
&\quad \{s_{next} \mid \exists t.\ t \in r\ s \wedge ((t.c = None \wedge s_{next} = f\ t) \vee \\
&\qquad\qquad (t.c = Some\ r' \wedge \\
&\qquad\qquad\quad s_{next} = (t.p, Some\ (\texttt{THENF}\ r'\ f), t.i)))\}
\end{aligned}
$$

Note that this does not involve a direct recursion as the recursive call is lacking an argument. This approach defines the technique's continuation in a lazily but recursive manner.

This also shows an interesting feature of the observational style, namely that a technique can examine another's unfolding. This is expressed by the first condition of the resulting set ($\exists t.\ t \in r\ s \wedge \ldots$) which we use to define the new techniques behaviour in terms of parameter technique r. We use this to examine when a reasoning state in the unfolding of r has no continuation, at which point the function f is applied.

THEN

We define the infix functional THEN which combines two techniques, unfolding the first until there is no continuation and then unfolding the second:

$$
\begin{aligned}
&\texttt{THEN} : \texttt{rtechn} \to \texttt{rtechn} \to \texttt{rtechn} \\
&r_1\ \texttt{THEN}\ r_2 \equiv \texttt{THENF}\ r_1\ (\lambda(p,c,i).(p, Some\ r_2, i))
\end{aligned}
$$

This uses THENF to unfold r_1 until it has no continuation, at which point the continuation is set to being r_2 by the function $(\lambda(p,c,i).(p, Some\ r_2, i))$.

MAP

The functional MAP takes a reasoning technique r and a function on reasoning states f which it applies to each intermediate state in the unfolding of r. Figure 3.3 illustrates the idea.

We define MAP as:

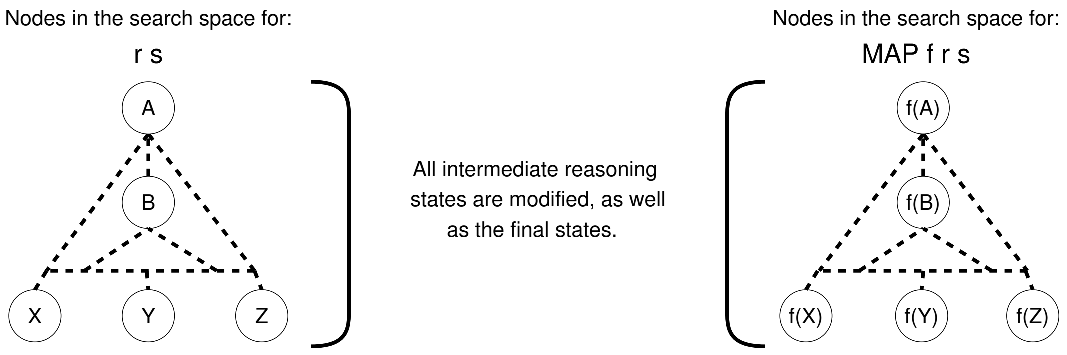

Nodes in the search space for:
r s
A
B
X
Y
Z
All intermediate reasoning states are modified, as well as the final states.
Nodes in the search space for:
MAP f r s
f(A)
f(B)
f(X)
f(Y)
f(Z)

Figure 3.3: *An illustration of the* MAP *operation.*

$$\begin{array}{l}\texttt{MAP} : \texttt{rtechn} \to (\texttt{rstate} \to \texttt{rstate}) \to \texttt{rtechn} \\ \texttt{MAP}\ r\ f\ s = \\ \quad \{s_{next} \mid \exists t.\ t \in r\ s \wedge (((f\ t).c = None \wedge s_{next} = f\ t) \vee \\ \qquad\qquad ((f\ t).c = Some\ r' \wedge s_{next} = ((f\ t).p, Some\ (\texttt{MAP}\ r'\ f), (f\ t).i)))\}\end{array}$$

We note that there is no `MAP` methodical for λ*Clam* that provides the same behaviour[1]. More generally, adding new methods, such an `MAP` to λ*Clam*, requires a modification to the interpretation function in interpreted approach. Furthermore, it is not even possible to define an equivalent tactical in the functional style. This is an example of the additional expressivity gained through the observational style.

FOLD

Another functional that can be defined in the observational style, but which has no equivalent in the functional styles and has not previously been implemented in the interpreted style, is `FOLD` which behaves in a similar way to the `FOLDL` list operation.

This operation provides a way for information to be paired with the reasoning state and used to update the state as a technique is unfolded. We note that, in contrast to the use of our contextual information, it is impossible for another technique to access the `FOLD`'s paired information, except by the use of reference variables. This makes the paired information local to the application of the `FOLD` operation. Figure 3.4 graphically shows this behaviour.

We define `FOLD` as:

[1]There is a `MAP` methodical in λ*Clam* but this applies a method to each resulting subgoal

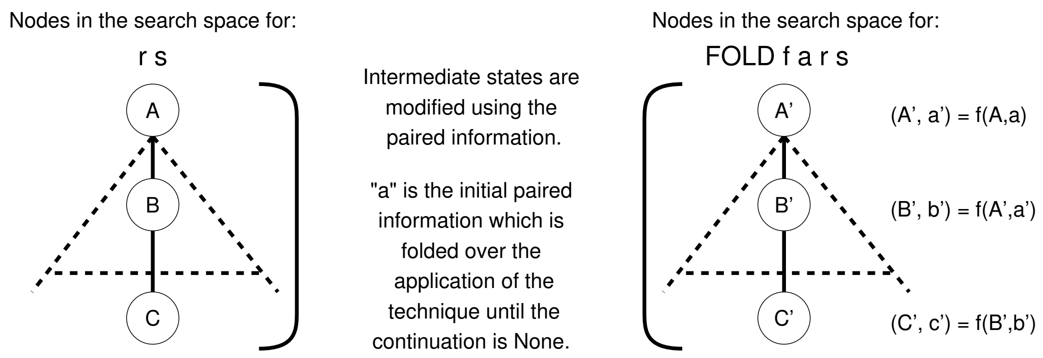

Nodes in the search space for:
r s
A
B
C
Intermediate states are modified using the paired information.
"a" is the initial paired information which is folded over the application of the technique until the continuation is None.
Nodes in the search space for:
FOLD f a r s
A'
B'
C'
(A', a') = f(A,a)
(B', b') = f(A',a')
(C', c') = f(B',b')

Figure 3.4: *An illustration of the* FOLD *operation.*

$$\texttt{FOLD} : (\text{`}\mathtt{a} \to \mathtt{rstate} \to (\mathtt{rstate} \times \text{`}\mathtt{a})) \to \text{`}\mathtt{a} \to \mathtt{rtechn} \to \mathtt{rtechn}$$

$$\texttt{FOLD}\ f\ a\ r\ s =$$

$$\{s_{next} \mid \exists t.\ t \in r\ s \wedge (t', a') = f\ a\ t \wedge ((t'.c = None \wedge s_{next} = t') \vee$$
$$(t'.c = Some\ r' \wedge s_{next} = (t'.p, Some\ (\texttt{FOLD}\ f\ a'\ r'), t'.i)))\}$$

In this definition, the parameter a is the initial value of the paired information and the function f is used to update the paired information and the reasoning state after each step in the unfolding of the technique r.

This operation can be interpreted as a tool for expressing an 'online' analogy to the proof attempt, where the function f is the function expressing the analogy. For example, we have used this to define a functional that applies a technique, caching the intermediate goals and avoiding duplicate proof states. The FOLD functional provides a convenient means to express such operations on techniques.

Similarly to MAP such an operation can be defined in the interpreted style, but requires modifying the interpretation function. In the functional style such an operation cannot be expressed. Again this shows the extra expressivity from the observational style.

REPEAT_UNTIL

The REPEAT_UNTIL functional is analogous to the commonly used methodical and tactical named repeat. It takes a boolean function on reasoning states and a technique. It repeatedly unfolds the technique checking the condition function after each application. When the condition is true the last reasoning state with no continuation is returned. This can be defined as follows:

$$\texttt{REPEAT\_UNTIL} : (\texttt{rstate} \rightarrow \texttt{bool}) \rightarrow \texttt{rtechn} \rightarrow \texttt{rtechn}$$
$$\texttt{REPEAT\_UNTIL}\ f\ r\ s =$$
$$\{s_{next} \mid (f\ s = true\ \wedge\ s_{next} = s) \vee$$
$$(f\ s = false \wedge \exists t.\ t \in r\ s\ \wedge s_{next} = (t.p, Some\ (r\ \texttt{THEN}\ (\texttt{REPEAT\_UNTIL}\ f\ r)), t.i))\}$$

The REPEAT_UNTIL operation conveniently expresses repeated application of a technique. For example, rippling can be expressed as the repeated application of measure decreasing ripple steps where the condition function checks if rippling is blocked.

ENDSPACE

In the functional and interpreted styles, search is separated from the reasoning to the extent that it is not easy to define behaviour of techniques in terms of the search space. However, in the observational style, the lazy evaluation of reasoning allows the search process to be monitored and thus for techniques to be defined in terms of the search space.

A particularly useful application of this is in noting when a technique has failed to solve a goal. It can be useful to cache the failed goals in order to avoid repeated attempts at their proof. To implement this in a generic way requires a general notion of when a technique has failed to solve the goal, and in particular, when the search space of attempts is exhausted.

The basis of such an operation is the ENDSPACE functional, which takes a technique r and a function f on reasoning states. It applies the function f to the last explored reasoning state in the search space of r iff there is no unfolding of r that completes successfully. This is done by maintaining a record (we will use the type *counter*) that holds the number of unexplored states in the unfolding of r. Figure 3.5 illustrates the overall behaviour.

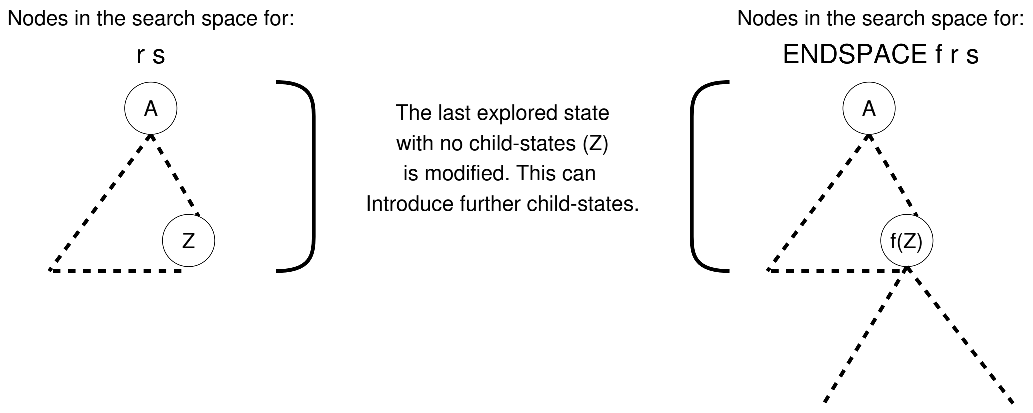

Nodes in the search space for:
r s
A
Z
The last explored state with no child-states (Z) is modified. This can Introduce further child-states.
Nodes in the search space for:
ENDSPACE f r s
A
f(Z)

Figure 3.5: *An illustration of the* ENDSPACE *operation.*

The ENDSPACE functional is defined as follows:

$$\begin{array}{l}
\texttt{ENDSPACE\_COUNT} : \texttt{counter} \to (\texttt{rstate} \to \texttt{rstate}) \to \texttt{rtechn} \to \texttt{rtechn} \\
\texttt{ENDSPACE\_COUNT}\ c\ f\ r\ s = \\
\quad \{s_{next} \mid (r\ s = \{\} \ \wedge\ (c\_iszero\ c)\ \wedge\ s_{next} = f s) \vee \\
\qquad\qquad (\exists t.\ t \in r\ s\ \wedge\ (t.c = None\ \wedge\ s_{next} = c\_stopcounting\ c\ t) \vee \\
\qquad\qquad\quad (t.c = Some\ r'\ \wedge \\
\qquad\qquad\qquad s_{next} = c\_add(c, (t.p, Some\ (\lambda\, s.\ \texttt{ENDSPACE\_COUNT}\ (c\_dec\ c)\ f\ r'\ s), t.i)))))\}
\end{array}$$

where the functions on the counter have the following behaviour:

1. "*c_iszero c*" is true when the counter *c* is zero.
2. "*c_dec c*" decreases the count of unexplored reasoning states by one.
3. "*c_stopcounting c s*" notes that counting the number of unexplored state is no longer necessary as a branch that has the *None* continuation has been found, indicating successful completion of the technique. After this function has updated *c*, it returns *s*.
4. "*c_inc c s*" increments the number of unexplored states and returns *s*.

We note that in this definition we pass the information regarding the number of unexplored states as an argument to the auxiliary function. Furthermore, the information held by the counter must be stored in such a way that it is shared across alternative branches of the search space. In terms of implementation, this can be done by using a reference variable. If we wish to make this information accessible to other techniques then we can place the number of unexplored nodes in the contextual information.

Using the above auxiliary function, the ENDSPACE element can be defined as:

$$\begin{array}{l}
\texttt{ENDSPACE} : (\texttt{rstate} \to \texttt{rstate}) \to \texttt{rtechn} \to \texttt{rtechn} \\
\texttt{ENDSPACE}\ f\ r = \texttt{ENDSPACE\_COUNT}\ (init\_counter\ ())\ r
\end{array}$$

where the function *init_counter* constructs an initial counter for use in the ENDSPACE_COUNT function described earlier.

ORELSE

We use the ENDSPACE function to create an functional that combines two techniques only trying the second one if the first one fails. The idea is that when the search space of the first technique has been exhausted we modify the last reasoning state to start performing the second technique.

If there is a way for the first technique to complete successfully, then the second technique is never applied. This is expressed as follows:

$\texttt{ORELSE} : \texttt{rtechn} \rightarrow \texttt{rtechn} \rightarrow \texttt{rtechn}$

$r_1\ \texttt{ORELSE}\ r_2 = \texttt{ENDSPACE}\ (\lambda(p,c,i).(p,r_2,i))\ r_1$

An interesting feature of our version of `ORELSE` is that it behaves correctly independently of the search strategy used.

TRY

The try element is analogous to the methodical and tactical of the same name. It is given a technique r and if r fails (every branch in the application of r fails), then it behaves as if it were the `NOTHING` techniques: it simply has no continuation indicating successful completion. The `TRY` functional is defined using `ENDSPACE` as follows:

$\texttt{TRY} : \texttt{rtechn} \rightarrow \texttt{rtechn}$

$\texttt{TRY}\ r = \texttt{ENDSPACE}\ (\lambda(p,c,i).(p,None,i))\ r$

This is useful for writing techniques that behave like the ACL waterfall strategy [11], which can simply be expressed as a sequence of `THEN TRY`'s. We remark that like the `ORELSE` functional, it behaves correctly independently of the search strategy.

3.6 Logical Dependency

A problem arising in both proof planning and theorem proving is that proof techniques are often dependent on the logic or theory for which they are developed. However, we wish to make encoded techniques as independent as possible from the underlying logic. In effect, we want to provide techniques that are parameterised by the logic in which they are to be applied. Logical dependency, in our approach to proof planning, can arise in either the encoding of a technique, or in the contextual information.

Logical frameworks, such as Isabelle, provide a meta logic which can express proof rules. Generic tactics can then be expressed in terms of abstract proof rules. This provides such theorem provers with a generic foundation for writing tactics. Domain specific tactics can then be written in a way where the domain data is expressed explicitly. For example, a tradition in the Isabelle theorem prover is to write tactics as (ML) functors that are given the minimal logical requirements for the tactic. Such an approach treats the encoding of generic techniques as a problem of methodology. This is convenient as techniques can be defined with respect to theories as well as other existing tools. For example, a technique can be defined dependent on a generalisation mechanism, as well as on the existence of a specific proof tool such as Isabelle's simplifier. Contextual information can be written in the same way, making it a functor that defines its logical dependency. Techniques can then be defined in a way that is also dependent on the contextual information they use.

An alternative approach is to provide some fixed data type which can be used to specify the logical dependency of a technique. For example, by providing a datatype that holds constants and assumptions that a technique depends on. Such an approach is less flexible, but is more amenable to automatic installation of proof tools in a new theory. It is less flexible because new extra-logical tools may need to be specified in a dependency, but we cannot pre-suppose what may be needed as part of the data type. For example, if at a later point in such a new theory, a technique is defined in such a way that it is dependent on a characteristic that is not expressible in the logical dependency datatype, then it is not possible to write the technique in a generic way. For instance, in our example dependency-datatype, it would be impossible to express dependency of a technique on the simplifier.

We have adopted the more flexible approach, in keeping with the Isabelle tradition, and encoded techniques as functors. We can take advantage of ML in the development of our techniques by ensuring that they can be type checked independently of any logic or theory. Within a theory, we can provide the needed logical information to create a concrete instance of the technique. This approach pushes the checking of logical dependency into ML type checking.

Another salient feature of using a logical framework is that we get a uniform approach to the execution of proof plans. This means that only a single language is needed to express proof plans, namely the language for proof scripts used by the logical framework.

3.7 Proof Plans and Proof Scripts

Proof plans are interpreted descriptions of tactics that when executed produce a calculus level proof. Thus they define the relationship between high level reasoning techniques, such as 'proof by induction', and the low level proofs derived by the underlying theorem prover. The representation of proof plans reflects the theorem prover's mechanisms to derive proof. In LCF-style interactive systems, proofs are typically expressed in *proof scripts* which employ *tactics* that reduce a goal to subgoals. These proof scripts are typically procedural (a sequence of tactics), or structured such as those of Isar [102] and Mizar [91].

Proof planning involves searching the unfolding of a technique for a reasoning state that contains a proof plan which, when executed, results in a proof of the initial conjecture. The encoded techniques construct and modify proof plans. This process focuses on the incremental derivation of proof plans which distinguishes it from tactic based theorem proving that, in contrast, focuses on the derivation of calculus level proofs.

A distinguishing feature of our approach, in contrast to other proof planners, is that we interleave the execution of the proof plan with its construction. This facilitates the use of tactics, from the underlying theorem prover, as part of the proof planning process. Proof plans can thus provide a way for techniques to interact with the underlying theorem prover in much the same way as a proof script facilitates human interaction with the proof assistant. Furthermore, a proof plan can itself be expressed as a proof script. Thus, we can see that if proof planning automatically conjectures and proves lemmas, then it can be used to aid interactive theorem proving by automatically generating proof scripts that provide needed lemmas.

In systems that support large proof developments, some means of storing proved theorems is needed. It is common for systems to store the proof of a theorem as a procedural proof script, which is a sequence of instructions that, when run, will reproduce the proofs. If a theorem is proved by proof planning then it can be stored as either the commands to perform proof planning, or as the extracted proof script. Re-execution of proof planning commands will be slower than proof scripts, as the search for the proof plan will be performed again. However, storing the proof planning commands will be more robust. This is because proof planning may be able to search for a new solution taking into account changes in definitions and alterations to the behaviour of tactics.

Starting in 1973, the Mizar project [91] pioneered the declarative style of writing proof scripts. More recently languages such Isar [102] and SPL [103] have brought a structured style of writing proof scripts to other systems. In terms of storing the proofs, the methodology associated with declarative scripts discourages the use of many different tactics as this can obscure the proofs. Thus it seems natural to try and derive declarative proof scripts from proof plans as opposed to extending the tactic language with proof planning commands.

The internal representation of proof plans is important both for efficient proof planning and to enable generation of proof scripts. To express proof plans as a declarative proof script requires either the representation of the proof plan to mirror the declarative style of proof, or the use of a mechanism to translate procedural proofs to declarative ones. We further examine using proof planning to derive declarative Isar scripts in chapter 6. In terms of efficiency, to avoid re-execution of the proof plan, it is useful to store the current executed proofstate at each node in the proof plan. In this way the proof plan can also act as an agenda containing the remaining subgoals to be solved.

3.8 Conclusions

In this chapter we have introduced the notion of a technique language and described what we consider to be the important properties of such languages. We used these characteristics to analyse two common styles of encoding reasoning techniques, namely the functional style, as used to write tactics in Isabelle, and the interpreted style which is used to express methods in λ*Clam*. We have compared these and used the analysis to motivate our observational style which forms the foundation of our approach to proof planning. This provides an extensible language for encoding common patterns of reasoning. We have also introduced a basic language for writing techniques in the observational style which forms a platform on which we can define more complex techniques.

Special attention has been paid to the management of contextual information. In particular, we have described various ways this can be expressed, and its use in the development of proof critics. We have also addressed issues of managing logical dependency and outlined the importance and role of proof plans as an intermediary between the proof planner and the underlying theorem prover.

Chapter 4

Tracing Technique Applications

In the previous chapter we gave a rather abstract account of observational proof planning. In this chapter we describe how the application of techniques can be traced which provides a concrete illustration of our approach. We also motivate our tracing machinery and show how it can benefit user interaction.

4.1 Introduction

Many theorem proving tools rely on traces to inform the user of the tool's behaviour, including first order resolution provers, rewriting systems, inductive theorem provers and proof planners. Unfortunately such traces are often overly verbose and difficult for the user to understand. Furthermore, they do not offer the user a way to easily interact with the proof attempt.

Tracing proof attempts is an important aspect of interaction with theorem provers. Some systems, such as ACL2 [57], use traces as the main means of communication with the user. Additionally, another important task to which tracing of techniques contribute, is the development of proof techniques. For systems which support user-extensions, such as Isabelle's support for adding new tactics, this is thus another important interaction to support.

In this chapter, we describe machinery, based on our observational style, for tracing the behaviour of proof planning techniques. The general motivation is to help the user view the proof process by providing them with a clear notion of their location in the proof attempt. In particular, we consider the following issues:

Debugging: How can the internal steps of a proof tool be examined at a suitable level of detail for debugging?

Development: How can the interface help developers gain insight into ways that theorem proving techniques might be extended and improved?

Explanation: How can the effect of powerful proof techniques be made comprehensible to the user?

Specialised user interaction: How can user interactions be guided to particular points in the proof process such as those that the proof planner is most likely to get wrong?

We present machinery for tracing techniques in §4.2 and the interactive interface in §4.4. We examine how it can help the process of debugging techniques and developing them in §4.5 and §4.6 respectively. We consider using the tool for explaining a technique's behaviour in §4.7, and how it might provide a specialised user interaction in §4.8. Related work is presented in §4.9. Finally, we conclude in §4.10.

4.2 Tracing Proof Planning

A trace of a proof planing technique describes the steps that the technique has performed so far. This captures the path taken through the search space. We make this hierarchical so that some states in the path may be considered child-states of an earlier state. The interpretation is that child-states provide specific details about how the parent-state is performed. For example, a parent state named "rippling" would have child-states that specify the specific rules that are applied during rippling.

Each reasoning state contains a hierarchical trace which, because it is not logical information, is held as contextual information. Traces are represented using a simple tree structure with ordered child nodes. These nodes hold descriptions of the reasoning states that have been passed through. Thus we also provide contextual information to describe each reasoning state. For example, in Figure 4.1, we show a trace for the proof attempt of the step case in a proof of the commutativity of addition using rippling and fertilisation.

Descriptions and traces are constructed incrementally as techniques are applied. To simplify the process of writing techniques that build traces, we make use of the extensibility of our technique language to provide an alternative version of the `THEN` technique combinator:

`THEN A B` takes two techniques `A` and `B` as arguments. It applies the technique `A` fully, then creates a new node in the trace structure for `B` which will then start to be applied.

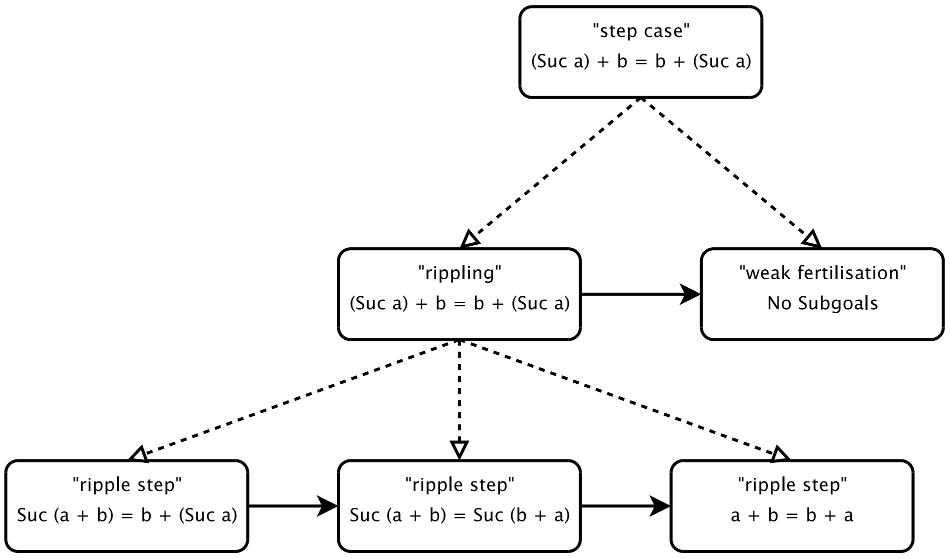


Figure 4.1: *An example hierarchical trace showing the descriptions and top goal of the reasoning states represented in the trace. The dotted arrows indicate child nodes and the solid arrows show the ordering of the children.*

This provides a way to automatically construct sibling nodes in the trace that is transparent to the technique developer. We also make use of the extensibility to introduce two new technique combinators which facilitate trace construction:

`GIVENAME N R` applies technique `R` once, then names the next state `N`. This is the basic primitive for providing explicit descriptions of reasoning states. It gives the next state a name to be used when it is added to the trace structure.

`REFINE N R` introduces an extra reasoning state in the proof planning attempt which is named `N` in the trace. The continuation of this introduced step is the technique `R` and the names of the child states unfolding from it are added as children nodes of `N` in the trace.

For example, these can be used to define the step case technique that produces the trace shown in Figure 4.1, as follows:

```
ripplestep = GIVENAME "ripple step" doripplestep
rippling = REFINE "rippling" (REPEAT ripplestep)
stepcase = REFINE "step case" (rippling THEN fertilisation)
```

where we omit the details of `doripplestep` as they are not relevant to trace construction. Note that, in Figure 4.1, the trace is constructed as a depth first traversal, which is the order in which the above techniques are applied. To ensure that the trace is constructed appropriately, other technique combinators, such as `REPEAT`, inherit the correct behaviour from the new version of `THEN`, or require slight modifications using the above primitives.

We remark that the hierarchical trace does relate proof techniques to open goals. Although we can show the top most goal, as we did in Figure 4.1 to clarify the technique's behaviour, in general, techniques can transform several goals simultaneously. This is needed for deductive synthesis problems, where an unknown meta-variable can occur in several goals and can be instantiated in all them during a single step in the application of proof planning techniques. The separation of the trace structure from the logical structure of the proof also enables the expression of proof critics. These can manipulate the proof plan rather than just affect an open goal. However, the lack of a well-defined relationship between the trace structure and the logical one can also make the trace a confusing representation of the proof. In future work, we hope to provide a more formal characterisation of this relationship.

4.3 Supporting Efficient Search with Traces

A naive implementation of the traces and descriptions for reasoning states is likely to introduce two significant inefficiencies for proof planning:

- Constructing a string to describe a reasoning state can involve significant computation, but is not needed during proof search itself. Such descriptions are only of interest when presenting a reasoning state to the user. Thus it is more desirable not to compute the descriptions of reasoning states during proof search.
- We need an efficient mechanism to add nodes to the trace. Using a traditional representation of trees will use time proportional to the size of the tree. As the trees grow large this will slow down proof planning increasingly. It is desirable to have a constant time machinery for adding nodes to the trace.

To avoid unnecessary computation when constructing a state's description, but maintain the ability to mix automatic search with user level guidance, we evaluate the description functions as they are needed for pretty printing. Thus state's description is actually a function that results in the string describing it. We achieve constant time addition of nodes to the trace by representing the trace using Huet's notion of zippers [47] which represent trees from the perspective of the leaf node.

4.4 The Tracing Interface

The main task for the interactive tracing tool is to allow the user to navigate through the search space within the application of a technique. This forms a custom search strategy for proof planning where the user decides which state to examine next.

The hierarchical nature of the trace structure allows the interactive tracing tool to step through 'chunks' of the proof attempt without the user having to examine the intermediate steps. In particular, it can perform a normal search algorithm, such as depth first search, until a state is found which has returned to the original trace-depth. For instance, this enables the tracer to step over the individual rewrites involved in rippling, allowing the user to navigate the search space at a higher level. For instance, rather than examine each step in the proof of a lemma, the user can step over the proof as if it were a single step.

In order to give the user a more logic-orientated view of the proof, IsaPlanner's tracer also shows the current partial proof plan as an Isar proof script, the current open goals, as well as the trace structure. This also helps clarify the relationship between the trace and the way it modifies the proof plan. For example, Figure 4.2 shows the output of the tracer during a proof of the commutativity of addition. The partial proof plan has proved the base case by simplification using the lemma $b+0=b$. To make the tree structure more readable, previous subtrees in the trace are not shown, unless the user can requests to see the full hierarchical trace.

The textual description of the trace shows child-nodes using indentation. A node starting with "+" indicates that it has child states, where those starting with "-" are leaf nodes in the hierarchical trace. The numbers in between the square parenthesis show which state was chosen from the or-choices and the total number of choices that were available. For example, `[1/2]` indicates that the first of two possible states was chosen. The trace structure does not show the search space related to these alternative branches.

The second part of the output from the tracer shows the options available to the user. The `eXit` command leaves the interactive tracer and returns the last reasoning state examined to the underlying ML interpreter. This allows the user to interact directly with the reasoning state, for example by changing the next technique to apply.

The numbered choices in the user options show the different or-choices in the search space.

```
Partial proof plan: -- {* Proving "a + b = b + a" *}
  proof (induct "a")
    have "b = b + 0" by (rule sym[OF add_0_right])
    thus "0 + b = b + 0" by (simp (no_asm))
  next
    fix N :: "Nat" assume H1: "∀ b. N + b = b + N"

Hi-Trace:
+[1/1] Induction and rippling...
  -[1/2] Induction on "a"
  +[1/1] Solve the base case using simplification...
  +[1/1] Solve the step case using rippling....
    -[1/1] Start Rippling with state:
           measure: (out)[2, 0, 0] (in)[0, 0, 0]
           annotated goal: [< suc >] N + |_b_| = |_b_| + [< suc >] N

Open Goals:
  1. suc N + b = b + suc N
-------------------------------------------------------------------
Commands: [x] eXit  [b] Back  [s] Step  [g] Go
View:     [c] Conjectures  [i] Isar Script  [v] Full HiTrace

[1]  Ripple Step by (subst add_suc)
     measure: (out)[1, 1, 0] (in)[0, 0, 0]
     annotated goal: [< suc >] (N + |_b_|) = |_b_| + [< suc >] N

[2]  Ripple Step by (subst add_suc_right)
     measure: (out)[1, 1, 0] (in)[0, 0, 0]
     annotated goal: [< suc >] N + |_b_| = [< suc >] (|_b_| + N)
```

Figure 4.2: *An example of the user interface during an interactive trace of the proof of the commutativity of addition. The top half presents information about the reasoning state and the bottom half shows the user's options.*

This allows the user to manually select which branch to explore next. There are three other navigational commands:

- `Back` returns to the previously examined node,
- `Step` is given a number and performs a depth first search from the selected or-choice in the numbered list. It searches for the next state that returns to the same depth in the hierarchical trace.
- `Go` performs a depth first search for a reasoning state that solves the initial given conjecture.

There are also three viewing commands. These allow the user to view: the conjectures made made so far with their state (proved, failed to be proved, or refuted), the full hierarchical trace, and the full Isar proof script including the automatically proved lemmas.

One of the main differences between this kind of command driven interaction in comparison with the traditional tactic driven approach, employed by interfaces such as Proof General [1], is that our interface shows the different ways in which a technique can be applied. In contrast to this, other command driven interfaces show only the first result. Such interfaces require the user to either modify the parameters to the tactic in order to get a different behaviour or explicitly enter backtracking commands. Our approach, which allows the user to see and select desired result, is particularly useful when parameterising the tactic requires intricate knowledge of its behaviour.

This utility of this can be seen in the following simple example. Consider the proof of $x+0=x$ in Peano arithmetic where addition is defined recursively on the first argument. In the inductive proof of this theorem, we arrive at the step case goal $Suc\ (x+0) = Suc\ x$. To complete the proof, we can apply the induction hypothesis, $x+0=x$ to remove the 0 on the left or introduce a 0 on the right. However substitution with the hypothesis can be done in many different ways, each of which results in an alternative subgoal. However, only two of these help complete the proof. The full list of choices are:

1. $Suc\ ((x+0)+0) = Suc\ x$
2. $Suc\ (x+(0+0)) = Suc\ x$
3. $Suc\ ((x+0)+0) = Suc\ x$
4. $(Suc\ (x+0))+0 = Suc\ x$

5. $Suc\ (x+0) = Suc\ (x+0)$ [*]

6. $Suc\ (x+0) = (Suc\ x)+0$

7. $Suc\ x = Suc\ x$ [*]

but only those marked with a '*' lead to a direct proof by reflexivity. The approach often employed in tactic driven interfaces is to allow the user to select a redex by either partially instantiating the rule or by giving a number referring to a redex in some ordering of the search for substitutions. However, such an approach requires the user consider how to apply the tatic in order to then guide the prover. In the above example, the user must know the ordering in which the subtiution tactic considers redexes, or provide the instantiation by hand. In contrast, our approach provides the user with the possible results of the technique's application and allows them to select which result they would like to arrive at. This makes the interface do more of the work and thus removes the need for the user to know the details of the proof tool.

We remark that this does not always work ideally. In some cases a tool can be applied in a large number of ways which can result in the user being swamped by choices. Such situations require the developer of the technique to provide further pruning of the search space. However, because such pruning is not always possible, we do not consider our proposed approach as an alternative to tactic based theorem proving but as an extension of it. It is still useful to allow the user to specify a redex or give an instantiation by hand.

4.5 Debugging

One of the chief difficulties in developing proof tools for Isabelle is debugging them. The black-box evaluation of ML functions makes it difficult to observe which part of the proof tool fails and why.

A common approach to debugging is to insert print statements into the technique and use these to observe which part of the code is failing. However, for complex proof tools which perform many steps, this leads to an excessive amount of output which the developer then has to examine. Using flags to turn on and off different kinds of printing can help, but tends to clutter the code with conditional pretty printing statement.

Another traditional approach is to use language-level debugging tools, such as an ML debugger. Such tools provide similar support to our interactive tracer. However, they tend to provide an overly detailed view of the proof process, showing many internal details. Our tracing mechanism provides an interface for debugging in higher level terms, and supports stepping over parts of the proof in a different way to the underlying function calls.

An example where we found the tool to be particularly useful was in debugging automatic lemma speculation. Errors we made in the development of this critic resulted in the wrong lemma being speculated. Running the technique resulted in non-termination. However, there are many possible errors that could have this result. Our tracing mechanism allowed us to identify the lemma speculation problem easily.

4.6 Development

The ability to observe the failure of a technique often gives a good indication of how the technique might be modified to solve the problem. The patching of failed proof attempts is the central idea behind Ireland's development of proof critics [52]. We found the tool helpful in the development of generalisation critics for higher-order logics.

For example, during the proof attempt of $a^{(b+c)} = a^b * a^c$ within Peano arithmetic, our initial technique, having performed induction on b, arrived at the step case subgoal $a * (a^n * a^c) = (a * a^n) * a^c$. Our initial lemma speculation critic conjectured, $a * (g * a^c) = a * g * a^c$, generalising only over a single common subterm, rather than suggesting the more general associativity of multiplication $x * (y * z) = (x * y) * z$. Stepping through the proof attempt to the location where the generalisation was made quickly made the problem and its solution apparent. We were then able to quickly modify the speculation mechanism to further generalise the conjectured lemma.

Another example of the benefit to the development of proof techniques is shown in the proof attempt of $x + 0 = x$ which we introduced earlier. Techniques which have a high branching factor can be are apparent during tracing. In the above example, considering the application of the induction hypothesis brought to light the following heuristic for pruning the search space: during an inductive proof, only substitutions with the induction hypothesis used from left to right should be considered in the left hand side of the subgoal. Symetrically, only right to left subsitutions should be considered in the right hand side. We then observed that this heuristic also has a theoretical motivation. In particular, only these subtiutions make the left and right hand sides of an equational subgoal more similar. Thus only these applications are likely to allow the proof to then be completed by reflexivity. Our weak fertilisation technique is described further in chapter 9.

Although the tracing tool allows the user to explore the search space and observe specific points when the proof fails, it gives only a limited indication of which parts of the search space are irrelevant to the proof. In particular, it allows us to observe points with a high branching factor, but does not show which branches do not lead to success. We suggest that the tracing tool should be combined with visualisation mechanisms in order to help developers devise further ways to prune the search space of their techniques.

4.7 Explanation

The converse of understanding why a technique fails is comprehending why it succeeded to prove a goal. In a sense, we are asking for an explanation of the proof. The motivation for such proof explanation comes from the increasing use of powerful proof techniques that can perform steps that are too large to be comprehensible to the user. For example, IsaPlanner can prove, from only the primitive definitions, the theorem $(a^b)^c = a^{b*c}$ in Peano arithmetic, whereas most presentations of this involve several lemmas. We note that new users to the theorem prover may also find proof explanations useful to understand how the underlying proof system behaves.

The tracing structure provides an approach to explaining the application of a technique. It tells the story of the proof attempt. For example, the trace produced for by proof planning $(a^b)^c = a^{b*c}$ is shown in Figure 4.3. At present, these descirptions are text based, but we believe that a visual presentation, such as that used in the XBarnicle system [54], which uses nested boxes to show the structure of the proof, could also help understanding of the proof attempt. A tool for constructing and interacting with such visualisations of traces is further work. However, we note that the relationship between the trace and the underlying proof would need to be formalised in order to allow the user to independently guide the proof attempts of different subgoals.

Unlike approaches that explain proofs at a logical level, examining the hierarchical-trace explains how the proof was found in terms of its path through the search space. It particular, it does not necessarily reflect the proof's structure. Although this can be confusing, it provides an ability to present the behaviour of proof critics. This makes it a distinct object from the proof plan, which presents only the final proof. For example, contrast the trace and the proof plan produced when proof planning the theorem $(a^b)^c = a^{b*c}$, shown in Figures 4.3 and 4.4 respectively. We note that using the Isar language as the representation of poof plans makes the logical structure of the proof readable. Furthermore, it allows the proof scripts constructed by proof planning to be copy and pasted into the theory developments. Providing improved mechanisms for managing this kind of interaction is another interesting area for further work.

```
+[1/1] Prove the goal using Induction and Rippling...
  -[3/3] induction on "c"
  -[1/1] Solve the base case by simplification.
  +[1/1] Solve the step case using rippling...
    -[1/1] Start Rippling with state:
          measure: out:[1, 1, 0, 0], in:[0, 0, 0, 0]
          aterm: "(a^b)^[<Suc>] c = a^(b*[<Suc>]c)"
    -[2/2] Ripple Step by subst mult_Suc
          measure: out:[0, 2, 0, 0], in:[0, 0, 0, 0]
          aterm: "(a^b)^[<Suc>] c = a^([<op +>] (b*c) [<b>])"
    -[1/1] Ripple Step by subst exp_Suc
          measure: out:[0, 1, 1, 0], in:[0, 0, 0, 0]
          aterm: "[<op *>] ((a^b)^c) [<a^b>] = a^([<op +>] (b*c) [<b>])"
  -[1/1] Weak fertilisation
  +[1/1] By proving lemma1: "a^g*a^b = a^(g+b)"...
```

Figure 4.3: An example trace produced by proof planning the theorem $(a^b)^c = a^{b*c}$ where exponentiation is written using the infix operator ^. We omit the trace for lemma which involves conjecturing and proving further lemmas for the sake of brevity.

```
theorem $(a^b)^c \ = \ a^{(b \ * \ c)}$
proof (induct $c$)
  show $(a^b)^0 \ = \ a^{(b \ * \ 0)}$ by simp
next
  fix $c$
  assume IH: $\forall \ a \ b. \ (a^b)^c \ = \ a^{(b \ * \ c)}$
  have $a^{(b \ * \ c)} \ * \ a^b \ = \ a^{(b \ * \ c \ + \ b)}$ by (rule lemma1)
  hence $(a^b)^c \ * \ a^b \ = \ a^{(b \ * \ c \ + \ b)}$ by (subst IH)
  hence $(a^b)^{(Suc \ c)} \ = \ a^{(b \ * \ c \ + \ b)}$ by (subst exp_Suc)
  thus $(a^b)^c \ = \ a^{(b \ * \ c)}$ by (subst mult_Suc)
qed
```

Figure 4.4: The Isar proof script produced by proof planning the theorem $(a^b)^c = a^{b*c}$. For brevity, the proof script for the lemma named *lemma1* is omitted.

4.8 Specialised User Interaction

Interactive navigation of proof attempts, rather than using fixed search strategies, allows the user to avoid unpromising paths in the search space. This provides a novel approach to working with the theorem prover. IsaPlanner further specialises this user interaction by allowing techniques to tag a reasoning state. The tracer can then be told to search normally, until a state containing the requested tag is found. The intuition is that tags will be used to indicate a particular point in the proof, for example a generalisation step. This would then allow search to continue until a generalisation is performed. Like the state description, the tag is held in the contextual information. It is an interpreted piece of information designed to held the automation of interaction with the prover. The difference between tags and the descriptions of states is that tags are static, for use by automatic tools, and quick to examine. In contrast, the descriptions are verbose, consturcted to describe the state to the user, and can be slow to compute.

In terms of writing techniques in IsaPlanner, we extend the basic language with a tagging constructor, "`TAG N R`", which unfolds the technique `R` once, and tags the resulting state with the tag `N`. This allows the tracer to search for a state in which some action is performed by the applied technique, indicated by the tag. This can be used to focus the user interaction on points in the proof where a technique is more likely to make an error, such as lemma speculation.

4.9 Related Work

Many proof tools provide a trace of their behaviour. For example, the λ*Clam* system has a step-by-step mode which allows the user to step through the system's proof planning attempt [32]. Like many other systems, λ*Clam* also provides different levels of printed output, from verbose messages to none at all. Similarly, Isabelle's simplifier provides an option to trace its application of rewrite rules. As mentioned earlier, such traces of a proof attempt are of even greater importance in the ACL2 system [57], as they are the main mechanism for the prover to communicate with the user. However, the traces provided by these systems are one-dimensional: the user can view the trace but cannot interact with the proof tool. In contrast, the approach to tracing that we have presented in this paper provides an interactive mechanism for exploring the search space and allows the user to modify with the proof attempt.

A few other systems also allow techniques to be applied in a hierarchical fashion. In particular, the tactics of Nuprl [30] and the methods of the Multi proof planner in the Omega system [73]. The main difference is that our notion of unfolding a technique can involve choice, where other systems have a deterministic result. We allow the user to interact with the choices in the unfolding of the hierarchy and use this as a mechanism for interaction. Another significant difference is that our hierarchy is not directly related to the structure of the underlying proof. Some steps can completely change the proof plan where steps in the unfolding of hierarchical techniques in Nuprl and Omega only refine part of the proof related to a particular subgoal. In this sense, our hierarchy is a generalisation of that used in these systems.

4.10 Conclusions

We have described the part of IsaPlanner's language for encoding techniques that constructs hierarchical traces. This is used by the system's interactive tracing machinery to support flexible navigation through the search space involved in the application of a technique. We provide the user with a clear notion of their location in the search space of a proof attempt. We believe this can help guide the interaction with the prover. We also found this tracing mechanism particularly beneficial to the development and debugging of proof techniques. The only additional burden on the technique developer is to use the provided constructs. We found this to have significantly less damage to the clarity of the code than the use of conditional printing statements.

The tracer can be used to explain a proof in varying levels of detail, without hiding the application of proof critics. Our tracing mechanism is text based and gives an operational, rather than visual, view of the search space. Providing a visual representation traces and machinery for interaction is further work. However, a clear relationship between the tracer and the open goals would be required. In future work we intend to examine ways to include some goal information in the trace. Further work also includes combining the tracing tool with a visual presentation of the search space. We believe this could help developers focus their attention onto portions of the search space that might usefully be pruned.

In order to further evaluate the interactive tracing of techniques we suggest performing an experiment with new users to Isabelle. The hypothesis would be that examining the trace of a technique will help users to learn how to prove theorems in an interactive prover. Users would be split into two groups. One group would be given the choice to use the interactive tracing machinery and the other would use techniques without being able to further examine their unfolding. The groups would both be given a large set of proofs to complete and they would be meaured based on how many proofs they completed. If the tracing tool helps users then we would expect that group to complete more proofs. In such an experiment the same level of automation should be provided to both groups. Such an experiment would help to identify if traces are useful for more than just the development of proof techniques.

Chapter 5

Search

In this chapter, we describe the relationship between search and the application of techniques encoded in our observational style of proof planning. In particular, we present a functional interpretation of search and consider how different strategies can be combined. We clarify our approach by providing an algorithm that supports the mixing of search strategies. We then describe how this abstract view is instantiated in IsaPlanner and introduce derived elements that allow techniques to specify a search strategy to be used locally.

5.1 Introduction

The separation of the search space from the algorithm used to explore it is fundamental to the notion of proof planning. The technique developer encodes a pattern of reasoning which defines the search space of possible proofs. The proof planner interprets and applies these encoded techniques to search this space for successful proof plans that solve the given problem.

Despite this, most proof planners use only depth first search. However, in recent systems, the mechanisms for encoding and applying techniques has become increasingly complex. For example, Multi, the latest in the Omega family of proof planners, includes methods as well as six additional structures for encoding patterns of reasoning[1], all of which are interpreted by the proof planner in a different way [73]. However, choice points within the application of these structures are still treated in a depth first manner. Similarly, the Clam family of systems has introduced a notion of proof critics in order to patch failed proof attempts [51,52], but still uses the underlying depth first search of (λ)Prolog.

[1]These are strategic control rules, backtracking strategies, planning strategies, control rules, methods, tactics, external systems, and general purpose lisp code written by the technique developer within any of these structures.

Both Omega and λ*Clam* extended their predecessor systems in order to support new kinds of techniques. This added complexity blurs the distinction between search space and strategy. It becomes less clear how much strategy has been encoded in the planner and how much in the proof methods. The close integration of the search strategy with the planner's interpretation of encoded methods also makes it more difficult to change or experiment with the search strategy. This is exemplified in the development of the Clam family of systems. An experiment with a modified search strategy for the Clam system is described by Manning et. al. [68]. While they report on the success of their approach, later version of Clam which used a structured method hierarchy and included proof critics lost the ability to perform best first search because of the increased complexity of the proof planning machinery.

Thus, we believe that to restrict theorem proving to a single search strategy is too strong a limitation on a research tool designed for the experimentation and exploration of common patterns of reasoning. In support of this view, we note that search is an important aspect of many techniques, from resolution theorem proving to rippling, and the effect of the search strategy is still an open issue for many techniques, including rippling. For instance, best first search has recently been suggested by Bundy et. al. [17] as a promising way to tackle problems that are normally outwith the reach of rippling. However, in all the existing proof planning systems that we are aware of, it is not possible to mix search strategies, or for the encoded "patterns of reasoning" to specify them. Thus such experiments have so far been difficult to carry out.

The main contribution of this chapter is the development of an approach to search, for our observational style of proof planning, that provides a flexible environment for experimentation. In particular, we describe how a technique can specify to a search strategy to be used locally. Our approach supports the combination of techniques which employ different search strategies and provides a clear mechanism for the relationship between different search strategies within a single proof planning attempt. We also describe how our approach supports combining search strategies and examine how further interaction between techniques and search can be facilitated. This provides a richer framework for experimenting with different search strategies than is available in other proof planning systems.

Overview

In this chapter, we firstly describe the role of search for our observational style of proof planning (§5.2). We then describe how search strategies can be combined (§5.3) and introduce a typed functional view of search (§5.4). We then use this representation to provide an algorithm for combining search strategies (§5.5). The observational technique language is thus extended to provide constructs that allow techniques to specify a search strategy to be used locally (§5.6). We then consider other ways in which techniques can interaction with the search process (§5.7) and finally present our conclusions.

5.2 Search in Observational Proof Planning

In chapter 3, we introduced the observational style of proof planning and motivated it by arguing that it provides a more flexible environment for the encoding of techniques. In particular, encoded techniques are functions within the state that produce possible new states.

Thus, unlike interpreted style proof planning which implicitly creates the search space as a result of the choices in the interpretation of methods, in the observational style, the search space is a direct result of the application of a technique. Choices that can be searched over in the observational-style are expressed as reasoning states in the lazy list produced by a technique's application. Recursively unfolding the continuation technique in these states results in a lazily evaluated search tree, as illustrated in Figure 5.1. The lack of a continuation indicates a leaf state that is considered a solution-node in the search space. On the hand, a state with a continuation technique that produces an empty list represents a non-solution leaf-node which will be backtracked over.

This lazy unfolding of techniques allows them to be applied in a possibly infinite number of ways, for instance corresponding to unifiers using Huet's higher-order unification [46]. While this is similar to Isabelle's approach of expressing the possible ways a tactic can be applied as a lazy list, it differs in that it provides a explicit representation of the search tree. In contrast, Isabelle's search space is evaluated eagerly, which results in a flattened view of the search space.

The explicit nature of the search space makes it easy to employ and experiment with different search strategies. For example, a generic version of depth first search can be defined as follows:

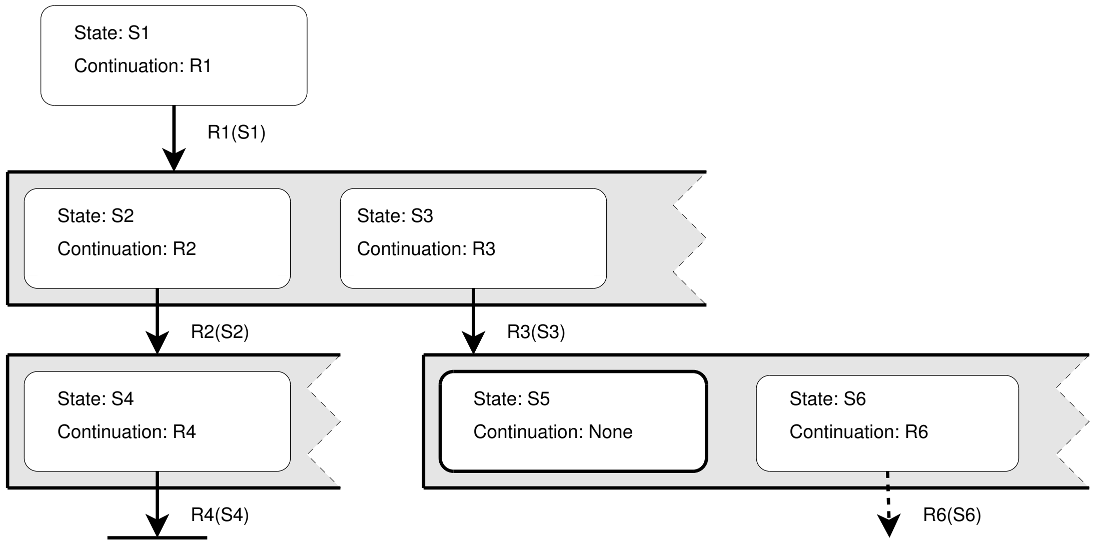


Figure 5.1: *An illustration of the search space that results from the lazy unfolding of a reasoning technique. Unfolding state* `S4` *results in the empty sequence indicating that it is a non-solution leaf node to be backtracked over. In contrast to this, state* `S5` *has no continuation, indicating that it should be considered as a solution to the search strategy.*

```
fun depth_fs goalf childf state =
    let fun dfs_aux [] = []
          | dfs_aux ( h :: t) =
            if (goalf h) then h :: (dfs_aux t)
            else dfs_aux ((childf h) @ tq)
    in dfs_aux [state] end;
```

This takes as arguments a function `goalf` to identify when a solution is found, a function `childf` to produce the child nodes from a parent node, and an initial node to start searching from. Using this to search an observational proof planning attempt simply involves letting `goalf` be true iff there is no continuation of the reasoning state, and `childf` be the unfolding function that applies a states continuation to itself. The initial state to search from is passed as `state`. We present search algorithms, such as the above depth first search, in a non-lazy manner for the sake of clarity, although it is a trivial exercise to make them lazy.

Because choice points in a technique are lifted to the resulting lazy list, simple generic search mechanisms, such as the depth first algorithm presented above, are directly applicable. The search strategy only has to select the next continuation state to unfold. Proof planning is then the process of searching over a technique's application for a proof plan that solves the goal.

5.3 Combining Search Strategies

One of the most obvious limitations of existing proof planning systems is that techniques cannot be defined in terms of search strategies. It seems desirable to use different kinds of search for different parts of the search space. For instance, when performing an inductive proof using rippling for the step case, rippling can benefit from best first search, whereas a simplification based technique, used to solve the base case, may not have a clear measure and thus it may be more appropriate to use depth first search. In particular, it is useful for a technique to be able to change the current search strategy. Failing to support this limits the expressivity of the technique language.

We observe that the heuristic in best first search can be manipulated to express any combination of strategies by changing the measure. However, this can easily become an 'ad-hoc' solution that makes it difficult to express natural combinations of search strategies and techniques. We believe that best first search should be used when there is a heuristic that can be expressed conveniently. If heuristics become over complex, they tend to become fragile. Furthermore, it can become unclear exactly what work the heuristic is doing. Complex heuristics can also slow down search process itself.

We describe two approaches to combining search strategies that maintain a clear notion of search and avoid such issues. Both mechanisms support a mixing of search in the observational style. The first approach, *nested search*, is easily implemented but reduces the flexibility of the technique language. The second approach, *stacked search*, is more complicated but preserves the expected behaviour of techniques and avoids the limitations of nested search.

5.3.1 Nested Search

Mixing search strategies can be supported by nesting a complete proof planning attempt within the application of a technique as illustrated in Figure 5.2. This is the obvious and naive approach to combing search strategies and is easily implemented.

In the functional style, nested search is the only way that search strategies can be mixed. This is because techniques are eagerly evaluated and thus the search has to be embedded within the evaluation of a technique.

For the observational style, we remark that nested search removes the ability to introspect

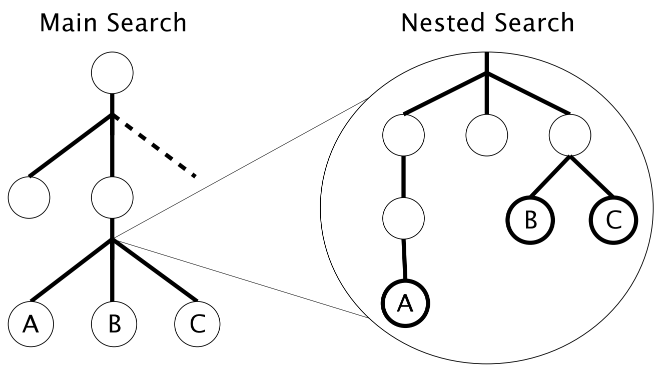


Figure 5.2: *An illustration of nested search, where the application of a reasoning technique can simply be a separate proof attempt which in turn can further involve search.*

into a technique's application. Reasoning states no longer express incremental snapshots of the proof planning process as a single reasoning step can now involve search over the application of an arbitrarily large technique. This breaks the ability to step-through the proof planning process. Instead the user is forced to step over the whole search.

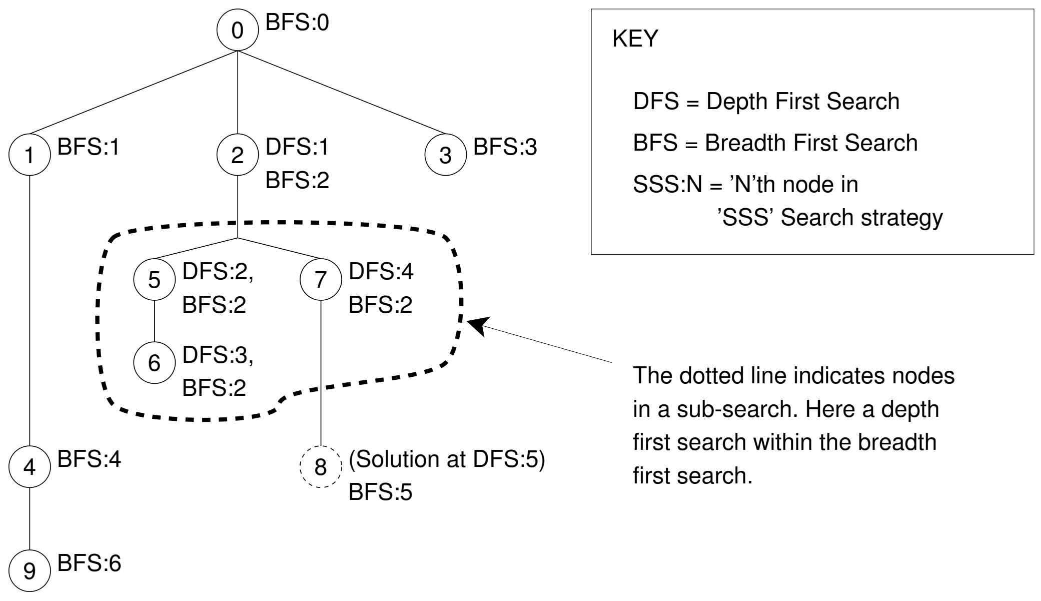
The problem is that the search process is performed eagerly, but the observational style's step-by-step notion of application is lazy. Thus a technique's application that uses nested search will hide the intermediate search space within its application. A nested search removes the extra expressivity in the technique language that allows one technique to examine another's unfolding, as is used by the `MAP` and `FOLD` combinators described in Chapter 3.

5.3.2 Stacked Search

Stacked search provides an approach more suited to the observational style than nested search by maintaining the lazy nature of applying a technique. It involves using a 'stack' of search strategies with their related agendas and allowing reasoning states to interact with this stack. The element on the top of the stack is the current search strategy and its current agenda. For example, Figure 5.3 shows a search tree, using stacked search, where depth first search is used within the context of a breadth first search.

When a sub-search finds *goal node*, one that meets the requirements of what was being searched for, it is returned as the next node to be added to the agenda of the parent search strategy. In the above example, goal nodes found by depth first search are returned as the next nodes to be added to the breadth first search agenda.

Stacked search behaves in the same way as nested search but has the advantage that tech-



BFS:0
BFS:1
DFS:1
BFS:2
BFS:3
DFS:2,
BFS:2
DFS:4
BFS:2
DFS:3,
BFS:2
BFS:4
(Solution at DFS:5)
BFS:5
BFS:6
KEY
DFS = Depth First Search
BFS = Breadth First Search
SSS:N = 'N'th node in 'SSS' Search strategy
The dotted line indicates nodes in a sub-search. Here a depth first search within the breadth first search.

Figure 5.3: *An illustration of stacked search, where the search strategy is held in a stack as part of the state. Note that node 8 is a success node for the depth first search, thus resulting in a return to the previous search strategy.*

niques are still evaluated in a 'lazy' fashion which supports introspection over the unfolding of a technique and interactive step-by-step debugging of its application.

Stacked search is implemented as a top level search strategy that interacts with the nodes in the space, allowing them to push search strategies onto the stack, and declare themselves to be solutions. This provides a flexible mechanism for experimenting with search strategies and their combination.

We develop this approach by first defining a general view of search strategies that expresses them as an instance of a datatype within a functional setting. We then present the machinery to combine such search strategies, forming the stacked search strategy.

5.4 A Uniform Functional View of Search

The first step in our development of a flexible tool for experimenting with search is to abstractly define the notion of a search strategy. We do this in an ML-like functional style, showing how different search strategies can be treated uniformly. This allows a simple and direct implementation and also facilitates mechanised reasoning about the search process.

Our account is minimal in the sense that it avoids introducing details specific to only a single strategy, such as notions of operator, parent nodes, depth, cost, and path. We believe that such search-specific details should be introduced as needed to specific strategies that uses them. This is in contrast to presentations such as that given by Russell and Norvig [92]. We define our functional notion of *search strategy* only in terms of nodes, for which we will use the type `'n`, and an agenda, which will have type `'a`.

Type Definition 5.4.1 (Search strategy):

```
('n, 'a) strategy =
 { agenda : 'a,
   addnodes : 'n list → 'a → 'a,
   popnode : 'a → ('n × 'a) option }
```

This type definition for strategies captures the state of a strategy using the agenda, and characterises the ways that it can be transformed using the `addnodes` and `popnode` functions. The `addnodes` function allows new nodes in the search space to be added to the agenda, and the `popnodes` function supports getting the next state to be explored. When `popnodes` is applied to an empty agenda then `None` is returned, otherwise it returns the next node to be examined and the agenda with that node removed. For its part, the `agenda` field initially holds the empty agenda for the search strategy, but later holds the current agenda for a search strategy.

As is common in the literature, we abstract over the type of the node we are searching for, and over how to get new nodes from an existing node. As such, it presents the behaviour of a search strategy only in terms of the way it treats its agenda. This makes an implicit assumption about the way such search strategies are interacted with, namely that any new nodes added to the agenda are the children of the last node popped from the agenda.

Using this characterisation of search we can now define specific kinds of search as instances of the general type `('n, 'a) strategy`. We now present the basic search strategies in this way. We will use the letter `a` for a variable representing the agenda and `nds` for one that holds a list of nodes.

Search Strategy Definition 5.4.2 (Depth First Search):

```
depth_first_search =
  { agenda = [],
    addnodes nds a = nds @ a,
    popnode [] = None
          | (h::t) = Some(h,t) }
```

This instantiates the agenda to being of type `'n list` and thus gives depth first search strategy type `('n, 'n list) strategy`. Depth first search simply involves adding new nodes to the start of the agenda.

Search Strategy Definition 5.4.3 (Breadth First Search):

```
breadth_first_search =
  { agenda = [],
    addnodes nds a = a @ nds,
    popnode [] = None
          | (h::t) = Some(h,t) }
```

Breadth first search is defined similarly to depth first search, but has a different `addnodes` function to place new nodes at the end of the agenda, rather than the start.

Search Strategy Definition 5.4.4 (Best First Search):
We implement best first search by letting the agenda be a list of nodes sorted by the heuristic measure. This heuristic is itself an argument to the search strategy:

```
best_first_search heuristic =
  { agenda = [],
    addnodes nds a = sortandmerge(heuristic, nds, a),
    popnode [] = None
          | (h::t) = Some(h,t)
```

This employs a function `sortandmerge` to sort the new nodes into the agenda using the heuristic.

While the above search strategies treat the agenda as a list, search strategies can have agendas of a different type. For instance, iterative deepening also holds the current depth to search until.

Search Strategy Definition 5.4.5 (Iterative Deepening Search):
We present a version of iterative deepening search that takes parameters indicating the amount to increase the search depth by and a list of nodes from which to create the initial agenda.

This strategy uses a richer notion of agenda to keep track of the current maximal depth (`maxd`), the depth of the last node popped from the agenda (`lastd`), the current agenda (`nodes`) organised in the same fashion as in depth first search, and the initial agenda to restart searching from (`initnds`). Additionally, it internally pairs each node with its depth. In order to keep track of the search depth, we use the implicit knowledge that added nodes come from the last popped node.

```
fun iterative_deepening_search depth nds =
  Strategy
  { agenda = let startnds = map (λ n. (0,n)) nds
             in { maxd = depth, lastd = 0,
                  initnds = startnds, nodes = startnds } end,
    addnodes =
      (λ nds { maxd, lastd, initnds, nodes } =
      { maxd = d, lastd = lastd, initnds = initnds,
        nodes = (map (λ n. (lastd + 1,n)) nds) @ nodes })
    popnode =
      (λ { maxd, lastd, initnds, nodes = [], }.  None
       | { maxd, lastd, initnds, nodes = ((d,n)::t) }.
           if maxd < d then
             popnode { maxd = maxd + d, lastd = lastd,
                       initnds = initnds, nodes = initnds }
           else
             Some(n,{ maxd = maxd, lastd = d,
                      initnds = initnds, nodes = t })) }
```

We remark that requiring multiple solutions from a search can result in a possibly unintended behaviour for search strategies that examine states more than once, such as iterative deepening. The problem is that goal states examined more than once will be returned each time they are examined. For iterative deepening search, this will give back all solutions at depth *n*, for each search of depth greater than *n*. Accounts of search in the literature, such as that presented by Russell and Norvig [92], avoid considering multiple solutions. Recording the position in the search space can be used to avoid repeated solutions, although it does further complicate the presentation of the search algorithm.

Abstracting Over the Agenda

One of the motivations for this characterisation of search is that it allows us to 'wrap up' the search strategies into objects of uniform type, abstracting over the agenda. This in turn allows a functional treatment of different search strategies in a uniform manner, thus providing us with an ability to combine them. We abstract over the different kinds of agenda by defining a type, `search`, in terms of only the functions `addnodes` and `popnode`:

Type Definition 5.4.6 (Search):

```
'n search = {add : 'n list → 'n search
             pop : unit → ('n × 'n search) option }
```

This captures a 'snapshot' of the strategy's search process, where `add` is the continuation function that gives the next snapshot when nodes are added to the agenda. The `pop` function gives the continuation resulting from removing a node from from agenda. This abstracts over the agenda in the functions `addnodes` and `popnode` that define a search strategy.

We now define two helper functions that update a strategy using the `addnodes` and `popnode` functions:

```
addnodes_strategy : ('n, 'a) strategy → 'n list → ('n, 'a) strategy
popnodes_strategy : ('n, 'a) strategy → ('n × ('n, 'a) strategy) option

addnodes_strategy { agenda, addnodes, popnode } nds =
  {agenda = addnodes agenda nds, addnodes = addnodes, popnode = popnode }

popnodes_strategy { agenda, addnodes, popnode } =
  case popnode agenda of None → None
     | Some(node, newagenda) →
         Some(node, { agenda = newagenda, addnodes = addnodes,
                      popnode = popnode })
```

To allow the wrapping up of strategies into an object of type `'n search`, we use these in the definition of two mutually recursive functions that have the following type and definition:

```
addf : ('n, 'a) strategy → 'n list → 'n search
popf : ('n, 'a) strategy → unit → 'n search

addf strategy nds = let newstrategy = addnodes_strategy strategy nds in
            {add = addf newstrategy, pop = popf newstrategy } end

popf s () =
  case popnodes_strategy s of None ⇒ None
     | Some (node,newstrategy) ⇒
         Some (node, {add = addf newstrategy, pop = popf newstrategy }) end
```

These hide the type of the agenda used by the search strategy as arguments to function calls. This allows the following function to create an object of type `'n search` from one of type `('a, 'n) strategy`:

```
mksearch : ('a, 'n) strategy ⇒ 'n search

mksearch strategy = {addnodes = addf strategy, popnode = popf strategy }
```

These definitions linearise the process of search to a series of snapshots which forms a lazy list of nodes arranged in the order that the search strategy will examine them. This allows us to define a simple function to step through the search process looking for a node. In addition, we can ask the search process to continue even after we have found a node, for instance in order to try and find more solutions.

The following function steps through the linearised search process until a solution node is found, or the search space is exhausted. Solution nodes are returned with the state of the search strategy in order to allow more solution to be searched for.

Function Definition 5.4.7 (Searching):

```
dosearch : ('n ⇒ bool) ⇒ ('n ⇒ 'n list) ⇒ 'n search
           ⇒ ('n × 'n search) option

dosearch goalf childf search =
  case popf search of None ⇒ None
     | Some (node,moresearch) ⇒ if goalf node then Some (node,moresearch)
                      else dosearch goalf childf moresearch
```

The argument `goalf` is a function that is true when a solution node if found. The function `childf` returns the children nodes of the node it is applied to, and `search` is the object of type `'n search` that holds the search strategy being used.

5.5 Combining Search Strategies using Stacked Search

We now consider how we can extend this functional view of search in order to allow different search strategies to be combined. We provide a *stacked search strategy* that allows states to modify the local behaviour of search by adding new kinds of search onto the stack. We also want to allow states to pop nodes from the stack. In particular, nodes that are solutions to a child-search-strategy should be given back to the super-search-strategy.

To allow states to indicate that a new search strategy should be started, or that an old strategy has ended, we introduce a notion of *search operator* that allows a node to interact with the search process:

Type Definition 5.5.1 (Search Operator):

```
'n searchop = addstrategy of 'n search
            | endstrategy
```

The `endstrategy` operator indicates that the node is a solution to the current search strategy. To allow the search process to get these operators, we also define a function to pop a search operator from a node:

Function Definition 5.5.2 (Pop Search Operator):

```
popsearchop : 'n → ('n searchop × 'n) option
```

When a node does not affect the search stack, the `popsearchop` function returns `None`, otherwise it returns the next operation to perform on the search stack.

In this view of search, nodes indicate when they represent a solution to the current search strategy by returning an `endstrategy` as their plan operator. A state that contains several `endstrategy` operators indicates that it is the solution to several of the stacked search strategies.

With this machinery in place, we can now define a function to perform stacked search:

Definition 5.5.3 (Stacked Search):

```
fun search childf sstrat st =
  let
    fun subsearch None = []
      | subsearch (Some (sstrat, st)) () =
        case (popsearchop st) of
          None =>
          (childf st) |> add_states sstrat
                      |> pop_state
                      |> subsearch
        | Some(st', addstrategy sstrat') =>
          (Some (sstrat', st'))
            |> subsearch
            |> add_states sstrat
            |> pop_state
            |> subsearch
        | Some(st', endstrategy) =>
          st' :: (subsearch (pop_state sstrat))
  in subsearch (Some (sstrat, [st])) end;
```

The first parameter, `childf`, is a function of type `'n -> 'n list` that given a node produces its child-nodes, and `s` is the initial state of the search captured as an object of type `'n strategy`. The second parameter `sstrat` is the initial search strategy for stacked search to start with.

This approach supports the evaluation of search in a lazy manner which is needed in order to allow a technique to examine another's unfolding in the observational approach to proof planning.

5.6 Extending IsaPlanner's Technique Language

Stacked search is implemented for observational proof planning in IsaPlanner by adding contextual information that holds the search operators. We let the function that computes the child nodes (`childf`) for stacked search be the unfold operation that applies the continuation technique of a state, to that state.

To further support the writing of techniques that specify the locally used search strategy, we add a technique `SEARCH` that uses a strategy to explore the application of another technique. This uses two simple functions: the first, `startsearch`, adds a search strategy to the list of search operators, and the second `endsearch` adds an `endstrategy` operator to the list of search operators. Using these we define the SEARCH technique as follows:

$$\texttt{SEARCH} : \texttt{rstate search} \rightarrow \texttt{rtechn} \rightarrow \texttt{rtechn}$$
$$\texttt{SEARCH}\, s\; r \equiv (\texttt{THENF}\; r\; \texttt{endsearch})\; o\; (\texttt{startsearch}\; s)$$

This takes a search strategy s and a technique r, resulting in a new technique that first indicates that s should be added to stacked search, then performs the technique r as normal. Each final state with no continuation is considered to be a solution to the s by calling endsearch at that point.

5.7 Further Interaction Between Reasoning and Search

We have already mentioned the need for techniques to be able to modify the search strategy being used. It is also useful to employ a more general interaction between reasoning and search. This can be particularly beneficial when conjecturing lemmas as it facilitates encoding heuristics. For example:

- if a conjecture is proved to be false, then the search space of possible alternative proofs for the statement should be pruned. Additionally, the search space of any conjecture that the false one is an instance of, should also be pruned.
- if the search space for the proof of a conjecture is exhausted, then it seems reasonable (and is useful in practice) to avoid making the same conjecture at a later point in proof planning.
- when a lemma is successfully proved, but later the proof of the main goal fails, it will not help to find alternative proofs for the lemma. This suggests that when a lemma is proved, the search space for other proofs of the lemma (or proofs of an instance of it) should be pruned.
- when rippling arrives at a goal that it has already seen, the search space can be pruned of by removing one of the branches. This can result in an exponential decrease in the size of the search space, which otherwise contains significant redundancy. See section 7.9 for more details.

We capture these heuristics using the contextual information to store the lemmas trying to be proved, as well as noting when a search space has been exhausted.

We use reference variables to share information between alternative branches in the search space. This allows dynamic pruning in the same way as the `ENDSPACE` element in the technique language, described in chapter 3, which modifies the last explored state independently of the search strategy being used. This information derived from the reference variable is employed to cut away parts of the search space. For example, if we prove a conjecture we can cut away the rest of the search space if we are not interested in alternative proofs. If we do wish to find alternative proofs, then we can use a global flag disable such pruning.

We remark that this interaction is fully compatible with stacked search and we use it in combination to perform dynamic pruning of the search space independently of the search strategy employed.

5.8 Related Work

The Omega family of systems fixes the search strategy to being essentially depth first, although it provides several layers in which techniques can be encoded. The complex machinery used to interpret these techniques then provides the ability to produce behaviour that would not normally be possible using only depth first search. For example, Omega's backtracking strategies allow the proof planning to stop a proof attempt and return to an earlier subgoal. The complex machinery makes it difficult to change the search strategy, and thus makes it hard to experiment with the effect of different search strategies.

The λ*Clam* proof planner also used depth first search. Similarly to Omega its use of proof critics supports more complex behaviour, but makes the implementation of a different search strategies difficult. λ*Clam* does provide a best first methodical which allows a dynamic ordering of sub-methods by some heuristic score. However this does not perform best first search. We are aware of only one experimental, but unavailable, proof planner that supported full best first search, namely a version of the *Clam* system described by Manning et. al. [68].

Most tactic based theorem provers such as HOL and NuPRL, do not have an explicit notion of search space and thus to not provide generic and powerful search facilities. However, the Isabelle proof assistant does have an explicit notion of the tactics which can be applied in a number of ways. This is used to provide search tacticals, which repeat the application of a tactic to create a search tree. The tacticals then search this space. The search space remains implicit within the tactic's application. Other systems such as PVS and ACL2 perform search in a hard-coded manner within individual proof tools. As such they also avoid having a general notion of search space.

This differs from our approach which supports a lazy evaluation of the search tree rather than just the results of the search process. We make use of this lazy evaluation to support techniques that could otherwise not be expressed, such as the `FOLD` technique described in chapter 3, and to provide an interactive tracing tool for technique applications. In other respects, our approach to search behaves in a similar manner to that used in Isabelle's tactic language.

5.9 Conclusions

We have argued for a more flexible approach to search in proof planning. In particular, we have presented an approach that allows encoded techniques to specify the way choice points within their application are searched over. We have presented and examined two solutions. The first is the obvious approach to nest a proof planning attempt within a technique application. The second approach is more sophisticated and maintains the traceability of techniques and the expressivity of the technique language, while providing the same functionality.

In order to support the proposed approach to search, we defined a general functional notion of search strategy that allows strategies with different types of agenda to be combined within a purely functional setting. We then described an algorithm for stacked search and how this can be used within IsaPlanner by making of use of the extensibility of the technique language.

Lastly, we examined another approach to sharing information across or-branches in the search space that is compatible with stacked search. In particular this supports techniques that dynamically prune the search space. The combination of these mechanisms provide a flexible approach to search that we use to experiment with our inductive theorem proving techniques, in chapter 10.

Chapter 6

Proof Plans

In this chapter, we describe and motivate our representation of proof plans. This is based on the Isar language developed by Markus Wenzel [102]. We give a detailed account of our characterisation of Isar and contrast it with the representation used in other proof planning systems. The contribution of this chapter is thus a representation of proof plans as Isar proof scripts that supports their automatic generation and manipulation. However, we found problems with this approach and thus we also outline various issues to guide further work.

6.1 Introduction

In chapter 3, we provided a novel framework for writing and combining proof planning techniques. Our approach abstracts over the representation of the proof plan. Thus it can be seen as a framework for developing proof planners. This chapter completes the picture by describing a particular implementation of proof plans that results in a specific proof planner for Isabelle.

Recall that proof plans, the result produced by proof planning, are the high-level interpreted descriptions of proofs. They are interpreted in the sense that they can be modified by the proof planner and high level in the sense that they abstract over calculus level operations. They are often also called declarative because it has been thought that the slots of a method can be described in a declarative way. This has typically been done by using a logic-programming language in which to write methods.

We first describe early experiments with a simple representation of proof plans as tactic lists. This motivates a new representation of proof plans as a characterisation of a declarative and structured proof language. In particular, we use the Isar language, introduced in chapter 2, which expresses proofs in Isabelle and is designed to be intelligible to humans as well as machine checkable.

The main work in this chapter is to provide a representation of Isar proof scripts that supports their automatic derivation and manipulation. We provide machinery that combines proof search with the generation the proof plan. In particular, we describe how chains of backward reasoning can be represented and automatically transformed into the forward style of Isar proof scripts. We also extend Isar with new commands to support gaps in proof and to express a step that can involve proof planning.

Although we found our machinery sufficient for expressing techniques, such as rippling, when meta variables or fixed parameters are introduced a different approach is needed. Our representation of proof plans also raises questions concerning how much is expressed in proof scripts. We then contrast our representation with that used in other systems. From this analysis and from the limitations of our machinery, we summarise the features we believe are needed for a more flexible representation of proof plans.

6.2 Initial Experiments with Proof Plans as Tactic Lists

We initially implemented proof plans as a list of tactics paired with the proof state resulting from each one's successive execution [37]. Although we found that this worked well for our early proof techniques, the following issues arose when trying to encode more complex ones:

Expressivity: Using Isabelle's theorem object to represent the proof state means that assumptions cannot be referred to unambiguously. We found it particularly useful to be able to refer to assumptions using variables in the programming environment. Using integers that refer to the location of an assumption is problematic as tactics can reorder and introduce assumptions arbitrarily.

Readability: Tactic scripts are difficult to read. To be understood by humans usually requires re-executing them and examining the intermediate subgoals. Providing more readable proof plans is beneficial for the debugging of techniques and is also important for their development. In particular, we would like to support the encoding of common patterns of interaction with the proof assistant.

Independence of sub-proofs: Lists of tactics do not provide an explicit notion of proof blocks. This means that to identify the part of a tactic script that prove a specific subgoal can be non-trivial. This is particularly important when trying to express techniques that work on several branches of a proof simultaneously, such those used for deductive synthesis of induction schemes [18]. Such techniques need to be able to modify part of a proof plan that affects only a specific goal.

The Isar language solves these problems by providing a readable language for writing proofs that explicitly names assumptions and has a clear notion of proof blocks. Because it is the interface to the theorem prover, using it as the language for proof plans also provides a natural way to mix proof planning and interactive proof development. In particular, the technique language for proof planning to be used as a macro language for automating common steps that the user performs.

User interaction with proof planning can also be supported by providing an Isar method that applies a proof planning technique. A salient feature of this is that the user can also examine the steps performed in further detail by examining the generated proof plan. These steps will also be readable as they will be expressed in terms of Isar proof scripts. This is in contrast to traditional Isar methods which simply apply tactics that cannot be examined in further detail.

Although the issues of expressivity, readability, and proof independence motivate using Isar as the language for proof plans, the existing Isar machinery is not sufficient. We need a declarative characterisation that allows manipulation of proofs within the programming environment. The existing approach employed by Wenzel executes scripts using a virtual machine with state transitions [102]. This avoids holding any representation of proof script and thus gives no support to their manipulation. This motivates the development of new machinery to maintain a declarative representation of Isar proofs that supports their manipulation, printing and re-parsing. This is the contribution of the proof plan representation and associated machinery described in the rest of this chapter.

6.3 Proof Plans as Isar Proof Scripts

One of the key difficulties with providing a datatype to describe Isar proof scripts is that the language is extensible. This means that we cannot have the expected correspondence between constructors and elements in the language. Thus the proof plans must be an interpreted object and tools that manipulate them must be able to act appropriately when a script contains an element of the language that was not defined when the tool was written. For instance, a generic proof planning tool for manipulating Isar proof scripts in any logic should still be applicable when working in spefic domains. One such example is Isabelle/HOL which adds a construct, `obtains`, for reasoning about properties of witnesses to existential variables. Our basic machinery for manipulating proof scripts must be independent of the `obtains` command. However, it should also remain applicable after this element has been added to the language.

To support this behaviour, we provide an abstract description of elements in the Isar language using a uniform interface. Proof plans are then lists of these abstract elements. To examine the parameters used by a language element requires interpretation within the programming environment.

The language elements that make up our proof scripts correspond directly to the basic transitions of the Isar state machine, as shown earlier in Figure 2.4 on page 12. The abstract interface for an element of the Isar language is as follows:

Definition 6.3.1 (Abstract Language Element):

- A unique name.
- An associated datatype that holds the parameters of the language element. This provides the interpretable information that proof planning techniques can examine in their manipulation of proof plans.
- An execution function that performs the underlying Isar proof state transition. An instance of the associated datatype is given to the this function in order to perform the Isar virtual machine step.
- A pretty printing function that can be parsed by Isar

For example, the language element corresponding to the Isar command `apply` is:

Name: apply.
Datatype: an interpretable description of an Isar method, as described below.
Execution Function: the Isar apply function given the method held by the datatype.
Pretty Printing Function: this prints "`apply M`" where `M` is the pretty printed method.

The details of implementing such an interface in ML make use of the ability to hide data in the exception type. This trick has been used extensively in Isabelle to support the extension of basic types. To make the provision of such an interface for elements of the language easier and to help avoid errors in their definition, we provide an ML functor to perform the messier details of working with the exception type and provides a uniform interface hiding the specific datatype. This allows us to provide a general notion of a language element within a proof plan as a record with the following fields:

Definition 6.3.2 (Instantiated Language Element):

Kind: A unique name for the kind of this object in order to allow a technique to safely extract the original parameter data.

Hidden Data: The real parameter data hidden within an exception type. This can be extracted only using the function from the defined language element.

Isar Proof State: The Isar state machine after executing this language element. This allows the Isar context to be propagated.

Pretty Printing Function: This is a function from unit type that will pretty print this element of the proof plan. We make it a function rather than a pretty print object (such as a string) in order to avoid the often computationally expensive task of pretty printing each state during the search for a proof plan. In this way, the work involved in pretty printing is left until the point that the user requests the printed proof plan.

The distinction between the abstract and instantiated language elements is analogous to the uninstantiated tactic slot in Clam methods and the instantiated versions which are used to construct the proof plan. Abstract language elements are given data to provide a concrete instantiated versions.

Proof Plans as Proof Scripts with Lemmas

Although our proof plans are essentially Isar proof scripts, at present, we do not include the ability to define new constants and types, but we do support using separate derived lemmas. In particular, a proof plan is represented as a proof script for a given problem with a collection of lemmas.

We remark that the use of named lemmas and named intermediate results allows sections of a proof to be reused. This feature of the Isar language corresponds to Omega's use of a directed acyclic graph for the representation of proof plans. The distinction between trees and acyclic graphs for proof plans is that results can be reused.

The Linearity of Isar

One of the characteristics of Isar is that it allows modification to a proof's context within a given proof attempt. This includes term syntax as well as the configuration of proof tools. Such local information is essential in order to write some proofs in a suitably brief manner. However, it creates additional dependencies within the proof script. For example, adding an intermediate result to the simplification set can make later calls to the simplifier dependent on the intermediate result.

In order to make use of the Isar context, the proof plan must be executed as elements are added. But because we do not know how elements modify the context, it is not possible to identify parts of the proof that are independent. Furthermore, the linear execution of Isar scripts makes it impossible to modify an earlier part of a proof without re-executing later steps. Because of this inherent linearity, our representation of proof plans is just a list of elements.

For example, the proof shown in 2.2 could be broken up into the list of steps shown in Figure 6.1.

Although execution of Isar is forced to be linear, internally Isar uses a block structure. Certain commands, such as `proof`, start new blocks and other commands end them, such as `qed`. Another example is the `next` command which ends one block and starts a new one. However, Isar does not provide any way to directly work with block structures. For the automatic generation of Isar proof scripts, an interpretable characterisation of the block related effects for the language elements could help the management of dependencies. Additionally, having an interpretable representation of blocks external to Isar would allow proof planning machinery to make use of this information. However, this requires significant modifications to the Isar machinery and has thus been left as further work.

The parameters to elements of the Isar language are usually methods and theorems with attributes. We now describe how we capture these aspects of the language in an interpretable way.

Interpretable Attributes

Attributes in Isabelle/Isar are functions of type "`'a * thm -> 'a * thm`" where `'a` is a polymorphic type variable. They modify a theorem and some other kind of object. In practice, they tend to do one or the other, but not both. In Isar proofs, local results and assumptions can be given attributes. For example, the `simp` attribute adds the given theorem to the simplification set but does not modify the theorem. On the other hand, the `symmetric` attribute swaps the left

1. **theorem** sum_of_odds: $\sum_{i<n} 2 * i + 1 = n^2$ (is $?sumto\ n = \_$)
2. **proof** (induct n)
3. **show** $?sumto\ 0 = 0^2$
4. **by** simp
5. **next**
6. **fix** n
7. **assume** IH: $?sumto\ n = n^2$
8. **have** $?sumto\ (Suc\ n) = ?sumto\ n + Suc(2 * n)$
9. **by** simp
10. **also**
11. **have** $\ldots = n^2 + Suc(2 * n)$
12. **using** IH
13. **by** (simp)
14. **also**
15. **have** $\ldots = (Suc\ n)^2$
16. **by** (simp add: power2_eq_square)
17. **finally**
18. **show** $?sumto\ (Suc\ n) = (Suc\ n)^2$
19. **by default**
20. **qed**

Figure 6.1: The list of steps expressing the Isar proof shown in Figure refisar-proof1-fig

and right hand side of an equation.

A criticism of Isabelle/Isar's approach to attributes is that it mixes those that modify the context with those that modify the theorem. For the management of dependencies, it is important to know when the context is modified. But without any distinction between attributes that modify the context and those that modify the theorem, it must be assumed that all attributes modify both. To improve the ability to analyse proof plans, we provide an interpretable notion of attributes that modify theorems separately from those that modify the context. This also supports the use of attributes that modify theorems without having to provide context in which the attribute is applied.

Similarly to Isar commands, we provide an abstract interface for attributes which have:

- A unique name.
- An associated datatype that holds parameters to the attribute.
- An execution function that modifies the context or the theorem.
- A pretty printing function.

For example, the simp attribute mentioned can be characterised as:

Name: simp.

Datatype: This is a constant, either `add` indicating that the theorem should be added to, or `del` removed from, the simplification set.

Execution Function: this adds or deletes rules form the simplification set.

Pretty Printing Function: This prints `[simp]` when a rule that is added to the simplification set and `[simp del]` when it is to be removed from the simp set.

The symmetric attribute is similar:

Name: symmetric.

Datatype: there are no parameters so this is just the unit type.

Execution Function: this applies resolution with the symmetry theorem to change the orientation of the equality.

Pretty Printing Function: because there is no associated data, this always prints as `[symmetric]`.

Other attributes can also be defined similarly. In Isar, attributes can be added in a domain specific manner by defining them within theories. When this is done, to use them in proof planning techniques, an interpretable attribute must also be defined.

Interpretable Theorems

In Isar scripts, theorems can be modified by attributes. As mentioned earlier, many language elements take these modified theorems as arguments. For instance, the `note` command allows a new a given list of theorems with attributes to be bound to a name. For instance, the following command would create a name for the definitions of addition, where the definitions have been resolved with the symmetry theorem.

```
note sym_add_defs = add_Suc[symmetric] add_0[symmetric]
```

We provide an interpretable notion of theorems with attributes. This is simply an abbreviation of a theorem and a list of attributes. However, we also store the intermediate modified theorems so that analysis of the proof plan does not require re-execution of the attribute functions.

Interpretable Methods

Another common parameter to language elements are methods. In Isar, these are functions that are given a proof context, a list of theorems (the chained results), their normal parameters, and finally the theorem which represents the proof state. Such methods result in a theorem sequence with new assumptions for the new subgoals. When the theorem is proved there are no additional assumptions.

Much like attributes and language elements, we provide an abstract interface for methods:

- A unique name.
- An associated datatype which captures the parameters to the Isar method.
- An execution function that applies the method.
- A pretty printing function that produces a string that Isar can parse.

For example, the equational reasoning method that was described in chapter 8, can be characterised in an interpretable way by the following:

Name: subst.
Datatype: a list of interpretable theorems.
Execution Function: applies the Isar substitution method with one of the given theorems.
Pretty Printing Function: This prints as "`subst THMS`", where `THMS` are the pretty printed interpretable theorems.

The Gap Command

As well as providing interpretable versions of the language elements for the existing commands in the Isar language, we also introduce a new command that corresponds to a gap which can optionally be annotated by a proof planning technique. Gaps represent goals that still need to be solved. They are characterised as follows:

Name: gap.
Datatype: an optional proof planning technique.
Execution Function: skips the proof using an oracle.
Pretty Printing Function: This prints as `gap` when no technique is given and as "`gap TECHN`", where `TECHN` is the pretty printed technique name.

The annotation acts as a hint to further proof planning. This notion of gap plays a similar role to unexecuted justification on Omega proof plans, and to goals paired with methodical expression in λ*Clam*. They are all intended to describe how the goal will try to be solved. Thus the intention is that the technique annotating the gap will be applied to derive a proof script script that will then replace the gap. When no annotation is given, this simply indicates that no technique has been suggested to try to fill the gap.

The Proof Planning Command

To allow the user to employ a proof planning technique when writing proof scripts and to allow a hierarchy of proof plans, we provide a new element to the Isar language that abbreviates the use of a named proof planning technique. This is characterised as follows:

Name: pp.
Datatype: a proof planning technique.
Execution Function: applies proof planning with the technique to the open goal.
Pretty Printing Function: This prints as "`pp TECHN`", where `TECHN` is the pretty printed interpretable proof planning technique.

These compound proof planning steps indicate a part of the proof which can be unfolded into a more detailed proof script. They can also be seen as a dual to the `gap` command. When a gap has been filled by the technique that annotated it, then a `pp` command can be used to *fold up* the proof planning into a single step. Where gaps express work still to be done by proof planning, `pp` steps express work that has already been been done.

6.4 The Generation of Proof Plans

In the declarative Isar style, intermediate results are stated before they are justified. However, in proof search, the intermediate results are not known until the proof methods are applied. It is generally impossible to know how such tactics will behave without applying them. Thus, in interactive proof, it is common for users to employ a procedural step which applies an tactic to find out what the generated subgoals are. The procedural step can then be removed and the intermediate result stated and proved. The original step can then be justified from the intermediate one by the exploratory tactic that was used to find the new subgoals.

To support this kind of backward search we provide a notion of exploration for proof plans. This is essentially an abbreviation mechanism for adding steps to the proof plan. An important feature of this is that it supports the non-determinism of proof methods. In particular, Isar methods can result in different possible subgoals. Each of these corresponds to a different way the method can be applied. Our exploration machinery lifts this choice to the level of proof planning in order to support search over the possible chains of backward reasoning and their automatic conversion into a forward style Isar proof.

For example, consider the goal $P(((Suc\ a)+b)+((Suc\ c)+d))$ and the Isar method that performs substitution with the theorem $(Suc\ x)+y = Suc\ (x+y)$, named `add_Suc`. There are two possible ways to apply the method which results in two possible subgoals. These correspond to the following two possible proof scripts that can be generated:

1. **have** $P\ ((Suc\ (a\ +\ b))\ +\ ((Suc\ c)\ +\ d))$ **gap**
   thus $P\ (((Suc\ a)\ +\ b)\ +\ ((Suc\ c)\ +\ d))$ **by** `(subst add_Suc)`

2. **have** $P\ (((Suc\ a)\ +\ b)\ +\ (Suc\ (c\ +\ d)))$ **gap**
   thus $P\ (((Suc\ a)\ +\ b)\ +\ ((Suc\ c)\ +\ d))$ **by** `(subst add_Suc)`

where the “`thus...`” command is an abbreviation for “`from this show...`”. This adds the last result to the list of chained facts in the Isar state which are passed to the show command and then the subsequent method used to prove the goal. After a proof method is applied, the `by` command automatically tries to unify any remaining goals with chained results. This allows the chained intermediate result to prove the remaining subgoals.

If the `by` command's method fails to prove the goal, the by command backtracks over the other possible applications of the method. This is what allows the same method to be used to show different results.

When applying a method that results in multiple subgoals, once these subgoals are proved, they are given explicit names so that they can be added as chained facts. For example, in a proof of a goal P using a rule $[\![Q;\ R]\!] \Longrightarrow P$ which we will refer to as `prule`, two subgoals are introduced. This creates the following proof script to show P from the named proofs of the subgoals:

```
have r1: Q ...
have r2: R ...
from r1 and r2 show P by (subst prule)
```

6.4.1 Chaining Results for Exploration

Although the exploration machinery described above works for many Isar methods, it does not work for all of them. The problem is that when a method is applied in the procedural style it does not guarantee that the method can be used to justify the original goal from chained facts that proved the subgoals.

This is because Isar methods do not treat chaining in a uniform manner and thus such a transformation is not always valid. For example, Isabelle's simplifier uses the chained rules for simplification, which can then cause a goal that was previously provable, to become unprovable. For instance, this occurs in ordinal arithmetic in the proof of $0^{Lim(\lambda u.\ g)+0} = Lim(\lambda n.\ 0^g) * 0^0$.[1] We can simply this to the subgoal $(\lambda n.\ 0^g) = (\lambda n.\ 0 + 0^g)$. However, when it comes to expressing this as an Isar script, we cannot prove the main goal from the chained subgoal because the simplifier uses the intermediate chained result to rewrite the original goal to $(\lambda n.\ 0 + 0^g) = (\lambda n.\ 0 + (0 + 0^g))$.

While this behaviour is desired in some cases, it confuses the effect of chaining and breaks the symmetry between exploring and expressing proofs. In order to clarify this we define what it means for a method to be stable over chaining:

Definition 6.4.1: A method M is said to be stable over chaining when applied to a goal A if it transforms it to subgoals $B_0 \ldots B_n$ and if A can be proved by M when $B_0 \ldots B_n$ are chained facts i.e. when the following is a valid Isar proof:

[1] See section 10.3, page 189 for more details of the formalisation.

have r_0: B_0 **gap**
$\vdots$
have r_n: B_n **gap**
from r_0 ... r_n **show** A **by** M

We can avoid the issue of stability of methods of chaining by using a different style of proof. In particular, we can explicitly apply the intermediate results. For example, instead of the above script, we write:

have r_0: B_0 **gap**
$\vdots$
have r_n: B_n **gap**
show A **by** `(`M`, rule `r_0`, ..., rule `r_n`)`

This ensures that the results $r_0 \ldots r_n$ are not used by the method M. However, results in a larger justification and what we believe is a less readable style of proof. The advantage of this is that it is independent of the behaviour of the exploratory method.

6.4.2 Proof Plan construction with Gaps

Gaps in proof plans are particularly interesting because they introduce two possible ways of working with Isar proof scripts:

- We can incrementally construct unfinished proof plans, passing techniques in that will continue the proof in the reasoning state continuation, or
- We can construct complete proof scripts with gaps, and then fill in the gaps using the techniques annotating them.

Unfortunately the existing Isar framework incurs a problem with the use of gaps. In particular, it is not possible to assume that a gap have been solved and later provide the proof without re-executing the rest of the proof script. This is because of the lack of type quantification in Isabelle's meta-logic, as discussed in chapter 2. A solution to the logical issues has been proposed by Melham [97], but has not been implemented as it requires significant changes to the logical framework.

Another difficulty with the second approach is the lack of block-level proof script management. To solve this, Isar needs a richer notion of modularity at the block level to support block recombination. The proof plan representation would also need a quick way to access the gaps in a proof, for example by using a hash table. Because of these difficulties, we have largely written techniques in the first style, where techniques to solve the gaps are passed as parameters rather than placed within the proof script.

Although we did not have any difficulty in using the first approach, we remark that the second one is still interesting from more than just a logical perspective. In particular, providing annotated gaps gives an explicit notion of how proof planning will continue in the future. This gives more information to techniques and thus may be of particular use in the expression of proof critics such as those developed by Gow [44] which are based on analysing and modifying a plan containing further proof planning commands.

6.4.3 Nice Fresh Names

A significant issue in the automatic generation of proof scripts is the creation of readable fresh names for fixed parameters and intermediate results. Isar provides local name spaces and the execution ensures correctness. However, it is possible to choose names in such a way that a proof cannot be completed. For example, by clobbering the name of a needed assumption.

In theory, a proof script which makes naming errors could be analysed and then modified to enable the proof to succeed. However, such analysis is difficult and tedious. Instead, we manage the names of assumptions and fixed parameters to avoid name clashes. However, to do this efficiently we would want to store name tables within the Isar proof state. At present we use an inefficient generate and test approach with a record of the last generated name. In practice this seems to be sufficient, although it is rather ad-hoc. The provision of a more sophisticated approach to choosing readable fresh names is left as further work and we suggest is one of the tools that future proof planning frameworks should provide.

We note that none of the other approaches to proof planning provide machinery for this. This is perhaps because it is of less importance when automatically generated proof plans are not looked at directly by the user. However, the lack of such machinery also suggests that the produced plans from other proof planning system will sometimes have name confusions.

6.5 Basic Tools for Proof Planning Techniques

We now describe some basic proof planning tools to support the construction of proof plans. In particular, we define functions to aid exploration and to build the context in which a goal is to be proved.

6.5.1 Lifting Methods to Techniques

For exploration, we use a function `OFMETH` which lifts the application of an Isar proof method to the level of a proof planning technique. This is just an instance of the `PPLANOP` technique consturctor introduced in chapter 6 in page 37. It takes an interpretable method as an argument and explores its application to the last gap in the proof plan. It then results in a new proof plan for each way the method can be applied. For example, if the reasoning state to which the technique `OFMETH` *M* is applied, contains the proof script:

show *P* **gap**

Then a method which solves the goal simply replaces the gap with Isar's `by` *M* command, resulting in a single new reasoning state that has the proof plan:

show *P* **by** *M*

In contrast to this, when the method *M* applied to the proof state at the gap results in either the subgoal G_1 or G_2, then the technique `OFMETH` *M* would produce two reasoning states with indemediate goals and new gaps. For this example, it would result in states with the following proof scripts:

1. **have** G_1 **gap**
   thus *P* **by** (*M*)

2. **have** G_2 **gap**
   thus *P* **by** (*M*)

This simple lifting of methods provides a practical and useful way to explore the construction of proof scripts. One issue with this lifting concerns termination. If the underlying Isar methods fails to terminate, then the proof planning technique will do likewise. Ideally, we would like the processes to be separate in order to allow proof planning to set a time limit on the underlying tactic. Alternatively, we can modify existing tactics to include an extra parameters which sets a limit on their activity and thus ensure termination.

6.5.2 Constructing Context

In Isar, the proof of a subgoal with assumptions or fixed parameters is typically written by building the context of the subgoal before proving it. For example, in an simple one-step inductive proof of some property of natural number, $\forall x.\ P\ x$, the step case goal is $\bigwedge x.\ P\ x \Longrightarrow P\ (Suc\ x)$. To prove this in Isar, a user would typically write the following:

```
fix x
assume P x
show P (Suc x) ...
```

More generally, the pattern including the preceding proof method that results in the subgoals is as follows:

```
proof (M)
  fix ...
  assume ...
  ⋮
  show ...
next
  ⋮
next
  fix ...
  assume ...
  ⋮
  show ...
qed
```

In this proof schema, a method M is used as a backward step and the subgoals produced are then given their appropriate context and solved separately. For automatic proof it is useful to write a single function to do this work. We define a function `prove_in_context` which takes a proof method, M, and reasoning technique, r, as parameters and produces the following schematic proof script:

```
proof (M)
  fix ...
  assume ...
  P_1
next
  ⋮
```

```
**next**
  **fix** ...
  **assume** ...
  $P_n$
**qed**
```

where the technique r produces the proof script P_i for the i^{th} goal. In effect, r is responsible for filling in the part of the proof for the subgoals within their context. The function `prove_in_context` does the work of constructing this context: it fixes parameters and assumes premises.

6.6 Exploring Subgoals with Context

Our approach to exploring the generation of Isar proofs is suitable for subgoals which do not introduce any additional context. However, if a subgoal introduces new assumptions or fixed parameters, then there is an additional choice as to what style should be used in the proceeding proof. Furthermore, our exploratory representation as a list of methods cannot express the building of context.

In particular, there are two obvious approaches which we illustrate with the following example. If we are given a goal P which is reduced by method M to the subgoal $\bigwedge x.\ Q\,x \Longrightarrow R\,x$, then in the proof of $R\,x$ we can take x to be fixed and assume $Q\,x$. To do this in an Isar proof script, we can either express the method in the exploration style described above which would produce the following proof script:

```
**have** $\bigwedge x.\ Q\ x \Longrightarrow R\ x$ **gap**
**thus** $P$ **by** $M$
```

If we then add commands to create the context for the intermediate result, this will become:

```
**have** $\bigwedge x.\ Q\ x \Longrightarrow R\ x$
**proof** -
  **fix** $x$
  **assume** $Q\ x$
  **show** $R\ x$ **gap**
**qed**
**thus** $P$ **by** $M$
```

Alternatively, we can use the method explicitly as a backward step:

```
show P
proof (M)
  fix x
  assume Q x
  show R x  gap
qed
```

The second result is briefer, but this is essentially another choice in the style of the produced proof script.

6.7 Proof Representation for Replay

For a user to make use of the generated proof plans, when they are printed as Isar proof scripts they must be parse-able by the Isar proof checking machinery. This requires that pretty printed terms must be parseable. This is also the case for elements of the Isar language, tactics, and proof planning techniques. At present printing and parsing of terms is not symmetric and thus errors are sometimes produced. In practice this has not been a serious issue as terms can simply be printed in a more verbose fashion showing extra type information. A more robust solution is left as further work.

The ability to present the proofs at different levels of detail for the viewer brings into question the representation of proof stored in a file for replay: should the user try to minimise the size of the proofs by expressing them with the fewest number of powerful proof planning techniques; or should they expand the techniques to fully fleshed-out Isar scripts?

We observe that the more verbose the user makes the proof, by explicitly stating intermediate results, the more likely the proofs are to break if the definitions are modified. Being able to capture a proof using a proof planning technique allows the technique to find a new proof when definitions are changed. However, many proofs require user interaction in selecting proof techniques and in the conjecturing appropriate lemmas. Moreover, the purpose of expressing a proof is often in order to present it. Thus the user may want to modify the derived proof plan. Thus there is a tradeoff between the robustness of proof and its presentation.

6.8 Meta Variables

Although Isabelle supports meta variables in theorems, the Isar machinery does not support them in statements. The consequences for proof search in our system are that when exploration involves meta variables or when instantiations are searched for, as occcurs in the proof of existential theorems, these tasks must be performed outside the Isar machinery. The effect of this is that we lose our ability to name intermediate results and build contexts in the usual way.

However, we note that even lifting Isabelle's support to the Isar machinery would fall short of what was needed for synthesis proofs. In particular, synthesis requires being able to refer to meta variables and their instantiation within the programming environment. However, Isabelle's support for meta variables does not provide tactics with information about their instantiation. There is also no way to maintain meta variables between different theorem objects. Thus trying to reuse Isabelle's existing meta variable machinery would force us to represent the proof state only within a single theorem object. This makes it awkward to refer to specific assumptions and fixed parameters.

We note that this level of support for meta variables is partially provided in the Clam and λ*Clam* systems, but not in Omega. The Clam family inherits its support from Prolog and λProlog's management of Prolog variables. The Omega system provides some support for existential variables with constraints which are solved in a lazy fashion. However, we consider this level of support to be somewhat lacking as manipulations to meta variables and their instantiations are not directly available to techniques encoded in these system. We believe that a rigorous approach to the management of meta variables outside is an important area of further work.

6.9 Stylistic Choices in Expressing Proofs

Isar scripts provide the user with choices regarding the presentation of their proofs. Using the Isar language as the mechanism for exploration then introduces stylistic choices into the proof scripts generated by proof planning. For instance, when should intermediate results be included within the main proof and when should they be considered as separate lemmas? Such choices arise in the encoding of techniques because there is no normal form for proof plans.

Thus techniques may not produce proof scripts in the style that the user wants. In the worst case, the script may be complex and ugly. It is hence the responsibility of the technique writer to create techniques that produce proof scripts that the user wants.

However, we note that this approach also supports proof planning machinery that can transform one style of proof into another. This represents a kind of proof by analogy. For the techniques we wrote, we found it fairly easy to make them generate proof scripts that we considered readable. Examining techniques that construct such analogous proofs is further work.

6.10 Type Checking for the Correct of Construction of Proof Plans

A common error made by new users of Isabelle/Isar is to try using a command when it is not applicable. For example, by trying to use a `show` command for an intermediate result rather the `have` command. When writing proof planning techniques similar errors can easily be made. Unfortunately, because Isar produces an error at execution time, such errors in the writing of techniques go unnoticed until runtime.

It would be desirable to provide some means of expressing techniques that allows them to be checked when they are written. However, the success of adding an extra element to a proof script is generally not decidable, so we would not expect to have full compile-time checking. One approach to this problem is to express more information about a technique at the level of types. This idea leads to the approach suggested by Pollack [88] which makes use of dependent typed language to express techniques so that applicability becomes a check that avoids the need to actually apply the tactic. To be practical, this approach requires powerful machinery to prove the correctness of tactics. However, we note that even a lighter-weight approach would still be able to greatly help in the writing of techniques. In particular it could avoid many of the errors we got at runtime. In this aspect our course-grained representation of elements in the Isar language is problematic as it makes it impossible to get type-checking to verify the correctness of techniques. One approach to to represent more information about Isar language elements, such as the mode, at the level of ML types and in this way try to make use of ML's type-checking to partially verify the correctness of encoded techniques. We believe this is an interesting area of further work.

6.11 Related Work

The Clam, λ*Clam*, and Omega proof planners each use different representations of proof plans. We introduce these and note their features and the ways in which they differ from our use of Isar proof scripts.

Clam

The basic elements of a proof plan in the Clam systems are instantiated methods. The proof plan is simply a tree of these, which can be represented by the following datatype:

```
datatype proofplan = method | THEN of method * proofplan list;
```

where the `method` type corresponds to an instantiated Clam method. When planning is complete, each of these has a fully instantiated tactic slot. The `THEN` constructor indicates that the method resulted in subgoals. These are solved by the given list of proof plans. This representation is essentially a tactic tree. In Clam, the goals are not explicitly represented in the plan. Instead, the planner manages this information outside of the proof plan.

Clam allows methods to employ sub-methods. This creates a hierarchy of method applications. Each method is then responsible for the portion of the proof plan that it constructs. Clam's representation of proof plans allows methods to contain arbitrary parameters, including further proof plans. However, there is not a defined language for this hierarchical structure. Each method can provide its own ad-hoc data representation. Thus some methods have arguments detailing their parameters and others haves sub-plans and other have both. For example, see Figure 6.2 which shows the Clam proof plan for the theorem $a+(b+c)=(a+b)+c$.

The ad-hoc nature of the method-submethod structure meant that when proof critics were later developed, the structured representation of proof had to be abandoned. This is because the critics needed a representation of the state of proof planning that they can interpret and modify. This caused a separate branch of the Clam system to be developed for proof critics, but which lacked a structured method hierarchy. In this version, the representation of proof plan is simplified by not having a hierarchical structure. Instead it is simply a tree of tactics. Proof critics can be attached to methods easily as the critic no longer has to examine a complex hierarchical representation of the proof.

The main branch of the Clam proof planner without critics works directly with the proof plan by adding methods to leaf goals until every goal has been proved. The proof planner maintains the proof plan and its representation in the agenda. However, this makes employing different search strategies somewhat complicated. Each search strategy has to maintain the close relationship between the agenda and the the proof plan as well as the generated tactic tree, rather than use a more convenient representation for the agenda. This makes the search strategies relatively long complicated pieces of code. For instance breadth first search has to store two kinds of agenda in order to manage backtracking which makes the code 76 lines of Prolog code (and 100 lines of comments to explain it). In contrast to this, using the traditional approach with the agenda as a list, results in breadth first search being implemented in 4 lines.

Another disadvantage of this representation is that forward reasoning is poorly supported. In particular, when reasoning forward, the unchanged assumptions and the goal's are not distinct from the modified assumption(s) and there is no clear way to refer to assumptions ex-

```
proof_plan( /* a + (b + c) = (a + b) + c */
[]==>a:pnat=>b:pnat=>c:pnat=>plus(a,plus(b,c))=plus(plus(a,b),c)
in pnat, assp, 940, ind_strat(
induction(lemma(pnat_primitive)-[(a:pnat)-s(v0)])
then
  [base_case(
     normalize_term([reduction([1,1],[plus1,equ(pnat,left)]),
                     reduction([1,2,1],[plus1,equ(pnat,left)])])
     then [elementary((intro(new[b]) then wfftacs)
                      then (intro(new[c]) then wfftacs)
                      then unfold_iff
                      then identity)]),
   step_case(
      ripple(direction_out,
             wave(direction_out,[1,1],[plus2,equ(pnat,left)],[])
             then [wave(direction_out,[1,2,1],[plus2,equ(pnat,left)],[])
             then [wave(direction_out,[2,1],[plus2,equ(pnat,left)],[])
             then [wave(direction_out,[],[cnc_s,imp(right)],[])]]])
             then [unblock_then_fertilize(strong,
                             unblock_fertilize_lazy([idtac])
                             then fertilize(strong,
                                            pwf_then_fertilize(strong,
                                            fertilization_strong(v1)))))])
  ]), dplan).
```

Figure 6.2: *A Clam proof plan that proves the theorem $a+(b+c)=(a+b)+c$.*

```
theorem assoc_plus: a + (b + c) = (a + b) + c
proof (induct' a)
  fix b :: N and c :: N
  show 0 + (b + c) = 0 + b + c by (simp)
next
  fix a :: N and b :: N and c :: N
  assume IH[rule_format]: ∀ c2 b2. a + (b2 + c2) = a + b2 + c2
  have a + (b + c) = a + b + c by (rule IH)
  hence Suc (a + (b + c)) = Suc (a + b + c) by (subst nat_inj)
  hence Suc (a + (b + c)) = Suc (a + b) + c by (subst add_Suc)
  hence Suc (a + (b + c)) = Suc a + b + c by (subst add_Suc)
  thus Suc a + (b + c) = Suc a + b + c by (subst add_Suc)
qed
```

Figure 6.3: *The Isar script version of the proof plan for the theorem $(a+b)+c = (a+b)+c$. This script uses a slight modification to Isabelle's induction tactic which is described in chapter 9.*

plicitly. However, most proof techniques developed in Clam are backward and so this has had little effect on the implementation. Another criticism is that the generated proof plans are rather large and unreadable. For example, contrast that shown in Figure 6.2, with the IsaPlanner proof plan shown in Figure 6.3.

A salient feature of the Clam systems is that they can make use of the underlying Prolog language to manage meta variables. These are simply Prolog variables. However, while this is convenient as much of the work of ensuring freshness and avoiding name conflicts is handled automatically, we note that it also means that the proof environment lacks direct support for manipulating meta variable instantiations. In particular, the only way to undo an instantiation is by backtracking. To undo the first of several instantiations is thus not possible.

In terms of executing the generated proof plasn, Clam uses a separate system, Oyster [21], to apply the the tactic part of the generated proof plans after proof planning has finished. The underlying calculus is a version of Martin Lof type theory closely related to that of NuPrl [30].

λ*Clam*

The λ*Clam* system also uses a tree of instantiated methods to represent the proof plan. However, rather than the ad-hoc language used by Clam, it provides a clearly defined type for proof plans based on its notion of methodicals. The proof plan is still used as an agenda as described in Richardson's lazy interpretation of methods [90]. This involves storing a methodical expression with each open goal. Each time a methodical expression is applied, it is translated into a single atomic method to be applied first, and a continuation methodical expression to be considered next. This allows proof planning to proceed incrementally and supports both a structured method hierarchy and the application of proof critics to atomic methods.

This was used by Gow to develop techniques than manage several proof attempts within the agenda for the deduction of induction schemes during rippling proofs [44].

Proof plans in the main branch of λ*Clam* have not yet been used to drive an object level proof checker. However, in a branch of it for quantified modal and temporal logics, described by Castellini [24], a separate tactic based theorem prover was developed and used to verify the generated proof plans.

Similarly to Clam's use of Prolog, λ*Clam* makes use of λProlog to manage meta-variables. Like Clam, a problem with this is that the only way to remove an instantiation is by backtracking.

The Omega Plan Data Structure

The Omega system holds a representation of the proof plan in terms of a structure refereed to as the Plan Data Structure (PDS) [27]. The PDS is a directed acyclic graph that represents a single proof attempt. The graph is represented as a table of lines. Each line contains a label used to index it in the table, a term representing the statement at that line, and an optional list of justifications. These justifications are either a method, a tactic, or a natural deduction rule. Each justification contains a list of references to other line numbers. These line numbers indicate the start of a proof section. In this way justifications are equivalent to proof blocks in Isabelle/Isar proof scripts.

The PDS does not provide any management of meta-variables. In applications of Omega to Residue Classes and Permutation groups, meta-variable-like machinery was used to hold constraints which are solved at the end of the proof attempt [29, 59, 69]. This then provided concrete initiations in the proof plan. Support for incremental instantiation is largely left to the developer of the proof technique.

An advantage of the PDS representation over that used in the Clam systems is that forward reasoning is supported efficiently. Intermediate results and assumption can be referred to unambiguously by a static line number. Isar offers the same feature by providing explicit names for assumptions and intermediate results. However, Isar provides no machinery for ensuring uniqueness of names. This has to be developed outside of Isar, within our proof planning machinery. Unlike the Clam systems, the PDS does not double as an agenda. Although it supports nodes with un-executed justifications, the developer of techniques specifies how these justifications are unfolded. This means that the PDS does not in any way reflect the future state of the proof planner.

Unlike Isar the Omega representation of proofs does not store any local non-proof related information. This means that independence between steps in a proof can easily be identified. This is used by Cheikhrouhou and Serge to support the removal of steps in the proof [27]. However, the lack of the contextual information also means that proofs cannot be as easily expressed in the PDS as in Isar proof scripts. For instance, assumptions cannot be given to the simplification machinery as they can in Isabelle/Isar.

6.12 Towards Ideal Machinery for Proof Plans

We now clarify and summarise the various features that we believe should be provided by the machinery for working with proof plans.

Gaps

Supporting gaps that can be filled in at a later point in time is one way of supporting different levels of abstraction in a proof. It is important in order to allow the proof planner to peruse several subgoals simultaneously. Ideally, gaps should be able to be named and referred to by variables in the language for expressing techniques.

Forward Reasoning and Context Setup

Supporting forward reasoning as well as backward is useful, especially in combination with modifying the proof context, such as adding rules to the simplifier. Providing explicit names for intermediate results and allowing variables in the language for encoding techniques makes the process of writing techniques significantly easier.

Managing Dependencies

In order to manipulate proof plans internal dependencies must be managed. This includes dependencies in the modification to a proof context as well as between meta variable instantiation and subgoals.

Meta-variables

Proofs of existential theorems often involve the introduction of a variable which can be incrementally instantiated. This approach to proof has been used in many proof developments. To automate such proofs meta variables must be able to be placed within a proof plan and manipulated by encoded techniques.

Correctness

If the execution of proof plans fails then it would strongly suggest that proof planning may also produce proof plans for non-theorems. Thus it is important to be able to verify proof plans. We have found that providing fully formal proofs is neither particularly difficult, nor causes a significant slowdown in the proof process. Furthermore, we found it useful to make use of tactics in the Isabelle. We believe this provides a strong argument for interleaving proof planning with the verification of the proof plan.

Nice Fresh Names

A common task in the construction of a proof plan is the selection of names for intermediate results and introduced parameters and meta variables. To provide an effective interface these must be human-readable. They must also be fresh in certain circumstances. To write proof planning techniques it is important to provide tools to manage the generation of such readable fresh names. Because the constraints on the names are essentially within the proof plan, it makes sense for this machinery to be integrated with the proof plan representation.

6.13 Conclusions

In this chapter we have described machinery for representing proof plans as Isar proof scripts. This is motivated by initial experiments using tactic lists to represent proof plans. Our representation provides a concrete implementation of our observational style of proof planning for

Isabelle. It can produce human-readable machine-checkable Isar proof scripts.

We also introduce machinery to facilitate using tactics in an exploratory backward fashion while expressing the constructed proof script in the normal forward manner. We provided some basic tools to automate common manipulations of proof scripts. This gives a basic framework for generating and manipulating Isar proof scripts.

In our presentation of this work, we noted several shortcomings of our representation. In particular, there are difficulties with meta variables, managing dependencies, and ensuring correctness of techniques at the time of their writing. In addition to these issues, further improvements can be made to the exploration machinery and the modularity of proof script composition. We contrasted our approach with existing systems and finally summarised the features that we believe would be needed for a more flexible representation of proof plans.

Chapter 7

Higher Order Rippling

As introduced in chapter 2, rippling is a rewriting technique that employs a difference removal heuristic to guide the search for proof [17]. Typically it is used to rewrite the step case in a proof by induction until the inductive hypothesis can be applied. This technique has been used within the context of proof planning [16] to automate proof in a variety of domains including hardware verification [22], higher order program synthesis [64], and more recently nonstandard analysis [67].

In this chapter we introduce and analyse the two main approaches to rippling. In particular, we look at the *static* approach which annotates differences at the object level, and then the *dynamic* approach which stores the annotations separately. We also examine Smaill and Green's particular approach to dynamic rippling [96] which describe the difference between two terms using *embeddings*. We then present a novel version of rippling in the dynamic style which describes embeddings using a modified term syntax. Our representation provides a closer correspondence with the traditional account of rippling [5], but also maintains enough flexibility to capture Smaill and Green's approach.

We also provide an algorithm for annotating terms using our representation of embeddings and consider various approaches to removing symmetries in the search space. Our approach forms an expressive framework that facilitates experimenting with many variations of rippling, as carried out in chapter 10. This leads to an efficient implementation of rippling suitable for a higher order setting [38], which we describe in chapter 9.

The structure of this chapter is as follows: we first introduce the general principles and terminology for rippling, then we describe the static and dynamic approaches and briefly compare the speed of rule selection. In section 7.4, we outline Smaill and Green's version of dynamic rippling and in section 7.5 we present our version. Following this, section 7.6 describes an algorithm for finding embeddings using our representation. Section 7.7 describes how an specialised notion of depth for term trees can be used to improve the efficiency of rippling and make its behaviour in higher order domains closer to that in first order theories. In section 7.8, we consider the advantages, in terms of the search space size, that are gained by analysing theorems before they are used in rippling. Section 7.9 describes another issue of efficiency arising from symmetries in the rippling search space and section 7.10 describe how dynamic rippling can hold multiple annotations with each goal to thereby further prune the search space. We conclude in section 7.13.

7.1 Static Rippling

We will refer to the rippling mechanism described by Bundy et al. [19], as *static rippling*. In this, annotated rewrite rules that decrease the measure, called wave rules, are generated from axioms and theorems before rippling is performed. These are then applied 'blindly' to rewrite the goal.

In static rippling, annotations are expressed by inserting object level function symbols (identity functions) for wave fronts and wave holes. For example, the function symbols "wfout", "wfin" and "wh" may be used to represent outward wave fronts, inward wave fronts and wave holes respectively. Using these, the annotated term $p(\boxed{g(\underline{c})}^{\uparrow})$, for instance, can be represented using p(wfout(g(wh(c)))).

Static rippling terminates because every wave rule decreases the measure. This also makes it unnecessary to compute the measure after a wave rule is applied. Eventually the annotations will be removed altogether, moved to the top of the term tree, or pushed into the location of a sink. This means that the sinks do not need any explicit object level representation.

Many wave rules can be created from a single theorem - in general, an exponential number with respect to the size of the term. However, once wave rules are generated, fast rule selection can be performed by using discrimination nets [26], for example.

The need for a modified notion of substitution

A formal account for static rippling in first order logic has been developed by Basin and Walsh [5]. They observe that if the normal notion of substitution is used, then it is possible for rewriting to produce strange annotations that do not correspond to the initial skeleton. The resulting effect is that rippling may no longer terminate and, even if it does so successfully then, due to the changed skeleton, fertilisation may not be possible.

For an example of incorrect annotation consider the following:

1. The wave rule: $Y \cdot \boxed{Suc(\underline{X})}^{\uparrow} \Rightarrow \boxed{Y + \underline{Y \cdot X}}^{\uparrow}$

2. The goal $\boxed{Suc(\underline{a})}^{\uparrow} \cdot \boxed{Suc(\underline{b})}^{\uparrow}$, which has the skeleton $a \cdot b$

3. This goal rewrites to the term $\boxed{\boxed{Suc(\underline{a})}^{\uparrow} + \underline{a \cdot b}}^{\uparrow}$, the skeleton of which $(a\ a \cdot b)$ is not even a well defined term.

This problem occurs whenever a variable from the skeleton, in the left hand side of a wave rule, matches an annotated term and also occurs in the right hand side. More generally, the problem comes from unification introducing new annotations into wave fronts. This can introduce incorrect annotations into the rewritten term. Furthermore, as remarked by Basin and Walsh [5], this makes a rewriting based approach to rippling fail to terminate on some examples.

To avoid these problems, Basin and Walsh present a calculus for rippling with a modified notion of term replacement, substitution, and matching. Similarly, Hutter and Kohlhase have presented a 'coloured' version of the lambda calculus with a modified notion of unification and matching [49]. This use of such a modified notion of substitution allows rippling to be performed independent of any contextual information. Rippling simply involves exhaustively using the modified rewriting machinery with wave rules on an annotated goal.

Implementation

Unfortunately, static rippling's use of a modified notion of matching makes existing tools for rewriting, such as fast term matching algorithms, inapplicable. Furthermore, in a system with a fixed logical kernel, such as Isabelle, rippling would then be forced to be performed outside the logic. Once rippling has ended, the steps it took would have to be repeated within the logical kernel. This gives the developer of such an implementation little reuse of existing tools.

Flexibility and Experimentation

Another important issue for the static approach is its lack of flexibility in experimenting with modifications to rippling. We are interested in experimenting with alterations to the rippling measure and with restrictions commonly added to rippling. Unfortunately such changes are not easily accommodated by the static approach. For example, the restriction on inward wave fronts occurring only over subterms which contains a sink needs to be checked as rippling is performed. Thus to implement such a modification to rippling, we either need to further change the substitution algorithm, or abandon the static approach and interleave rewriting with additional checks.

The general reason for the difficulty in experimenting with static rippling is that the measure is implicit during the rewriting process. This means that only measures which can be used to pre-compute the set of valid measure decreasing rules can be used. This is more restrictive than is necessary. For example, consider a measure that sums of the distances between wave fronts and their nearest sink. This measure is well founded, but cannot be expressed in an algorithm that pre-computes the set of applicable rules and then simply rewrites using them. The measure must be computed between each rewrite because it requires examination of the relationship between the annotations and the goal. This requires abandoning the static approach.

Split Wave fronts

A question left unanswered in the initial description of rippling is the treatment of adjacent wave fronts. In particular, they can be annotated separately or collected to form a single compound annotation. For example, consider the skeleton $P(a)$ and the goal $P(k(h(a)))$. This can be annotated either as $P(\boxed{k(\boxed{h(\underline{a})}^{\uparrow})}^{\uparrow})$ or as $P(\boxed{k(h(\underline{a}))}^{\uparrow})$. This difference can lead to different measures and thus different behaviours for rippling. For example, consider the following annotated equations:

1. $P(\boxed{k(h(\underline{a}))}^{\uparrow}) = P(\boxed{g(\underline{a})}^{\uparrow})$

2. $P(\boxed{k(\boxed{h(\underline{a})}^{\uparrow})}^{\uparrow}) = P(\boxed{g(\underline{a})}^{\uparrow})$

The first equation uses compound annotations but does not change the value of the measure. In contrast to this, the second annotated equation splits up the wave fronts which, when the equation is used from left to right, decreases the measure from $([0,2],[0,0])$ to $([0,1],[0,0])$. Thus the second annotated equation could be used as a wave rule but the first cannot. One solution to this is to make the ripple measure count the height of the term in the wave front. This makes the measure invariant to the choice in how the wave front is split.

However, as noted by Bundy et al [17], the treatment of wave fronts in a compound manner can cause matching to fail unexpectedly. For example, given the goal $P(\boxed{Suc(Suc(\underline{x}))}^{\uparrow} + y)$ then the wave rule $\boxed{Suc(\underline{x})}^{\uparrow} + y \Rightarrow \boxed{Suc(\underline{x+y})}^{\uparrow}$ fails to match. There are two solutions described by Bundy et al, namely to introduce new rules that split compound wave fronts, or to put all annotations in a maximally split normal form. The latter solution benefits from decreasing the number of wave rules generated by a theorem, and by decreasing the search space. This simply eliminates redundancy in the search space by putting all annotations in a normal form.

7.2 Dynamic Rippling

An alternative approach to annotations for rippling is taken by Smaill and Green [96], extended by Dennis, Green and Smaill [33], and used by Dennis and Smaill [35] to automate proofs in the domain of ordinal arithmetic. Their approach avoids the need for a modified notion of substitution by recomputing the possible annotations each time a rule is applied. We call this *dynamic rippling*. The key feature of dynamic rippling is that the annotations are stored separately from the goal and are recomputed each time it is rewritten.

The central motivation for dynamic rippling given by Smaill and Green arises from problems with object level annotations when working in the lambda calculus:

- Object level annotations are not stable over beta reduction. For example, consider the term $\boxed{(\lambda x.c)(\underline{a+b})}^{\uparrow}$ which loses the wave hole when beta-reduced to $\boxed{c}^{\uparrow}$. Thus, if wave fronts are expressed at the object level, then it is not possible to use pre-annotated rules and 'blind' and exhaustive rewriting as the rules may not be skeleton preserving after beta reduction. We note that even in Hutter and Kohlhase's approach, which uses a modified calculus, beta-reduction can still change annotations [49]. Thus they still require a separate skeleton preservation check to be interleaved with rewriting.

- In a context with meta variables, incorrect annotations can accidentally be introduced by unification. This happens in a similar way to the strange annotations in static rippling, described in the previous section. Although we are not aware of a fix to this problem in the higher order setting, we believe that a solution may be possible by modifying higher order unification, similarly to Basin and Walsh's alteration to first order substitution. This would involve removing any annotations that are introduced into the context of a wave front.

As well as these motivations, we observe that the dynamic approach also beneficial for implementation and provides a more flexible framework for experimentation:

- Rippling can take advantage of existing tools and does not require a new implementation of substitution or unification. For example, term matching and manipulation utilities are not affected. This allows the use of tools, such as discrimination nets, to provide fast rule selection, without any extra modification.
- The separation of rewriting from the annotation process simplifies experimenting with rippling. For example, limiting inward wave fronts to occurring over subterms containing a sink, can be incorporated naturally either as an additional filter on rewriting or as part of the annotation process.
- Explicitly calculating and representing measures supports a much larger class of them. For example, unlike the static approach, we can experiment with a measure that computes the distance from every wave front to the top of the term tree or the nearest sink. See section 7.12, for a further analysis of such a measure.

While dynamic rippling simplifies some characteristics of an implementation, it also makes an additional requirement that the annotations and measures are independent from the goal. By placing this information outside the object level, a search technique requires tools to hold the additional information. However, one of the goals of proof planning is to provide tools for the management of this extra-logical information. Our observational approach provides contextual information as described in chapter 3.

In terms of search, an interesting feature of dynamic rippling is that multiple annotations result from some rewrites. This opens up the possibility of storing multiple annotations with each goal and thus reducing the size of the search space. We discuss this further in section 7.10.

From a theoretical point of view, dynamic rippling is a more direct implementation of the measure decreasing characteristic of rippling. The measure is represented explicitly and only steps that are measure decreasing are performed. Thus the termination proof simply reduces to proving that the measure is a well founded ordering. This avoids any complicated reasoning about the annotated terms as was needed for Basin and Walsh's approach [5].

α-β-η-Conversion

In systems using HOAS, α-β-η-convertible terms, while being syntactically different, denote the same object. As has been observed above, β-reduction can remove a possible embedding.

However, we note that embeddings are preserved by α-conversion, η-conversion, and also by β-expansion.

Furthermore, we remark that, for any skeleton and term, it is trivial to construct a β-long version of that term such that the skeleton embeds into it. Thus it is clearly not just the fact that an embedding exists that should guide the proof. The idea is that the goal will be rewritten such that eventually the theorem used as the skeleton can be used in the proof. Thus we suggest that, unlike some versions of rippling implemented in λClam [35,96], β-reduction is not treated as a separate rewrite rule. Rather, we propose that the skeleton is β-reduced and then rippling only consider embeddings into terms in β-short short form. This provides a canonical way to treat embeddings in HOAS and gives clear heuristic guidance. When rewriting produces a term, which through β-reduction has no embeddings, then this is counted as a case when there are no embeddings.

The implicit assumption behind this approach is that parts of the term which disappear through β-reduction are not of interest for proof heuristics. Finally, we note that this approach also produces a smaller search space than including β-reduction as a rewrite rule without weakening the strength of rippling. We believe that this provide a natural approach to rippling and the treatment of β-reduction.

7.3 Analysis of Rule Selection for Dynamic and Static Rippling

We now examine the factors affecting the selection of rules to rippling in both the dynamic and static setting. Both static and dynamic rippling slow down the process of rewriting. Dynamic rippling slows down rule selection by requiring that each matching rule is also checked to be measure decreasing after its application. The separation of matching from measure checking causes matching but non measure-decreasing rules to slow down the search process. This can be minimised, as described in section 7.8, by using some filters on the set of rules considered for rippling. Dynamic rippling is also slowed down by recomputing the possible annotations.

For static rippling, the measure decrease is implicit and never needs to be considered. Because all rules have been pre-checked to ensure that their application decrease the measure, static rippling can simply apply these rules. However, the pre-annotation of rules does increase the number of wave rules, which in turn slows down the selection process. Furthermore, the matching process is itself more complicated as it must avoid introducing badly formed annotations. At the very least we need to perform an extra check during matching to correct possible introduction of bad annotations. If we also want to check that wave fronts only occur over

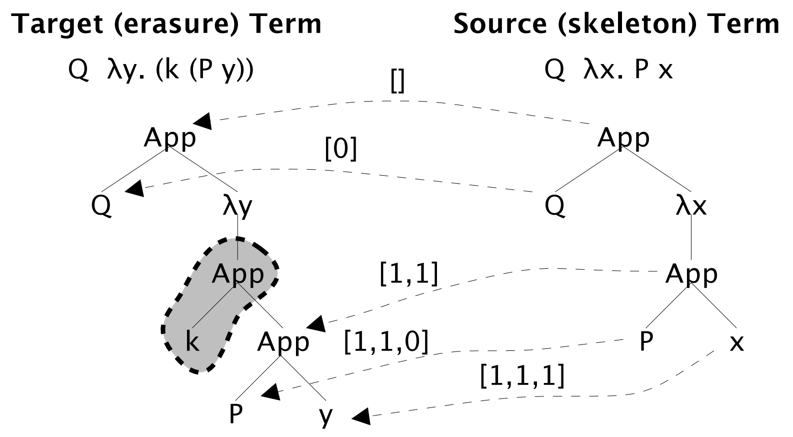


Figure 7.1: *A simple example of Smaill and Green's embedding of* $Q\ \lambda x.(P\ x)$ *into* $Q\ \lambda y.(P\ (k\ y))$. *The grey area corresponds to the part of the term not embedded. This becomes the wave front.*

subterms with a sink, then this time will be larger.

It seems unclear to us if either approach is significantly more efficient that the other. A more detailed analysis is left as further work. However, we note that the results presented in chapter 10 show that dynamic rippling is efficient enough for the domains we examined.

7.4 Smaill-Green Embeddings for Annotating Difference

A central issue in the dynamic approach to rippling is the representation of annotations. Smaill and Green use embedding trees to represent these [96]. Their embeddings are injective maps between term trees that preserve label names as well as the 'horizontal' order. The wave fronts are the parts of the target term not mapped onto by the embedding. The wave holes are implicitly later parts of the term target that are embedded into. Given this correspondence we will refer to the source term on an embedding as the skeleton and the target term as the erasure.

The embeddings are represented by trees that have the same shape as the skeleton (without abstractions) but where the leaves are addresses indicating the location where they are embedded into in the erasure. For their part, the addresses are lists of natural numbers that indicate which sub-branch should be taken.

For an example embedding see Figure 7.1 which shows how the source term "$Q\ \lambda x.\ (P\ x)$" embeds into "$Q\ \lambda y.\ (P\ (k\ y))$". Using the box notation with undirected annotations this embedding can be viewed as "$Q\ \lambda y.(P\ (\boxed{k\ \underline{y}}\ ))$".

Embedding

Smaill and Green define their version of embedding for the possible cases of the target term as follows:[1]

- **base case:** t embeds into t for all atomic t.
- **application:** t embeds into $App(u_1, u_2)$ iff
  1. $t = App(t_1, t_2)$, and t_1 embeds into u_1, and t_2 embeds into u_2 or
  2. t embeds into u_1 or
  3. t embeds into u_2
- **abstraction** t embeds into $\lambda x.\ u$ iff
  1. for some fresh constant z, t embeds into the beta reduction of $App(\lambda x.\ u, z)$ such that the bound variable x is replaced by z, or
  2. $t = \lambda y.\ t_1$, and for some fresh constant z, the beta reduction of $App(\lambda y.\ t_1, z)$ embeds into the beta reduction of $App(\lambda x.\ u, z)$, where the beta reductions replace the bound variables by z.

The different possibilities in the case for application and the case for abstraction result in terms having multiple embeddings. For rippling, this means that annotations are not uniquely defined. These can simply be searched over, although we describe a less naive strategy in section 7.10.

We observe that while the embedding trees do not represent abstraction, the definition of embeddings does place a strict requirement that every abstraction in the skeleton must occur in the term that it is embedded into.

Rippling Measures

For embeddings to be used to guide rippling, a function must be defined on the embedding trees to compute the measure. Although Smaill and Green's initial measure [96] did not correspond to that used by Basin and Walsh [5], a newer version, described by Dennis, Green and

[1]Smaill and Green's presentation includes products and treats them analogously to application.

Smaill [33], when using a tuple rather than curried representation for functions, does correspond to Basin and Walsh's calculus.

This measure for embedding trees can be computed in two steps:

1. A *difference tree* of the same shape as the embedding tree is calculated, where each node is an integer expressing the size of the wave front at that point. The size can be calculated by subtracting one plus the length of the parents address from length of the current address.

2. The rippling measure is a lexicographically ordered list, where the i^{th} element corresponds to sum of the differences at depth i in the difference tree.

For an example, see Figure 7.2 which shows the embedding tree, difference tree, and measure computed from the embedding shown in Figure 7.1.

Directed Annotations

Directed annotations are implemented by adding to each node in the embedding tree a direction. This is ignored if the difference at that node is zero. This allows both the outward and inward measures to be computed and thus for the embedding tree to express similar measures to Basin and Walsh's calculus.

One important departure from the static account of rippling, is that adjacent wave fronts are forced to have the same direction. This might be seen as an advantage because the search space is smaller. However, it also limits the flexibility of the representation. It remains an open question as to whether there is a practical need for adjacent wave fronts with differing directions. In chapter 10 we test this empirically. This provides evidence that in general using compound wave fronts works well.

Strange Embeddings

We observe that Smaill and Green's approach has the strange behaviour that the embedding of a bound variable is not restricted by its associated quantifier. For example, an embedding is possible from the term $\exists x.P\ x$ into $\exists a.\forall b.R\ a\ (P\ b)$ (See Figure 7.3), where the x is existentially quantified in the skeleton, but embedded into b which is universally quantified. We believe that this is due to the lack of a well defined relationship between the annotations for difference and the underlying semantics. However, in practice this is rarely an issue and we have not found any domains where this causes a problem.

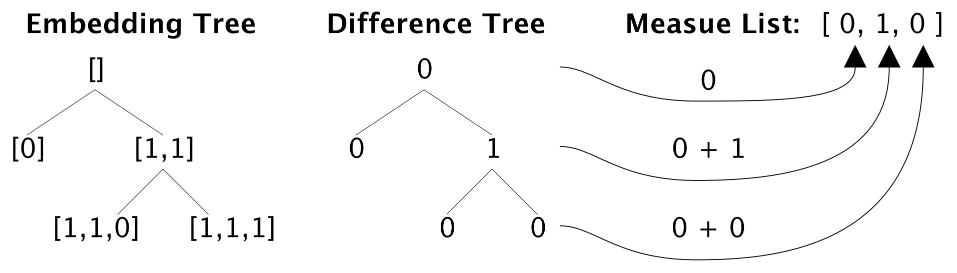


Figure 7.2: *The embedding tree, difference tree and measure computed from the embedding shown in Figure 7.1*

Nonetheless, if rippling is to be used in a domain with many different quantifiers then it may be worthwhile to impose further restrictions on embeddings. For example, by requiring, for each quantified variable being embedded, that the quantifier in the skeleton and the goal should be identical, or that the quantifier in the skeleton should embed into the one in the goal. Such constraints would prune the search space and bring a closer semantic relationship between the embedding of bound variables and their quantifiers.

7.5 Embedding Terms for Annotating Difference

In this section we present an alternative to the embedding trees used by Smaill and Green's to represent rippling annotations. This is motivated by the wish to have a more flexible framework where we can easily experiment with variations of rippling.

Furthermore, we wish to examine empirically the modifications to rippling, including those made by Smaill and Green. In particular, their approach deviates from the first order account given by Basin and Walsh in the following ways:

- Annotations with adjacent wave fronts in different directions, such as $\boxed{f(\boxed{g(\underline{x})}^{\downarrow})}^{\uparrow}$ cannot be expressed with their embedding representation. The reason for this is that the wave fronts are expressed implicitly by the difference in the length of addresses in the embedding tree, described in the previous section.
- When curried function representation is used, the measures no longer correspond to the equivalent first order representations. This results in a different search space and behaviour of rippling. In section 7.7, we describe an interpretation of depth for HOAS that allows the traditional measures to be derived.

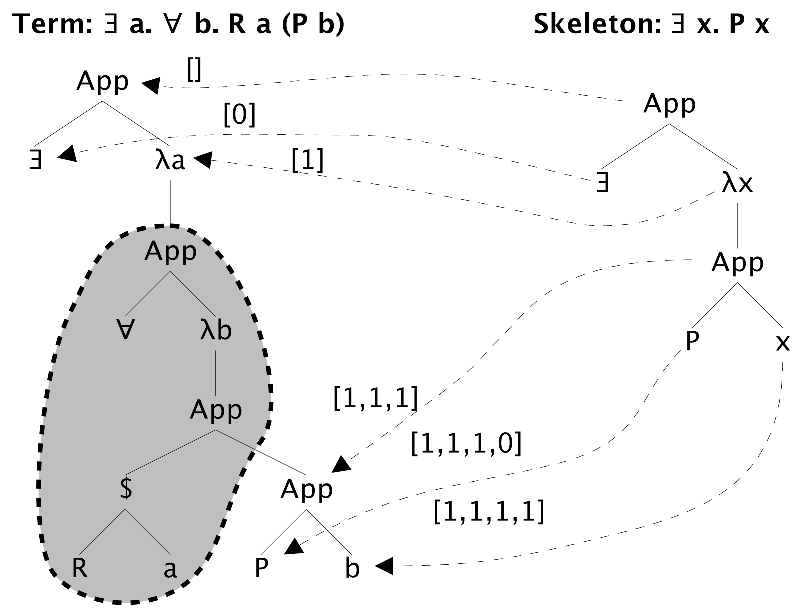


Figure 7.3: *An embedding where the quantifiers do not correspond with the bound variables. The greyed area indicates parts of the term that differ and dotted arrows correspond to the embedding address.*

To examine whether these differences are beneficial or detrimental to proof search, requires a more flexible representation, capable of reflecting Basin and Walsh's approach in a higher order setting. Furthermore, if we wish to reason about rippling itself, we find that embedding trees produce complicated proofs. In particular, we are required to reason about well formedness of embeddings and the result of following addresses through term trees.

We now present a richer representation for embeddings that expresses the annotations and their contents explicitly. Our approach avoids having to consider badly formed embeddings which makes it easier to reason about. It also provides a more flexible, efficient and modular mechanism for expressing annotated terms than embeddings trees.

Embedding Terms

We now define our notion of *embedding terms* for annotating difference. Like embeddings, they maintain an independence from the term calculus. In IsaPlanner, we define embedding terms over Isabelle's terms although in general it is easy to define them over any term syntax. To keep the presentation clear but avoid hiding important details, we use de Bruijn indices and ignore types. With this in mind, we will use the following term syntax:

$$\begin{array}{rcl} \mathit{term} & = & \texttt{Bound}\ \mathit{nat} \\ & | & \texttt{Const}\ \mathit{name} \\ & | & \texttt{Abs}\ \mathit{term} \\ & | & \texttt{App}\ \mathit{term}\ \mathit{term} \\ & | & \texttt{Var}\ \mathit{name} \end{array}$$

Note that we include a notion of meta variable (`Var`). This behaves like a universally quantified variables, and can be instantiated to any term of the appropriate type. This allows sinks in the skeleton term to be expressed using "`Var` *name*", where *name* is the name of the variable. We will abbreviate "`App` P (`Var` x)" to "P $?x$" when writing terms with implicit abstract syntax, following the style of Isabelle's pretty printer. In a term syntax without meta variables, a separate record is needed to record which bound variables are to be universally quantified. However, the treatment for embeddings remains essentially the same.

Embedding terms are terms where the variable constructor expressing a universally quantified variable has an extra term parameter holding the subterm at the location of the sink. If the term syntax does not have such a constructor then the embedding term datatype contains an extra constructor for this purpose. Additionally, all the other constructors of embedding terms have an additional parameter, *upterm*, which holds the annotation and its contents at this point in the term tree. The idea is that the embedding term has the same shape as the skeleton and the upterms capture the annotations. For the term syntax we introduced above, this gives us the following datatype for embeddings terms:

$$\begin{array}{rcl} \mathit{eterm} & = & \texttt{eBound}\ \mathit{upterm}\ \mathit{nat}\ \mathit{nat} \\ & | & \texttt{eConst}\ \mathit{upterm}\ \mathit{name} \\ & | & \texttt{eAbs}\ \mathit{upterm}\ \mathit{eterm} \\ & | & \texttt{eApp}\ \mathit{upterm}\ \mathit{eterm}\ \mathit{eterm} \\ & | & \texttt{eVar}\ \mathit{name}\ \mathit{term} \end{array}$$

Note that the constructor for an embedded bound variable has two natural numbers. These correspond to the bound variables in the skeleton and the erasure respectively. Although we do not need to hold both, as one can be worked out from the other, it is convenient to do so when reasoning about embedding terms. It also simplifies later definitions.

Informally, one can think of upterms as describing the contents of a wave front from the perspective of a wave hole. These are essentially a specific instance of Huet's zippers [47]. For now we will consider upterms that correspond to undirected annotations. This gives us the following datatype:

$$\begin{array}{rcl} \textit{upterm} & = & \texttt{uAbs}\ \textit{upterm} \\ & | & \texttt{uAppL}\ \textit{term}\ \ \textit{upterm} \\ & | & \texttt{uAppR}\ \textit{term}\ \ \textit{upterm} \\ & | & \texttt{uRoot} \end{array}$$

Upterms can also be thought of as expressing a term context. For example, if we consider the term "`App` a b" then, from the perspective of b, the context is "`uAppL` a". Symmetrically, from the perspective of a, we get the context "`uAppR` b". The upterm constructor `uAbs` corresponds to being in the context of a lambda abstraction and `uRoot` indicates that there is no further context.

Directions can easily be added to the upterm a datatype by including an additional argument in the appropriate constructor(s). This provides a flexible solution to describing annotations which can also express adjacent wave fronts with different directions.

For rippling, this notion of upterm for the embedding term allows the expression terms where the undirected annotations are expressed in the upterms. For example, consider the skeleton "$\forall z.\ P\ c\ z$" which is expressed as "$P\ c\ ?z$". This uses a meta variable for the universally quantified z. In the term syntax, this is written explicitly as follows:

```
App
   (App
      (Const P)
      (Const c))
   (Var z)
```

Given this as the skeleton, the term $P\ (f\ c)\ (g\ y)$ can be as annotated as $P\ (\boxed{f\ \underline{c}}\ )\ \lfloor (g\ y) \rfloor$, which corresponds to the embedding shown in Figure 7.4. This is be expressed as the following embedding term:

```
eApp uRoot
   (eApp uRoot
      (eConst uRoot P)
      (eConst (uAppL (Const f) uRoot) c)
   (eVar (App (Const g) (Const y)) z))
```

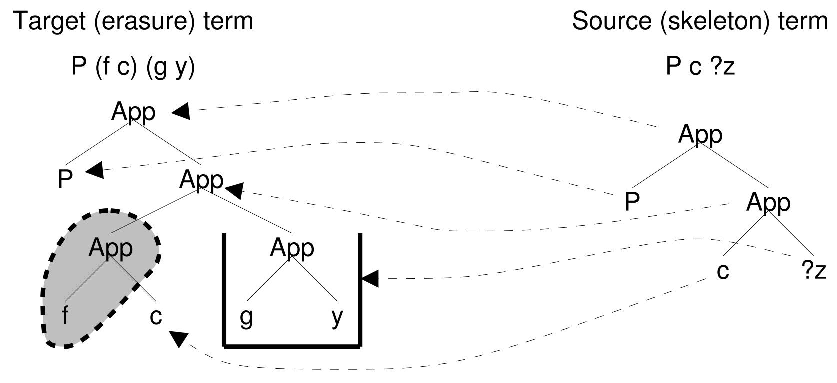


Figure 7.4: *The embedding of "P c ?z" into "P (f c) (g y)", where ?z is treated as a sink.*

Basic Operations on Embedding Terms

One of the advantages of embedding terms over embedding trees is that the typical operations on annotated terms can easily be defined without having to interpret term addresses. In particular, getting the skeleton and erasure of an embedding term are simple recursive functions. Computing the skeleton of an embedding term involves simply ignoring the upterms and contents of sinks:

```
skeleton_of_eterm  (eBound u n1 n2)  =  Bound n1
                 | (eConst u m)      =  Const m
                 | (eAbs u e)        =  Abs (skeleton_of_eterm e)
                 | (eApp u e1 e2)    =  App (skeleton_of_eterm e1)
                                            (skeleton_of_eterm e2)
                 | (eVar m t)        =  Var m
```

Finding the erasure of an embedding is slightly more involved and requires incorporating the context represented by the upterm. We do this by first defining a function that, given an upterm and a term, places the term within the the upterms context:

```
apply_upterm  (uAbs u) t     =  apply_upterm u (Abs t)
            | (uAppL l u) t  =  apply_upterm u (App l t)
            | (uAppR r u) t  =  apply_upterm u (App t r)
            | uRoot t        =  t
```

Using this, we can now define a function that is given an embedding term and results in the erasure:

```
erasure_of_eterm  (eBound u n1 n2)  =  apply_upterm u (Bound n2)
                | (eConst u m)      =  apply_upterm u (Const m)
                | (eAbs u e)        =  apply_upterm u
                                         (Abs (skeleton_of_eterm e))
                | (eApp u e1 e2)    =  apply_upterm u
                                         (App (skeleton_of_eterm e1)
                                              (skeleton_of_eterm e2))
                | (eVar m t)        =  t
```

In the following section we present an algorithm for finding the embeddings and representing them as embedding terms.

7.6 An Algorithm for Finding Embedding Terms

Having introduced a representation for expressing embeddings in the previous section, we now describe a general algorithm for computing possible embeddings. We present this with respect to the previously introduced term syntax.

Our embedding algorithm is given a source term (the skeleton) s and a target term t (the erasure). It results in a list of possible embeddings, expressed as embedding terms. The basic idea is to traverse the erasure looking for a location where we can embed the next constructor of the skeleton.

Our embedding algorithm takes two additional arguments, namely the annotation at this point in the embedding process which is expressed as an upterm, u, and a boolean list, bl, that relates abstraction in the skeleton term to abstractions in the target term.

Notation

We will use `false` and `true` for their respective boolean values. We will let :: and [ ] represent the cons operation and the empty list respectively, although we will also use the common list abbreviation $[a,b,c,...]$ for a :: b :: :: c :: ... ::[ ]. The infix append operator will be written as @. For pattern matching, we will use "_" to match any constructor not previously described.

Boolean Lists as Abstraction Contexts

An essential part of the embedding process involves managing the local context, namely the abstractions. We use boolean lists to hold this information. The intended interpretation is that the empty list indicates that there are no further abstractions in either the skeleton or the erasure. A `false` in the list indicates that the target contains an abstraction that has no corresponding abstraction in the source term. In contrast to this, `true` indicates that there is an abstraction in the skeleton that embeds into an abstraction in the erasure. The list is ordered from the current location to the root.

For example, consider the embedding of the skeleton "$\mathtt{App}\ P\ (\mathtt{Abs}\ t)$" into the erasure $\mathtt{Abs}\ (\mathtt{App}\ P\ (\mathtt{Abs}\ t))$. The context of the subterm t is $[\mathtt{true}, \mathtt{false}]$. This indicates that the previous abstraction is in both the skeleton and the erasure, but the abstraction before that is inside a wave front.

Given such a boolean list and two bound variable indices, we define a function `check_bnds` to determine if two bound variables correspond to each other with respect to the embedding. We compute this by moving up along the boolean list that represents the lambda abstractions correspondence. When both the indices of the bound variables correspond to the current abstraction (they are both zero), we simply examine the value at the head of the list which defines their relationship:

$$\mathtt{check\_bnds}\quad (\mathtt{b} :: \mathtt{bl})\quad 0\quad 0\quad = b$$

In this case the skeleton bound embeds into the erasure bound when the the abstractions correspond. Interestingly, this suggests that while the abstractions are not explicitly represented by the embedding trees of Smaill and Green [96], there must be an implicit correspondence between abstractions.

The other base case is when boolean list is empty:

$$\mathtt{check\_bnds}\quad [\,]\quad n1\quad n2\quad = \ (n1 = n2)$$

This indicates that there are no more abstractions. If this case is reached then the bound variables have no corresponding abstractions. We then assume that they are within the same context and simply check that their indices are equal. Note that if we intend only to work with valid terms then this case need not be considered as the embedding algorithm will never produce such a context.

Lastly, we consider the recursive cases where we have at least one abstraction and the bound variables are nonzero:

$$\begin{array}{rlllcl} \mathtt{check\_bnds} & (\mathtt{true} :: \mathtt{bl}) & n_s & n_t & = & \mathtt{check\_bnds}\ bl\ (n_s - 1)\ (n_t - 1) \\ | & (\mathtt{false} :: \mathtt{bl}) & n_s & n_t & = & \mathtt{check\_bnds}\ bl\ n_s\ (n_t - 1) \end{array}$$

We update the bound variable by removing the head boolean value from the list. If its value is true then it indicates that the last abstraction considered was embedded from the skeleton into the target, and thus removing it will decrease the index of both the skeleton and erasure variables. On the other hand, if the value removed from the list is false, then it indicates that the previous abstraction occurs in the wave front, and thus only the erasure bound variable is decreased.

An Embedding Algorithm

We will now present the various cases for the algorithm to compute embeddings. The algorithm results in a list of possible embeddings. We first consider the case of embedding a bound variable:

```
embeddings  $u\ bl$ (Bound $n_s$) (Bound $n_t$) =
            $if$ (check_bnds $bl\ n_s\ n_t$) $then$ [eBound $u\ n_s\ n_t$] $else$ []
          | $u\ bl$ (Bound $n$) (Abs $t$) =
            embeddings (uAbs $u$) (false :: $bl$) (Bound $n$) $t$
          | $u\ bl$ (Bound $n$) (App $t_1\ t_2$) =
            (embeddings (uAppR $t_2\ u$) $bl$ (Bound $n$) $t_1$)
            @ (embeddings (uAppL $t_1\ u$) $bl$ (Bound $n$) $t_2$)
          | $u\ bl$ (Bound $n$) _ = []
```

Embedding a bound variable succeeds only when the erasure term is a bound variable and has the same corresponding abstraction. If the erasure is another kind of leaf then embedding fails. Upon success the annotation is held in the upterm u.

The recursive calls occur when we try to embed into an abstraction or an application. For abstraction, we need to update the abstraction correspondence list to note that we have an abstraction in the erasure without a corresponding one in the skeleton. For application, we give back the embeddings into the right appended onto those from the left branch. For both abstraction and application we add the appropriate constructor to the upterm before we recurse. This notes that we have increased the upterm which expressed the wave front.

The embedding of constants is similar to bound variables:

```
embeddings   u bl (Const m_s) (Const m_t) =
               if (m_s = m_t) then [Const u m_s] else []
           | u bl (Const m) (Abs t) =
               embeddings (uAbs u)(false :: bl) (Const m) t
           | u bl (Const m) (App t_1 t_2) =
               (embeddings (uAppR t_2 u) bl (Const m) t_1)
               @ (embeddings (uAppL t_1 u) bl (Const m) t_2)
           | u bl (Const m) _ = []
```

The only difference is that to check if one constant embeds into another, we only need to check that they are the same constant.

Embedding of a meta variable, which represents a sink and corresponds to a universally quantified variable in the skeleton, is even simpler:

```
embeddings u bl (Var m) t = [eVar (apply_upterm u t) m]
```

This indicates that a sink embeds into any term. Furthermore, we place any wave front in the current upterm into the sink, using the `apply_upterm` function. As mentioned earlier, in a logic with polymorphism or type classes, sinks also have wave fronts, in which case we use the type information to decide what can be placed in the sink. In practice this arises very rarely as it requires a skeleton of more abstract type that the goal.

The case of embedding an abstraction is:

```
embeddings   u bl (Abs s) (Abs t) =
               (map (eAbs u) (embeddings uRoot (true :: bl) s t))
               @ (embeddings (uAbs u) (false :: bl) (Abs s) t)
           | u bl (Abs s) (App t_1 t_2) =
               (embeddings (uAppR t_2 u) bl (Abs s) t_1)
               @ (embeddings (uAppL t_1 u) bl (Abs s) t_2)
           | u bl (Abs s) _ = []
```

The interesting sub-case is that of embedding an abstraction into an abstraction. If the abstractions correspond to each other then any embedding of the contents on the skeleton abstraction in the content of the target one produces a valid embedding. On the hand, if abstraction do not correspond and then we look for embeddings in contents of the target and note that the previous abstraction in the target was not embedded into. Both lists of possibilities are found and appended to give a list of all possible embeddings.

Embedding an application involves a similar combination of sub embeddings:

```
embeddings  u bl (App s1 s2) (Abs t) =
            embeddings (uAbs u) (false :: bl) (App s1 s2) t

         |  u bl (App s1 s2) (App t1 t2) =
            (merge_embs u (embeddings bl uRoot s1 t1)
                          (embeddings bl uRoot s2 t2))
            @ (embeddings (uAppR t2 u) bl (Abs s) t1)
            @ (embeddings (uAppL t1 u) bl (Abs s) t2)

         |  u bl (App s1 s2) _ = []
```

The `merge_embs` function simply combines every embedding of the first list with every one of the second. It uses the current upterm as the annotation for the combination.

We now consider various issues related to the embedding algorithm.

Unification of Sink Variables

Our embedding algorithm makes no requirement that the different subterms at the location of a sink unify. This facilitates the later application of proof critics that try to fix fertilisation failure. Producing no embeddings would make it impossible to tell *why* rippling had failed without further analysis.

Meta Variables

It is interesting to consider the treatment of meta variables in the erasure. In the above presentation of our algorithm algorithm we only allow meta variables to embed into meta variables. However, in order to allow fertilisation, it suffices to allow the embedding of any term into a meta variable. If there are multiple occurrences of the meta variable then fertilisation will require that they unify. This can be treated in a similarly to sinks, delaying the unification check and allowing a proof critic to correct such failures.

In order to allow meta variables in the erasure to be represented in the embedding term, we need an extra constructor, `eTVar`, similar to `eVar`, but whose interpretation for skeleton and erasure is reversed:

$$\begin{aligned} \texttt{skeleton\_of\_eterm} \quad (\texttt{eTVar}\ m\ t) &= t \\ \texttt{erasure\_of\_eterm} \quad (\texttt{eTVar}\ m\ t) &= \texttt{Var}\ m \end{aligned}$$

When the embedding algorithm considers a meta variable in the erasure, it then constructs a single embedding with the `eTVar` constructor.

In a setting with polymorphism or a type hierarchy, we have to take additional care with the embedding to or from meta variables. In particular, we observe that meta variables may need to be provided with an argument to express a wave front. This occurs when the type of a compound term in the target is not a subtype of the skeleton term. For example, consider:

1. The skeleton "$f\ ?x$" where f has polymorphic type $\alpha \to \alpha$ and $?x$ has type *nat* (an instance of the type α).

2. The erasure "$f\ (g\ y)$" where g has type $nat \to bool$.

3. The embedding which produces the annotation "$f\ \lfloor (g\ y) \rfloor$".

In this case fertilisation is not possible because "$g\ y$" has type bool, but needs type *nat* to match the skeleton. However, the alternative annotation "$f\ \boxed{(g\ \lfloor y \rfloor)}^{\uparrow}$" reveals that that the function g requires further rippling.

To incorporate this into our embedding terms, we only need to provide an extra upterm argument to the constructors expressing meta variables. This then holds the part of the term that is outside of the appropriate meta variable's type.

η-normal form

We observe that the target term must be in η-long form before embeddings are calculated. If this is not the case, then examples such as the embedding of "$\lambda x.f\ x\ y$" into "$(g\ f)\ y$" fail. If the skeleton is η-contracted then additional annotations will sometime be introduced. In order to avoid these problems we compute the η-long normal form of the skeleton and erasure before using the embedding algorithm. We can also combine the process checking the η-long normal form with the embedding algorithm. This is done by storing additional information about the path to the current location in the target and erasure.

Directed Annotations

This embedding algorithm does not consider directed annotations. These can be incorporated into the algorithm easily, or expressed separately as a second process. In IsaPlanner, we do the latter as this provides us with the ability to implement the measure computation in different ways and to consider the possibility of using a measure that uses undirected annotations. In particular, if we separate this process, then we only need a single implementation of embedding.

7.7 A Notion of Depth for Measures and Inward Rippling

Smaill and Green's presentation of embeddings includes tuples in the term syntax. As mentioned earlier, their motivation for this is to minimise the number of embeddings. An additional effect of this is that the measure is not the same for a function that takes curried arguments as one that uses tuples. This makes the use of the tuple representation essential for their representation to produce the same measures as Basin and Walsh's account.

The measure is defined by the notion of depth. This notion is also needed to express the restriction that only allows inward wave fronts to being placed *above* a subterm that contains a sink. In particular, for a higher order abstract syntax (HOAS) the idea of 'above' or 'below' is not immediately obvious as function symbols are leaf nodes in the term tree.

We define a suitable notion of depth which also makes the measures robust over the choice of tuple or curried argument representation. Additionally, our approach results in the same measures as the first order account. In chapter 10 we examine experimentally the effect of using this notion of depth in comparison with that used by Smaill and Green.

The central idea is to treat depth in the following way:

- the root of a term tree has depth 0.
- if $\lambda y.u$ has depth d then u also has depth d.
- if $App(u,v)$ had depth d then u has depth d and v has depth $d+1$

This 'uncurries' the syntax in the way we would expect: no height ordering is given to different curried arguments of a function. For example, the term $Suc(a)+b$, expressed in the HOAS as $App(App(+,App(Suc,a)),b)$, gives a depth of 0 to $+$, 1 to Suc and b, and 2 to a. In contrast, the usual notion of depth in HOAS is 1 for b, 2 for $+$, and 3 for Suc and a.

7.8 Selection of the Wave Rule Set

One of the advantages of rippling over simplification based approaches without an ordering, is that the annotation process provides a means of ensuring termination and that therefore all axioms and proved theorems can be used.

Static rippling facilitates this by first generating only measure decreasing wave rules from theorems and axioms. While the number of generated rewrites is exponential on the size of the term, it avoids using equations oriented in a way which has no valid annotation, such as $x = 0 + x$ and $0 = 0 * x$.

In dynamic rippling, theorems are used to transform the goal and then the possible annotations are checked in order to avoid steps which do not decrease the measure. Unfortunately, this approach can causes equations such as $x = 0 + x$ which are not beneficial but frequently applicable to significantly slow down search.

One solution to this is to filter the possible ways an equation can be used, removing those with a left hand side that is identical to a subterm of the right, such as $x = x + 0$. It is also beneficial to remove those that would introduce a new variable, such as $1 = x^0$. This is a simple and quick approach based on the notion of a valid rewrite rule.

Another approach is to look at every possible skeleton which can embed into the left hand side. This can then be used to check if a measure decreasing annotation exists for the right hand side. If so, then the rule can be used during rippling. This corresponds more closely with static rippling but is more complex and slower. In practice, it seems to result in the same behaviour as the simpler mechanism.

7.9 Avoiding Symmetries in Rippling Search

A simple observation which can be made during rippling is that it is often possible to ripple many different parts of a goal independently, and thus it is sometimes redundant to explore both branches of the search space. For example, in the proof of the commutativity of addition presented earlier, either the right hand side or the left hand side can be rippled out first, as shown in Figure 7.5

These are independent choices in the search space. Both lead to the same final fully ripped-out goal $\boxed{Suc(\underline{a})}^{\uparrow} + b = \boxed{Suc(\underline{b + a})}^{\uparrow}$.

One solution is to cache the unannotated goals, so that the same rippling state is not examined more than once. This removes symmetry in the search space and thus provides an efficiency improvement. The size of the search space would naturally increase exponentially, the benefit yielded from caching is also exponential.

While the caching approach provides significant improvement, it still includes some of redundancy in the search space. In particular, redundant branches are explored and only pruned

$$\begin{tikzcd}
 & \boxed{\begin{array}{c} \boxed{Suc(\underline{a})}^{\uparrow} + b = b + \boxed{Suc(\underline{a})}^{\uparrow} \\ \text{Measure: ([2,0,0], [0,0,0])} \end{array}} \arrow[dl] \arrow[dr] & \\
\boxed{\begin{array}{c} \boxed{Suc(\underline{a+b})}^{\uparrow} = b + \boxed{Suc(\underline{a})}^{\uparrow} \\ \text{Measure: ([1,1,0], [0,0,0])} \end{array}} \arrow[dr] & & \boxed{\begin{array}{c} \boxed{Suc(\underline{a})}^{\uparrow} + b = \boxed{Suc(\underline{b+a})}^{\uparrow} \\ \text{Measure: ([1,1,0], [0,0,0])} \end{array}} \arrow[dl] \\
 & \boxed{\begin{array}{c} \boxed{Suc(\underline{a+b})}^{\uparrow} = \boxed{Suc(\underline{b+a})}^{\uparrow} \\ \text{Measure: ([0,2,0], [0,0,0])} \end{array}} \arrow[d] & \\
 & \boxed{\begin{array}{c} Suc(b+a) = Suc(b+a) \\ \text{Fertilised} \end{array}} &
\end{tikzcd}$$

Figure 7.5: *Redundant branching in the search space of the rippling proof of the commutativity of addition.*

when the goal becomes identical to another node in the search space. A more sophisticated solution is to analyse the rewrites to determine their independence. This avoids the expense of caching and remove unnecessary branching altogether.

7.10 Storing Multiple Annotations with Each Goal

Whether using Smaill and Green's embedding mechanism or our annotated terms, one still has to worry about the direction of wave fronts. Initially, they are always outward but after applying a rule there is a choice of direction for each wave front.

For example, returning to the proof the commutativity of addition, the initial annotated goal is $\boxed{Suc(\underline{a})}^{\uparrow} + \lfloor b \rfloor = \lfloor b \rfloor + \boxed{Suc(\underline{a})}^{\uparrow}$, but after applying the theorem $Suc(x) + y = Suc(x + y)$ from left to right, there are two possible ways the new goal can be annotated:

$$\boxed{Suc(\underline{a + \lfloor b \rfloor})}^{\uparrow} = \lfloor b \rfloor + \boxed{Suc(\underline{a})}^{\uparrow} \tag{7.1}$$

$$\boxed{Suc(\underline{a + \lfloor b \rfloor})}^{\downarrow} = \lfloor b \rfloor + \boxed{Suc(\underline{a})}^{\uparrow} \tag{7.2}$$

In the λ*Clam* system, these annotations are searched over independently examining each rewrite for each possible annotation of the goal. However, we observe it is possible hold all possible annotations with a single copy of the goal and only consider rewriting the goal once. This is an interesting difference between dynamic and static rippling as it can decrease the size of the search space by a possibly exponential amount. This is not possible for static rippling as the annotation is part of the goal. Thus static rippling is forced to search over the all possible annotations of each goal separately.

In order to manage the multitude of annotations, only a single measure needs to be stored. We call this the *threshold* measure. Initially, this is the highest measure in the ordering. After a rule is applied, the new annotations are analysed to yield the highest measure lower than the current threshold. This becomes the new threshold. If no such measure can be found then search backtracks over the rules application. This strategy ensures that all possible rippling solutions are in the search space.

7.11 Redundant Search Over Annotation Directions

While only a single measure is needed to represent all annotations, we observe that the mere existence of multiple annotations for a goal can result in rippling applying redundant proof steps. This happens because rewriting can change outward annotations to inward ones even outside the context of the redex. This can happen because beta reduction after rewriting can change the term's structure.

The effect of reconsidering all possible annotation directions after each rewrite then introduces redundant search even for simple examples. Figure 7.6 shows such an example when trying to prove $a+0=a$ in Peano arithmetic, having arrived at the annotated step case of $\boxed{Suc(\underline{a})}^{\uparrow}+0=\boxed{Suc(\underline{a})}^{\uparrow}$, which is rewritten with the theorem $Suc(X)+Y=Suc(X+Y)$, named add_Suc.

This redundancy in rewriting steps is an important inefficiency for a number of reasons: the search space will be larger, the proofs found will be less readable, the proofs may be more brittle (have unnecessary dependencies), and when being used for program synthesis [64] inefficient programs will be created.

While the number of redundant proof steps is smaller if inward wave fronts are restricted to occurring above a sink, the problem still manifests itself when there are multiple sinks and wave fronts.

Initially it may seem that the extra steps can be avoided by only recomputing annotations

$$\begin{array}{rcl}
\boxed{Suc(\underline{a})}^{\uparrow} + 0 & = & \boxed{Suc(\underline{a})}^{\uparrow} \quad Measure : ([1,1,0],[0,0,0]) \\
 & \downarrow & \text{1. Ripple using add\_Suc from left to right} \\
\boxed{Suc(\underline{a+0})}^{\uparrow} & = & \boxed{Suc(\underline{a})}^{\uparrow} \quad Measure : ([0,2,0],[0,0,0]) \\
 & \downarrow & \text{2. Ripple using add\_Suc from right to left} \\
\boxed{Suc(\underline{a})}^{\downarrow} + 0 & = & \boxed{Suc(\underline{a})}^{\uparrow} \quad Measure : ([0,1,0],[0,0,1]) \\
 & \downarrow & \text{3. Ripple using add\_Suc from left to right} \\
\boxed{Suc(\underline{a+0})}^{\downarrow} & = & \boxed{Suc(\underline{a})}^{\downarrow} \quad Measure : ([0,0,0],[0,2,0]) \\
 & \downarrow & \text{4. Fertilise using the inductive hypothesis.} \\
Suc(a) & = & Suc(a) \\
 & \square & \text{5. Solved by reflexivity}
\end{array}$$

Figure 7.6: *An example rippling proof with redundant steps caused by changing outward annotations to inward ones.*

within the redex of the rewrite, however beta reduction can cause a change in term structure outside of the redex which requires, in general, that all possible annotations are considered.

We propose a similar solution, but which is robust over beta reduction. This involves reexamining the whole embedding term but only changing the direction of annotations if the contents of the wave front have changed. This is facilitated by our representation of annotations using embedding terms where the contents of the wave front are within an upterm. The effect is to stabilise the direction of annotations and thus reduces the search space, removing all cases where annotations are changing outside of the effect of rewriting.

7.12 Identifying when Rippling has Finished

A general problem with rippling using the pair of lists measure concerns how to identify when rippling has finished or is blocked. For example, in Figure 7.6, fertilisation should have been applied after the second ripple step. While fertilisation could be tried eagerly, the problem of identifying a final ripple state is still an important issue if we consider applying proof critics to blocked states. Considering each rippling state in this way results in an very large branching rate.

The problem is that the measure of every fully rippled-out goal can be improved by undoing the rippling out and introducing inward wave fronts. However, we want to consider the most rippled-out state as a candidate for applying proof critics.

Initially, we examined alternative measures that avoid giving inward annotations a lower measure than outward ones. In particular, we considered variations that count the sum of the distances between wave fronts and either the top of the term or a sink. These avoid redundant steps and seem to correspond to our intuition of moving the differences outward or into sinks. Although this makes inward rippling dependent on the distance between a wave front and the top of the term being larger than the distance to a sink, the experiments detailed in chapter 10 show that it works more effictively than list-based measirs. We leave other experiments with rippling measures as future work.

A second solution to identify when rippling is blcoked is to provide a separate check that identifies blocked states. We use this for list-based measures. This approach considers rippling states to be blocked in two different directions, namely outward and inward. A rippling state is blocked outward if, when considering all wave fronts to have an outward direction, there is no rule that will improve the outward measure. This avoids considering states where an outward wave front is change to an inward one at the same location. A state is blocked inward if there is no rule that will improve the ordinary rippling measure, as this naturally moves wave fronts toward sinks. Additionally, we consider a state inward-blocked only if it is not also outward-blocked. This approach applied to the proof in Figure 7.6 gives the state after step 1 as outward-blocked, and the state after step 2 as inward-blocked. A problem with this approach is that if a rippling proof exists which combines outward and inward rippling, then when looking for the inward proof we may undo the outward rippling.

We believe that a promising and more unified approach would be to develop a measure that does not provide a different measures based on the direction of the annotations, but instead considers the distance to possible locations ways in which the goal could be completely rippled, allowing fertilisation. This is left as future work.

7.13 Conclusions

We have introduced the two main approaches to rippling and analysed Smaill and Green's version of dynamic rippling based on embeddings. We then described our notion of embedding terms and an algorithm for finding embeddings. We also provide a notion of depth that gives the same measures for curried and tupled representation of functions. Furthermore, this allows our annotations for rippling to correspond in the first order case with Basin and Walsh's account.

We also consider the search space explored by rippling and describe how the dynamic approach can take advantage of the separation of the measure from the goal to reduce the redundancy in search. This introduces difficulties in determining when rippling is finished. We propose an additional check for considering when rippling is blocked. This provides a version of rippling with a significantly smaller search space.

We have implemented our version of rippling in IsaPlanner for use in the higher order logic of Isabelle. This provides a framework for comparing and experimenting with extensions to rippling, such as the addition of proof critics and the use of modified measures. We have carried out experiments on various varieties of rippling which we discuss further in chapter 10.

A major source of inefficiency in rippling arises from the symmetry in the search space. In particular, this comes from the and-choices in the possible ripple steps being treated as or-choices and thus search considering the permutations of and-choices. Further work includes avoiding this symmetry altogether.

Chapter 8

Generic Equational Reasoning

Equational reasoning is essential for human as well as mechanised proof. Automatic techniques such as simplification and rippling make extensive use of equations, and for interactive theorem proving fine grained control in the application of equations greatly simplifies the proof process. This chapter describes the development of the needed support for equational reasoning in Isabelle. In particular, we describe a generic algorithm, in terms of Isabelle's meta logic, for applying an equation to a goal or previously proved theorem. This involves searching for a unifying subterm of the goal and then constructing an intermediate equation that allows the it to be rewritten. We take special care to support meta-variables and conditional equations. This provides the basic underlying machinery for rewriting techniques such as rippling.

8.1 Introduction

Although Isabelle is a well-established system, its support for fine-grained control of equational reasoning is limited. In this chapter, we describe a generic equational reasoning tactic that provides significant improvements over Isabelle's existing tools. The motivation for developing this functionality further comes from our desire to provide a generic implementation of rippling, which is based on equational rewriting. It is also needed in order to apply weak fertilisation when rippling becomes blocked. We remark that the fine grained control of equational reasoning is also useful for interactive proof and its absence has been a long standing annoyance to new users of Isabelle.

The main part of the tactic is an algorithm to perform substitution that is given an equation as well as unification information that details how to apply it. This is then developed to include searching for the unifiers as well as performing the single step of equational reasoning. The tactic introduces a rule's conditions as new subgoals and allows meta-variables to be introduced and partially instantiated. This provides the user with a tactic that we argue captures the intuitive notion of 'applying an equation'. This improves Isabelle's support for fine-grained control of equational reasoning and has been integrated in the distribution of Isabelle 2005. The tactic is defined in term of Isabelle's meta-logic and this thus generic. It has been implemented for several logics in Isabelle including ZF, HOL and CTT.

Motivation

The motivation for a new tactic to perform equational reasoning comes from the lack of flexibility and applicability of Isabelle's existing tools. Namely, the simplifier and resolution with substitution theorems are both overly restrictive. Isabelle's simplifier is limited in the following ways:

- Applying conditional equations with the simplifier requires that the conditions can be solved immediately: they cannot be solved later by the user or another tactic.
- It uses higher order matching, but for many cases unification is needed.
- It cannot apply equations that introduces meta variables.
- The simplifier applies equations exhaustively. This means that an equation cannot be applied to only one redex and those which cause the simplifier to loop cannot be applied at all.
- The user cannot specify which redex should be used to rewrite a goal.

These limitations are appropriate for simplification based rewriting. However, when a finer level of control is desired, they become a hindrance to the user. Furthermore, in some applications of rippling, such as middle out reasoning [45], appropriate treatment of meta variables during rewriting is essential. This means that the basic machinery implemented in the simplifier is not suitable as the basis for a generic equational reasoning tactic. Extracting this basic machinery from Isabelle's simplifier is also non-trivial as the code has been carefully tuned over many years by many authors and lacks detailed comments.

The other traditional approach to equational reasoning in Isabelle is to use resolution with substitution theorems. This is essentially higher order paramodulation. Although this can be carefully controlled and treats meta variables and conditional rules correctly, it fails for a large class of situations. In particular, it fails when a variable in the equation matches a variable that is bound outside the redex.

For example, given a goal of the form "$P(\lambda n.\ f\ n\ c)$", and an equation "$f\ ?x\ c = g\ ?x\ c$" which, when resolved with the substitution theorem, gives the lemma "$?Q\ (g\ ?x1\ c) \Longrightarrow ?Q\ (f\ ?x1\ c)$", then the lemma's conclusion fails to unify with the goal. This behaviour is too limiting to provide an adequate basis for rippling. For instance, higher order settings such as ordinal arithmetic, variables that are bound outside the redex need to be rewritten (See chapter 10.3 and the work of Dennis and Smaill [35]).

Overview

We now give a brief overview of how our equational reasoning tactic works. The tactic is composed of three stages:

1. A subterm that unifies with the left hand side of a given rule is searched for.
2. A specialised version of the equation is created that includes the context in which the unifying subterm was found.
3. This specialised equation is used to perform the substitution on the desired goal.

Finding an a subterm that unifies with the left hand side of the equation is described in section 8.2, using the information returned by this process to perform a substitution is presented in section 8.3, and combining these to provide an generic equational reasoning tactic is described in section 8.4.

8.2 Finding Unifying Subterms

The first stage of equational reasoning involves searching the target term for a subterm that unifies with the rule's left hand side. Isabelle already provides an implementation of Huet's higher order unification algorithm. However, we cannot simply use the obvious approach of searching through the term based on pattern matching. This is because we need to provide the redex's context with the instantiation information.

For instance, given a target term of the form $P\ (\lambda x.\ Q\ (L\ x)\ y)$ and a rule's left hand side, $L\ ?z$, when we find the unifying subterm in the target, we need to know the redex's context. In particular, we need to be able to construct a term of the form $\lambda l.\ (P\ (\lambda x.\ Q\ (l\ x)\ y)$, where the bound variable l is the location of the unifying left hand side. We call such a term a *context term* as it captures the context of the matching left hand side. This requires knowledge of the rest of the term as well as the variables bound outside the redex. The context term is a modified copy of the target term. The matching left hand side is replaced by variable of higher order type that is bound at the top level. In the above example this is l. The variable takes as parameters the bound variables that occur within the redex.

To do this, we maintain the redex's context using an implementation of Huet's Zippers [47]. This allows us to search through the target term while maintaining the examined subterm's context. We abstract this process into two functions:

- A *matcher* that applies unification or matching to the subterm at the focus of the zipper, returning the zipper's path with the unification result.
- A *searcher* that moves the focus of a zipper around the term applying the matcher to different locations. The searcher function takes a matcher as an argument and results in a lazily evaluated list of results from the matcher.

This separation allows us to parameterise substitution in terms of the search algorithm used to find the redex. This supports specialised search ordering for our equational reasoning tactic. For example, a bottom-to-top, right-to-left searcher can be defined as follows[1]:

```
fun search_up_left matcher initfocusterm =
    let
      fun searchup focusterm =
          case moveup_abs_or_left focusterm of
            SOME higherfocusterm =>
              (matcher focusterm) @ (searchup higherfocusterm)
          | NONE => matcher focusterm
    in
      flattenmap moveup (rev (leafs_of initfocusterm))
    end;
```

By also abstracting out over the matcher we can easily modify the way applicable rules are found. For example, we can use rule-nets to perform quick lookup from a large collections of

[1]We use the eager list operations for clarity. See the IsaPlanner source code for a lazy version of this and other search algorithms.

rules. Similarly, we can easily perform matching instead of unification. We use the zippers path, returned by the matcher, to construct a context term which is provided to the substitution algorithm described in the following section.

8.3 The Substitution Algorithm

We now describe our substitution algorithm, in terms of Isabelle's meta logic, for applying a single step of equational reasoning. We assume that a match or unification result has already been found in the manner described above. This provides instantiation information for meta-variables in the goal and the rule. These instantiations may also introduce new meta-variables and new type variables.

We use two additional pieces of information from the process that searches for a match:

The context term: a term of the form $\lambda l.\ P'\ [l\ y_0\ \cdots\ y_n]$, where l is the location of the matched/unified left hand side of the rule and y_0 to y_n are the variables bound outside the redex. We use the notation $f\ [g]$ to indicate that g is a subterm within f. The allow the term P' to contain variables that have been affected by the unification process. When these variables are replaced by their instantiations, we will write P.

Locally bound variables: the list of the bound variables, y_0 to y_n, which occur in the redex. Although these can be derived from the context term and unification information, it is convenient to treat them separately. For brevity we will refer to these bound variables simply as y.

We allow variables in the rule's left hand side that also occur in a condition to match bound-variables whose binder is outside the redex. Such variables become universally quantified in the introduced subgoal for each condition in which they occur.

We let the (conditional) equation that we will use to rewrite the goal be of the form:

$$C\ ?x\ ?y \Longrightarrow A\ ?x\ ?y\ ?z = B\ ?x\ ?y\ ?z$$

where $?y$ represents variables in this equation that match variables bound outside the redex, $?x$ stands for other variables that also occur within the condition, and $?z$ for variables that do not occur in the condition. Separating these classes of variables allows us to clearly describe how they are dealt with. We do not need to make any assumption regarding the possible introduction

$$\dfrac{\dfrac{\text{Reflexivity of the context term}}{\lambda r.\ P\ [\lambda y.[r\ y]] = \lambda r.\ P\ [\lambda y.[r\ y]]}\,5 \qquad \dfrac{\dfrac{\dfrac{\text{The instantiated conditional equality}}{C\ x'\ y'' \Longrightarrow A\ x'\ y''\ z' = B\ x'\ y''\ z'}\,1 \qquad \dfrac{\bigwedge y.\ (C\ x'\ y)}{C\ x' y''}\,2}{A\ x'\ y''\ z' = B\ x'\ y''\ z'}\,3}{\lambda y.\ (A\ x'\ y\ z') = \lambda y.(B\ x'\ y\ z')}\,4}{\dfrac{(\lambda r.\ P\ [\lambda y.\ [r\ y]])(\lambda y.\ (A\ x'\ y\ z')) = (\lambda r.\ P\ [\lambda y.\ [r\ y]])(\lambda y.\ (B\ x'\ y\ z'))}{P\ [\lambda y.[A\ x'\ y\ z']] = P\ [\lambda y.\ [B\ x'\ y\ z']]}\,7}\,6$$

Figure 8.1: *A proof schema that illustrates the algorithm for creating a specific instance of the initial equation within the context that the redex was found.*

of new meta variables. This is handled by instantiating introduced meta variables to fresh free variables which can then be transformed back into meta variables at the end of the proof.

The goal to be rewritten, is thus of the form:

$$P'\ [\lambda y.\ [A'\ x'\ y\ z']]$$

where A unifies with A', and y expresses the bound variable(s) outside the redex. We let x' and z' indicate the instantiation's for $?x$ and $?z$ in the equation. Although these instantiation may contain additional meta variables, the algorithm places fresh free variables as placeholders for the introduced meta variables and changes them back to meta variables at the end.

Using this notation, we give a schematic natural-deduction style proof in Figure 8.1 which expresses the tactic's behaviour. The rules used are those of Isabelle's logical kernel. Thus, following the LCF style, this tactic is a conservative extension of Isabelle that preserves soundness. The steps in the figure correspond with the steps of the algorithm which are as follows:

1. **(Instantiation)** Initially, the context term and the equation, including its conditions, are instantiated with the result of the match or unification. In these instantiations, variables bound outside the redex but occurring within it are replaced by fresh free variables. Additionally, any meta-variables and type-variables that remain in the conditions are instantiated to fresh free variables. This allows the condition to be assumed[2].

2. **(Assume and Instantiate)** A term is constructed for each instantiated condition and assumed. Schematic and type variables are replaced with temporary free variables as in

[2]Isabelle's kernel disallows a term to be assumed that contains meta-variables

the previous step. Additionally the placeholder variables in the condition that represent bound variables outside the redex (y'') are replaced by the fresh meta-level universally-quantified variables (y).

Each of these terms constructed from a conditional is then assumed. At the end of the proof these become new subgoals. These assumptions are then instantiated to derive a version of the condition with the free variables y'', instead of the meta-level universally-quantified ones (y).

3. (Implication elimination) The conditions are then discharged from the instantiated rule using implication elimination. This leaves a version of the equation without any explicit assumptions, but which depends on a proof of the term constructed in Step 2.

4. (Abstraction) The fresh free variables y'' which represent the location of bound variables in the redex are abstracted over using the abstraction rule. The side condition of the abstraction rule is met as the free variables are fresh by construction.

5. (Reflexivity and Instantiation) The context term is instantiated from the unification/matching result. This term is then used to instantiate the reflexivity rule which creates the trivial reflexive equation of the context term.

6. (Combination) The abstracted equation and the instantiated theorem constructed from the context term are combined using the combination rule. This forms a specific instance of the rule (not yet beta contracted) for the exact context of the goal.

7. (Beta Reduction) The specific rule is beta reduced, producing an equation specifically designed for use with the specific instantiation of the match against the goal.

At this point two further steps can be taken to apply the created special purpose rule to the goal:

8. (Equality Elimination) The goal-specific rule is applied to the instantiated goal using equality elimination.

9. (Implication Introduction) Finally, the assumed conditions are introduced as new subgoals by implication introduction, and any schematic and type variables that were frozen are reintroduced as meta-variables and type-variables respectively.

Remarks

This tactic is unsafe in the sense that rewriting a goal that is provable can result in one that is unprovable. In particular, even if the equation is applicable and the conditions are met in the goal, applying an equation can result in an unprovable subgoal. This happens because of the universal quantification of bound variables within the subgoal generated from a condition. For example, consider the following:

- a function: $appzero\ f = f\ 0$
- a conditional equation: $x = 0 \Longrightarrow gx = hx$
- a goal: $P\ (appzero\ (\lambda x.\ f\ (gx)))$

Using our equational reasoning tactic will result in the two subgoals:

1. $P\ (appzero\ (\lambda x.\ (f\ (hx))))$
2. $\forall x.\ x = 0$

The first of these is the rewritten version of the initial goal and the second comes from the equation's condition. Within the context of the *appzero* the bound x will be reduced to 0, however applying the rule at this point results in the unprovable subgoal $\forall x.x = 0$. The user must first rewrite *appzero* then apply the conditional equation if they wish to avoid the unprovable condition. The lack of safety makes it the users responsibility to apply the tactic at the right time. Our motivation for the increased applicability is to gives the user more choice and thereby facilitate their exploration of proof. Furthermore, for the many cases when the condition's subgoal is provable, it avoid them having to prove that the context is independent.

If the equational reasoning tactic was used to make up an automatic rewriting engine then the safety property might be helpful in order to minimise the search space. A safe version of this tactic which avoids matching variables bound outside the redex to variables that also occur in the equation's conditions is a trivial alteration to the code.

A flexible mechanism that supports extending this tactic to handle cases where bound variables outside the redex can match equation variables occurring in the conditions, in a safe manner, is further work.

$$\dfrac{\dfrac{\text{Initial proof state}}{[\![\cdots;[\![A_1;\cdots;A_n]\!] \Longrightarrow SG_i[lhs];\cdots]\!] \Longrightarrow G} \quad \dfrac{\dfrac{\text{The equation}}{[\![C_1;\cdots;C_m]\!] \Longrightarrow lhs = rhs}}{[\![C'_1;\cdots;C'_m;SG_i[rhs']]\!] \Longrightarrow SG_i[lhs']}}{\begin{array}{c}[\![\cdots;\ [\![A'_1;\cdots;A'_n]\!] \Longrightarrow C'_1;\cdots[\![A'_1;\cdots;A'_n]\!] \Longrightarrow C'_m; \\ [\![A'_1;\cdots;A'_n]\!] \Longrightarrow SG_i[rhs'];\cdots]\!] \Longrightarrow G\end{array}}$$

Figure 8.2: *The proof schema to apply an equation to the conclusion of a subgoal.*

8.4 The Interactive Equational Reasoning Tactic

We now describe the top level of the equational reasoning tactic. This provides an interface for the Isabelle user, and allows us to test it independently of its use within proof planning.

The substitution algorithm we present is in terms of Isabelle's meta logic equality. In order to make use of this, any equations defined in an object level equality must be translated into the meta level. We do this by providing a *logical stub* for our tactic. We define our tactic as a functor in terms of its logic dependent characteristics. In particular, it takes a function that is given an object level equation and produces a meta level one.

Once we have a meta level equation we can employ our substitution algorithm. However this cannot be done directly to the theorem representing the proof state. This is because any conditions of the equation will be introduced as subgoals outside the context of existing assumptions, which can make them impossible to prove. For example, directly using the equation "$Q \Longrightarrow B = C$" to rewrite the subgoal "$Q \Longrightarrow P\ B = P\ C$" would result in the two new subgoals "Q" and "$Q \Longrightarrow P\ C = P\ C$", rather than "$Q \Longrightarrow Q$" and "$Q \Longrightarrow P\ C = P\ C$".

In order to carry the assumptions over correctly, we rewrite a separate temporary theorem and then use resolution to apply it to the one representing the proof state. In particular, we treat applying an equation to the conclusion of a subgoal in a distinct manner from application to one of its assumptions. We illustrate these with proof schema for both scenarios in Figure 8.2 and Figure 8.3 respectively.

In order to apply an equation to a subgoal's conclusion, we construct a theorem for reasoning backward by resolution. In particular, one in which the rewritten conclusion implies the original one. To do this, we pull out the conclusion of the subgoal (SG_i) and construct a trivial theorem from it ($SG_i \Longrightarrow SG_i$). We then rewrite the premise, which results in the theorem $[\![C'_1;\cdots;C'_m;SG_i[rhs']]\!] \Longrightarrow SG_i[lhs']$. This can then be used directly by resolution with the

$$\cfrac{\cfrac{\text{Initial proof state}}{[\![\cdots;[\![A_1;\cdots;A_j[lhs];\cdots;A_n]\!] \Longrightarrow SG_i;\cdots]\!] \Longrightarrow G} \qquad \cfrac{\cfrac{\text{The equation}}{[\![C_1;\cdots;C_m]\!] \Longrightarrow lhs = rhs}}{[\![A_j[lhs'];C'_1;\cdots;C'_m]\!] \Longrightarrow A_j[rhs'];}}{\begin{array}{c}[\![\cdots;[\![A'_1;\cdots;A'_n]\!] \Longrightarrow C'_1;\cdots;[\![A'_1;\cdots;A'_n]\!] \Longrightarrow C'_m;\\ [\![A_1;\cdots;A_j[rhs'];\cdots;A_n]\!] \Longrightarrow SG_i;\cdots]\!] \Longrightarrow G\end{array}}$$

Figure 8.3: *The proof schema to apply an equation to an assumption of a subgoal.*

current subgoals conclusion.

To apply an equation to an assumption, we construct a theorem for reasoning forward from the unmodified assumption to the rewritten one. We first construct a trivial form of the assumption being rewritten ($A_j \Longrightarrow A_j$). We then rewrite this theorem's conclusion using the substitution algorithm described in the last section, which results in a theorem of the form $[\![A_j[lhs'];C'_1;\cdots;C'_m]\!] \Longrightarrow A_j[rhs'];$. This can now be applied to the subgoal's assumptions using Isabelle's resolution tactic.

8.5 Flex-Flex Constraints from Higher Order Unification

One of the complexities of working in higher order domains is the possible introduction of flex-flex constraints in unification. Isabelle's unification algorithm generates these. Although trivial solutions can be found, these can result in unprovable goals.

Isabelle's approach to their management is to store them with each theorem object. This allows them to be solved lazily, but requires unification to be within the logical kernel. Because our substitution algorithm is outside the logical kernel and we apply the instantiations directly to term level objects, we cannot store the flex-flex constraints in the theorem object.

This leaves two possibilities: to either introduce them as explicit subgoals, or to eagerly try to solve them. Our current implementation eagerly solves them, although in future work we plan to experiment with the lazier approach.

8.6 Related Work

The development of suitable tools for reasoning with equations is a problem that every interactive theorem prover encounters. In first order systems, working with equations is simpler as goals can be skolemised and thus no special machinery is needed to work with bound variables. In higher order systems, specialised tools are needed. However, the solutions employed in systems with a fixed logical kernel are dependent on the underlying logical calculus and thus each different logical system requires its own machinery. For instance, in the calculus of inductive constructions, as implemented in the Coq system, applying an equation to a variable bound outside the redex is not possible due to the intentional nature of equality. Thus the notion of applying an equation is necessarily different is such a setting.

In the HOL and HOL-Light systems, which use a similar logic to Isabelle meta-logic, equational reasoning is handled by rewriting machinery that has analogous limitations to Isabelle's. The machinery described in this chapter could easily be implemented for these systems to provide improved and finer level control of equational rewriting.

In other implementations of rippling, such as that in λ*Clam* and *Clam* which did not check their proofs in an underlying logical kernel, the basic equational machinery was trusted. An error within this machinery could then render the system unsound. The motivation for writing the basic equational machinery in terms of the primitive inferences in the logical kernel is that it provides a conservative extension which avoids increasing the trusted code base.

8.7 Conclusions

We have motivated the need for improved equational reasoning support in Isabelle, showing the limits of the exiting tools. We then presented a tactic for Isabelle that provides a flexible and generic approach to reasoning with equations. This is composed of a searching tool based on an implementation of Zippers for Isabelle, and a tactic that supports substitution in an arbitrary part of a theorem using a meta level equation.

We combine the basic tactic to provide a useful notion of a single step of equational reasoning that is roughly analogous to rewriting in traditional approaches to rippling. It carries assumptions correctly into new subgoals and treats meta variables in the expected manner allowing their partial instantiation and introduction. This somewhat technical work provides the foundations for our later implementation of rippling, which requires a rich notion of equational reasoning.

Chapter 9

An Inductive Theorem Prover

In this chapter we discuss the development of an inductive theorem prover for Isabelle using our proof planning framework. The purpose of this development is three-fold: it provides a generic and useful proof tool to users of Isabelle; it clarifies the choices available to an inductive theorem prover; and it provides a means to evaluate our proof planning framework.

9.1 Introduction

This chapter describes the development of an inductive theorem proving technique within our observational approach to proof planning. It uses the representation of proof plans as proof scripts which we described in Chapter 6. The prover is generic in the sense that it can be applied to any logic in Isabelle that supports induction. This generality comes from employing the Isabelle methodology to writing tactics for the definition of our techniques: the different parts of the prover are each defined as functors that explicitly specify their logical dependency. In particular, our inductive prover is a combination of the following otherwise independent and generic tools:

- The selection and application induction schemes (§9.4).
- Simplification machinery (§9.5).
- Rippling machinery (§9.6).
- The caching of proof exploration (§9.7).
- Fertilisation (§9.8).

- Lemma conjecturing and generalisation proof critics (§9.9).
- The caching of proof attempts of lemmas (§9.11).
- Machinery based on embeddings to help avoid non-terminating branches of the search space (§9.11.2).

We first motivate the need for inductive theorem proving and then introduce a simple inductive proof technique. We develop this into our top level of our induction technique and then describe the individual components in further detail. This results in a powerful inductive theorem prover that we compare with λ*Clam* in the following chapter.

Throughout the description of the implemented techniques we take care to examine the role of the observational approach to proof planning. In particular, we note that the `MAP` and `FOLD` functions, which do not exist in other approaches, are useful in the expression of techniques. In particular, they are used to expressing caching of the search spaces and to describe one technique as a modification of another one.

9.2 Motivation

Inductive inference is required for reasoning about repetition. This includes proving properties of mathematical objects, such as the natural numbers, as well as inferring characteristics of datatypes, recursive functions, and many other objects that involve some kind of iteration or looping, such as electronic circuits. Inductive proof thus covers a wide variety of applications. In some general-purpose proof systems, such as ACL2 [57], all properties are essentially proved by induction. Many other systems also use an inductive theorem strategy as the main proof tool. These include INKA [50], Quodlibet [2], Clam [40] and λ*Clam* [32].

Inductive inference is necessary for many proof but results from Gödel and Kreisel show that it introduces an infinite branching point in the search space [42, 62]. Furthermore, it impossible to build a inductive theorem prover that is complete. This results from Godel's first incompleteness theorem which states that for any formal theory that can express arithmetic there will be a formulae that is true but unprovable. Because any non-trivial inductive theory can express arithmetic, the power of any automated inductive theorem prover is limited.

An inductive theorem prover should thus provide a practical approach to inference. Inductive inference is a hard but important problem and in practice, many inductive proofs can be automated. In interactive systems the user chooses and and applies the induction scheme and then tries to prove the remaining subgoals. In this chapter we describe machinery to help the user do this.

9.3 The Top-level Induction Technique

In this section we describe the development of our inductive proof technique. We first introduce a basic technique describing the components and then introduce features until we arrive at the full inductive theorem prover which we experiment with in the next chapter.

9.3.1 A Basic Induction Technique

We introduce a simple inductive theorem proving technique in order to illustrate our language for writing techniques. This simple prover applies induction and simplifies the resulting goals. If the subgoals are not solved then it introduces gap statements to complete the proof. This technique, named `induct_and_simp`, can be expressed as follows:

```
induct_and_simp =
  induction (TRY simplify)
            (fn IHs => (PPLANOP (add_froms IHs))
                THEN (TRY simplify))
```

where the functional `induction` performs induction and uses its first argument, which is a technique, to solve the base cases. In the step cases, the technique `induction` fixes the induction parameters and assume the induction hypothesis. It then applies its second argument, which is a function that is given the induction hypothesises as a list of theorems, to solve the step case. After the base and step case techniques are applied successfully the `induction` technique adds a `qed` to solved subgoals and a `gap` to unsolved subgoals.

For more details of the `induction` technique see section 9.4. The technique `simplify` performs an exploratory step using simplification and is discussed further in section 9.5. We wrap the simplification technique in a `TRY` statement so that if the goal cannot be simplified a proof planing continues and a gap is inserted. This is needed because if simplification is not applicable, the empty list will be returned and proof planning will backtrack.

Recall from chapter 3 that the `PPLANOP` functional wraps-up a modification to the proof plan as a reasoning technique. We used it on the `add_froms` function which inserts a `from` statement in the Isar proof. This allow the induction hypothesis to be used in the following simplification.

9.3.2 Adding Rippling to the Induction Technique

We now show how the step case use of simplification can be replaced with rippling and fertilisation. We do this by writing a step-case technique that performs rippling and then tries strong fertilisation. If this is not applicable it tries weak fertilisation followed by simplification. This is expressed as:

```
ripple_and_fertilise IHs =
  ((CACHE (rippling IHs))
   THEN (strong_fertilse
         ORELSE ((TRY weak_fertilise)
                 THEN (TRY simplify))))
```

The `rippling` technique is described further in section 9.6. For now it suffices to observe that it takes the list of induction hypothesises as an argument and uses them as the skeleton(s) for rippling. The `CACHE` functional, which we detail further in section 9.7, caches the exploration steps during rippling. This helps avoid symmetries in the search space. When rippling finishes, we attempt to fertilise the goal. We make use of the `ORELSE` element of our technique language which forces the first technique to be tried before considering the second. This allows us to first try strong fertilisation. If it is not applicable the `ORELSE` functional will try weak fertilisation and then simplification. We apply simplification after weak fertilisation leaves the goal in a simpler form and sometimes solves it directly.

We can now create a top level induction and rippling technique using the new step case technique as follows:

```
induct_and_rippling =
  induction (TRY simplify)
           ripple_and_fertilise
```

9.3.3 Adding Lemma Conjecturing to the Induction Technique

We now introduce the conjecturing of lemmas into our technique. The idea is that after simplification or rippling with weak fertilisation fails to solve the goal, we will conjecture a lemma using common subterm generalisation to solve it. We do this using a function, `conj_critic`, that takes two techniques as arguments. It uses the first to solve the conjectured lemma. The second technique is the one that it tries to patch the failure of. It will apply it as normal until the proof attempt ends without being able to solve the remaining goal. At this point, the `conj_critic` function suggests a conjecture, starts a new proof attempt, and upon successfully proving the conjecture, uses the new lemma to solve the goal. For efficiency, the result of proof attempts are cached as described in section 9.11

To use the `conj_critic` technique, we simply wrap it around the simplification and step case techniques as follows:

```
induct_ripple_conj =
  induction (TRY (conj_critic induct_ripple_conj simplify))
            (conj_critic induct_ripple_conj ripple_and_fertilise)
```

This combination of techniques is shown graphically in Figure 9.1. We note that, within the `ripple_and_fertilise` technique, simplifying the remaining subgoals not solved by fertilisation also helps avoid conjecturing many variations of the same lemmas. In effect, it normalises of the remaining subgoals and thus avoids conjectures that are equivalent modulo simplification.

We now present an example to illustrate the application of this technique. Consider proving the theorem $(x^y)^z = x^{(y \cdot z)}$ in ordinal arithmetic. Details of our formalisation are given in chapter 10 and the full theory file is in appendix B. Our naive selection of induction schemes will try using the transfinite induction rule for each variable, eventually selecting z. The base case as well as the limit step case are solved by simplification. The successor step case ripples to the subgoal to $(x^{(y \cdot z)}) \cdot x^y = x^{(y \cdot z + y)}$. From this a lemma $x^{v_0} \cdot x^y = x^{(v_0 + y)}$ is conjectured with the subterm $(y \cdot z)$ generalised to a new variable. The proof of this proceeds in a similar fashion and succeeds automatically. The lemma is then used to solve the pending goal. This proof is shown graphically in Figure 9.2. The Isar script generated for this example is presented in chapter 10, page 190.

In the following section we describe in more detail each of the parts of our inductive prover and discuss issues they raise.

9.4 The Selection and Application Induction Schemes

The selection of induction schemes is an essential problem to be addressed by inductive provers. Given a specific induction rule, it is often possible to apply it in several ways. Typically, we want to examine the goal and select both the induction scheme and the way it to apply it simultaneously.

Isabelle already has some basic interactive machinery to select and apply induction schemes. In this section we describe an extension to it (that universally quantifies non-induction variables) and present two techniques to automate the selection of variables and induction schemes.

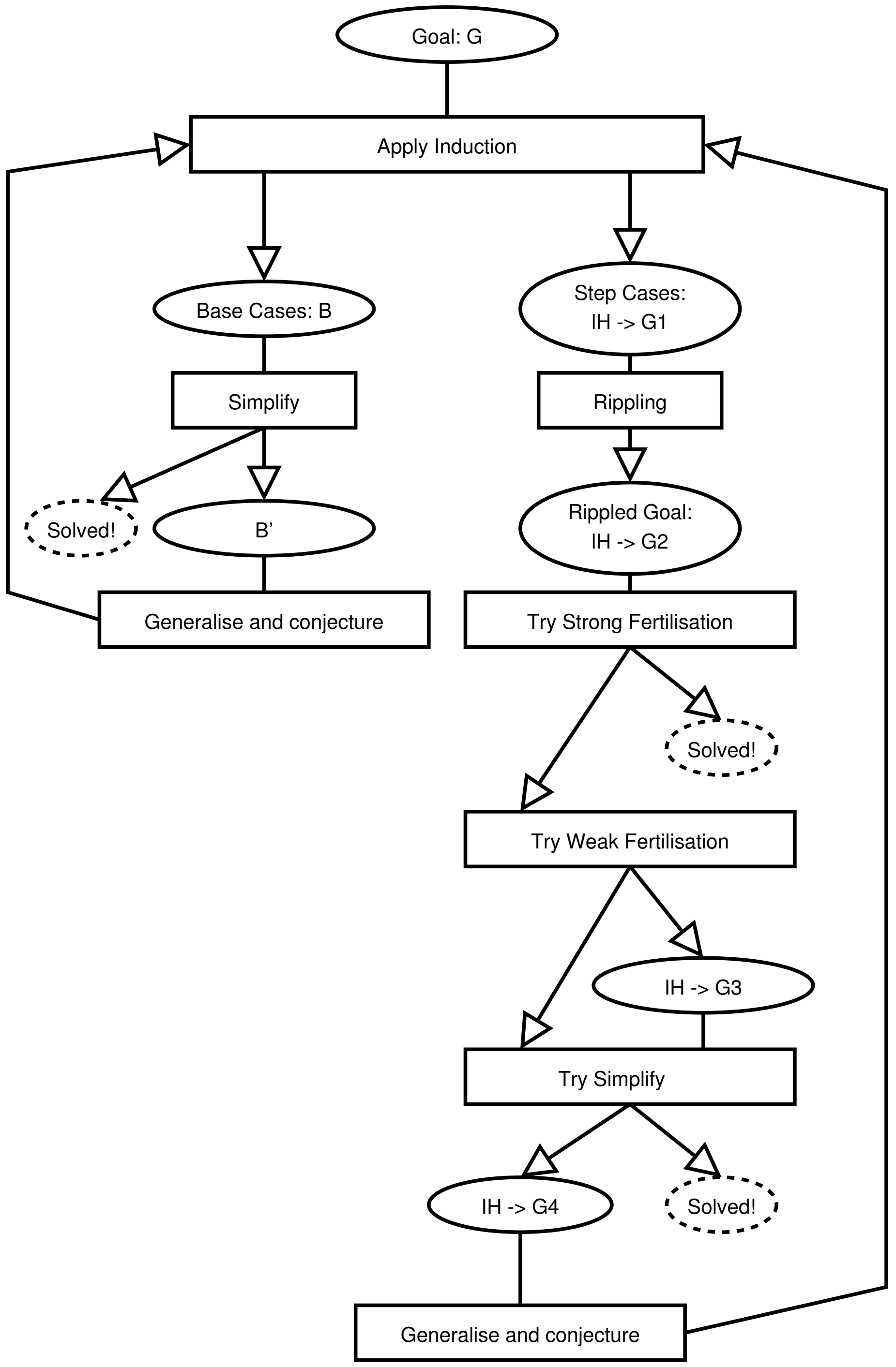


Figure 9.1: *An illustration of the technique combining induction, rippling and the generalisation and conjecturing of lemmas.*

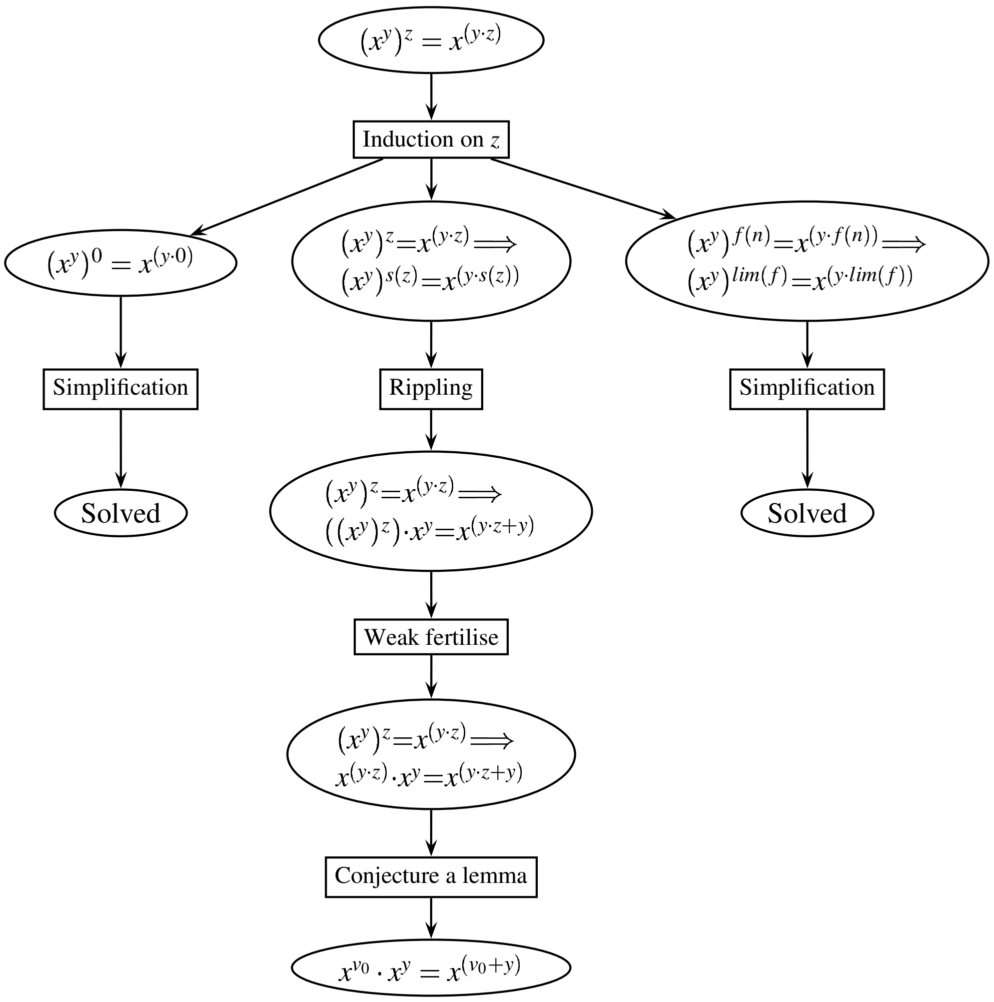

$(x^y)^z = x^{(y\cdot z)}$
Induction on z
$(x^y)^0 = x^{(y\cdot 0)}$
$(x^y)^z=x^{(y\cdot z)} \Longrightarrow (x^y)^{s(z)}=x^{(y\cdot s(z))}$
$(x^y)^{f(n)}=x^{(y\cdot f(n))} \Longrightarrow (x^y)^{lim(f)}=x^{(y\cdot lim(f))}$
Simplification
Rippling
Simplification
Solved
$(x^y)^z=x^{(y\cdot z)} \Longrightarrow ((x^y)^z)\cdot x^y=x^{(y\cdot z+y)}$
Solved
Weak fertilise
$(x^y)^z=x^{(y\cdot z)} \Longrightarrow x^{(y\cdot z)}\cdot x^y=x^{(y\cdot z+y)}$
Conjecture a lemma
$x^{v_0}\cdot x^y = x^{(v_0+y)}$

Figure 9.2: Proof planning $(x^y)^z = x^{(y\cdot z)}$ to the point where the conjecture $x^{v_0} \cdot x^y = x^{(v_0+y)}$ is speculated. This lemma will then be proof planned separately.

One is a naive approach that tries an induction scheme associated with the type of each variable in the goal. This technique can be applied in as many ways as there are variables of inductive type. The other approach is based on *ripple analysis*, which takes advantage of our rippling machinery to look ahead into the proof and suggest a single induction rule. Finally, we consider a reasoning technique that constructs part of a proof plan for inductive proofs.

9.4.1 Applying Induction Schemes in Isabelle

In Isabelle, induction schemes are theorems. For example, the theorem expressing the Peano arithmetic induction is $[\![?P\ 0;\ \bigwedge n.\ ?P\ n \Longrightarrow ?P\ (Suc\ n)]\!] \Longrightarrow ?P\ ?x$. Isabelle's datatype package automatically derives such induction schemes from the definitions of recursive datatypes. The user can also derive additional induction schemes manually.

To apply these induction rules, Isabelle provides some basic machinery in the form of an induction tactic. This takes as parameters the variables on which induction is to be performed and optionally the name of a specific induction scheme. If no induction scheme is given, it uses the induction scheme associated with the variables inductive datatype as derived by the datatype package.

The induction tactic is essentially a guided application of resolution. For example, the above one-step Peano arithmetic induction rule can be applied directly using resolution to the goal "$a+b=b+a$". However, there are several possible instantiations of $?P$, most of which will not yield a proof. For instance if "$?P \;=\; \lambda x.\, x=b+a$" then the resolution would leave the new subgoals "$0=b+a$" and "$\bigwedge n.\, n=b+a \Longrightarrow \mathit{Suc}\; n=b+a$". The first is clearly false and thus the proof cannot be finished. Isabelle's induction tactic allows the user to select a variable on which to apply induction.[1] It partially instantiates the induction rule allowing resolution to then be used.

While Isabelle's induction tactic is suitable for interactive use, for application within our automated theorem prover, we require machinery that decides how to apply the induction scheme automatically. Furthermore, Isabelle's induction tactic provides a notion of induction that is too weak. In particular, the variables on which induction is not applied are not universally quantified in the induction hypothesis. This means that they cannot be used as sinks during rippling. This is needed for many proofs about tail recursive functions. For instance, proving that the 'quick' tail recursive version of reverse, *qrev*, has the same effect on lists as the traditional version, *rev*, is illustrated in Figure 9.3. For interactive proof, the user can manually modify the conjecture to universally quantify non-inducted variables. For the automatic inductive theorem proving, we have developed machinery to do this as well as select appropriate induction schemes. We describe this in the following sections.

9.4.2 An Induction Tactic that Quantifies Non-Induction Variables

We now describe our tactic that automatically generalises the goal before induction is applied in order to make all non-induction variables universally quantified. Like Isabelle's existing tactic, it takes as parameters the variable on which to perform induction and an optional induction

[1]Isabelle's induction tactic also provides a special treatment of meta level assumptions, although this is redundant when using Isar style of proofs where they are stated separately. The interested reader should consult the Isabelle/HOL tutorial for further details [78].

$$\forall y.\ (rev\ (\boxed{h :: \underline{x}}^{\uparrow}))@\lfloor y \rfloor = qrev\ (\boxed{h :: \underline{x}}^{\uparrow})\ \lfloor y \rfloor$$

$$\Big\downarrow \quad \text{forall introduction, we take a fixed } y$$

$$(rev\ (\boxed{h :: \underline{x}}^{\uparrow}))@\lfloor y \rfloor = qrev\ (\boxed{h :: \underline{x}}^{\uparrow})\ \lfloor y \rfloor$$

$$\Big\downarrow \quad \text{using rev\_def}$$

$$(\boxed{\underline{(rev\ x)}@[h]}^{\uparrow})@\lfloor y \rfloor = qrev\ (\boxed{h :: \underline{x}}^{\uparrow})\ \lfloor y \rfloor$$

$$\Big\downarrow \quad \text{using qrev\_def}$$

$$(\boxed{\underline{(rev\ x)}@[h]}^{\uparrow})@\lfloor y \rfloor = qrev\ (x)\ (\lfloor h :: y \rfloor)$$

$$\Big\downarrow \quad \text{using assoc\_append}$$

$$(rev\ x)@\lfloor([h]@y)\rfloor = qrev\ (x)\ (\lfloor h :: y \rfloor)$$

$$\Big\downarrow \quad \text{strong fertilisation with the induction hypothesis.}$$

$$\square$$

Figure 9.3: *The step case in the proof of "$\forall x\ y.\ (rev\ x)@y = qrev\ x\ y$" where induction on x yields the induction hypothesis "$\forall y.\ (rev\ x)@y = qrev\ x\ y$" in which x is fixed. This example proof shows the need to universally quantify at the object level the non-induction variables in order to have sinks to push wave front into. Treating non-induction variables as arbitrary but fixed values results in the weaker induction hypothesis "$(rev\ x)@y = qrev\ x\ y$" with fixed x and y. This is not sufficient to prove the above goal as the final strong fertilisation step requires y to be universally quantified. The rippling proof also uses this to guide wave fronts into sinks.*

scheme. Our tactic simply performs some initial steps before employing Isabelle's existing induction tactic. In particular, it examines the free variables and fixed parameters and transforms the goal into one where all free parameters on which induction is not applied are universally quantified. This involves the three steps:

1. The first step for this tactic is to calculate all non-induction variables ($\bar{y}$). This is a simple operation done by comparing the variables in the goal with those on which induction is being applied.

2. We then use these variable names to create a trivial implication theorem based on the subgoal where the non-induction variables are universally quantified. In this theorem's

$$\dfrac{\dfrac{\text{Initial proof state}}{[\![\cdots;G_s[\bar{y}];\cdots]\!] \Longrightarrow G_c} \qquad \dfrac{\dfrac{\text{Trivial}}{(\bigwedge \bar{y}.\ G_s[\bar{y}]) \Longrightarrow (\bigwedge \bar{y}.\ G_s[\bar{y}])}}{(\bigwedge \bar{y}.\ G_s[\bar{y}]) \Longrightarrow G_s[\bar{y}]}\ \textit{Instantiation}}{[\![\cdots;\bigwedge \bar{y}.\ G_s[\bar{y}];\cdots]\!] \Longrightarrow G_c}\ \textit{Implication elimination}$$

Figure 9.4: *The proof schema to generalise a goal for induction by placing universal quantifiers over all non-induction variables ($\bar{y}$). The term G_c is the main goal being proved and G_s is the subgoal being considered.*

conclusion, the universally quantified variables are then instantiated to their corresponding free variables from the original goal.

3. This is then used to solve the subgoal it was constructed from which leaves as a new subgoal a version of the original goal where all non-induction variables are universally quantified.

The tactic that performs these steps is illustrated in Figure 9.4.

Having carried out these steps, we then employ Isabelle's existing induction tactic. Because non-induction variables are now universally quantified in the induction hypothesis, they can be treated as sinks for rippling. For interactive use, this is also convenient as it saves the user from having to modify the initial conjecture.

To use the induction tactic in proof planning, we provide an interpretable description of induction following the approach described in Chapter 6. The data that this interpretable Isar method holds is simply the variable(s) to perform induction on and the optional scheme to employ.

9.4.3 Naive Selection of Induction Schemes

A simple approach to the selection and application of induction schemes is to invoke the tactic described in the previous section on every free variable in the goal. This uses the induction scheme associated with each variable of inductive type. This results in a possible application for each variable on which induction can be performed. This naive approach is surprisingly effective as we reported in [37, 38]. More detailed results in comparison with λ*Clam* are presented in Chapter 10.

9.4.4 Ripple Analysis

While our naive approach works well for many cases, many proofs require the use of induction schemes more complex than those associated with the inductive types. For example, the proof shown in Figure 9.5 requires a two step induction rule for Peano Arithmetic. A more sophisticated approach to the selection of induction schemes is called *ripple analysis* and was suggested by Bundy et al [17, 18]. The basic idea is to look ahead into the rippling proof of each possible induction scheme and select the one that is most promising. This is measured by the number of wave fronts that can be moved in one step of rippling.

Ripple analysis has many extensions, most notably it was used by Gow in the Dynamis system to automatically derive suitable induction schemes during the proof attempt [44]. A simpler approach is to select an induction scheme from a predefined set. This was used in the PhD project of Kraan [60, 61] and implemented in the *Periwinkle* system. Implementing ripple analysis in IsaPlanner has been left as further work. However, we note that given the well-developed state of the rippling machinery this should be fairly strait foward.

One question which arises from the use of ripple analysis concerns the treatment of the less favourable induction schemes. In particular, should their application(s) be pruned out of the search space or just examined later. In the Clam system, the other branches of the search space were pruned and induction revision critics were employed to later change the chosen induction scheme. However, it is unclear how well this works in practice. Performing experiments to answer such questions is left as further work.

9.4.5 Other Approaches to the Selection of Induction Schemes

Perhaps the best known approach to the selection and construction of customised induction rules is *recursion analysis* which is due to Boyer and Moore and has been implemented in Nqthm [11,12] and more recently in ACL2 [57]. This approach involves identifying recursively defined functions in the conjecture and combining their corresponding induction rules. They have developed techniques for merging the induction schemes into a single rule that subsumes the ones its was constructed from. The basic heuristic underlying recursion analysis is to choose an induction hypothesis that contains the same destructor functions as the recursive definitions. This increases the chances of being able to apply the induction hypothesis after unfolding the definitions.

A variation of recursion analysis that uses a different technique for merging induction rules has been proposed by Walther [100]. As well as approaches based on a finite set of explicit induction rules, there are also approaches to the dynamic construction of induction rules during proof search. Of most relevance to our work is that of Gow [44] mentioned earlier. His approach attempts to derive the induction scheme during rippling using the annotations to aid both rippling and the derivation of the induction scheme. This allows a specific induction rule to be derived for the conjecture at hand. The interested reader should examine his work for further details. Another promising approach to the construction of induction rules during proof search has been proposed by Brotherston [13]. This performs proof search with case analysis and observes the proof process to identify cycles. However, the strength of the cyclic system he presents has not yet been proved to be equivalent to the use of an explicit induction schemes.

9.4.6 The Proof Planning Induction Technique

We now describe how the tactic machinery described above can be used to provide a proof planning technique that constructs part of an Isar proof script for an inductive proof attempt.

Our induction technique does not require any special purpose contextual information to be held by the proof planner. However, it does make use of the induction rules held by the datatype package. This information is from the current theory information.

Induction schemes can be applied in both the procedural and declarative styles of writing Isar proof scripts. Our technique that implements induction for the procedural *apply*-style is trivial as the proof plan simply needs to be augmented with a single line:

```
apply_induct rule_select rst = PPLANOP (add_apply (rule_select rst)) rst
```

This technique takes as a parameter the function that selects which induction scheme to apply. The `rule_select` function returns an interpretable method that applies the induction scheme. For instance, this can be naive selection or ripple analysis, described earlier. The proof plan modification function `add_apply` simply adds the Isar command "`apply` *M*" where *M* is our Isar method to apply the induction scheme. Recall from chapter 3, that the `PPLANOP` takes a function on proof plans and produces a reasoning technique that modifies the proof plan.

When employing the declarative Isar style of proof, we adopt a different approach to proof plan construction. The Isar style uses the `proof` construct to perform a backward step. This also changes from Isar's *proof* mode to *state* mode. We can then construct the appropriate context and introduce gaps:

```
induct_proof rule_select r rst =
  (add_proof (rule_select rst))
```

```
lemma "Even(x + y) ∧ Even (y + z) ⟶ Even(x + z)"
  proof (induct "x" rule: two_step_induct)
  show "Even (0 + y) ∧ Even (y + z) ⟶ Even (0 + z)" gap
next
  show "Even (Suc 0 + y) ∧ Even (y + z) ⟶ Even (Suc 0 + z)" gap
next
  fix n
  assume "Even (n + y) ∧ Even (y + z) ⟶ Even (n + z)"
  show "Even (Suc (Suc n) + y) ∧ Even (y + z)
    ⟶ Even (Suc (Suc n) + z)" gap
qed
```

Figure 9.5: *An Isar style proof script for the transitivity of even with respect to addition. This has gaps for the subgoals after induction is applied. The proof script uses the induction scheme "*$[\![?P\ 0;\ ?P\ (Suc\ 0);\ \bigwedge n.\ ?P\ n \Longrightarrow ?P\ (Suc(Suc\ n))]\!] \Longrightarrow ?P\ ?x$*" to perform induction as a backward step using the* `proof` *command with the induction tactic that instantiates "?x" in the induction rule to the free variable "x".*

```
|> ((REPEAT_UNTIL solved (incontext r))
    THEN (add_qed NONE))
```

Again we take as a parameter a function `rule_select` which selects a method to perform induction. The reasoning technique `incontext` creates the Isar script that builds up the context by fixing parameters and assuming assumptions. It takes as an argument an a reasoning technique which it uses to prove the remaining goals. The function `solved` is true when all goals in a block have been solved. As introduced in chapter 3, `REPEAT_UNTIL` applies the technique until the condition function (in this case `solved`) is true. The technique `add_qed` simply adds the Isar `qed` command to the proof script which ends the proof block. For example, Figure 9.5 shows the Isar script generated when proving "$Even(x+y) \wedge Even(y+z) \Longrightarrow Even(x+z)$" using a simple technique that inserts gaps.

9.5 Simplification

We make use of Isabelle's powerful existing simplification machinery by wrapping-up the tactic as an IsaPlanner technique. We use the `OFMETH` function described in Chapter 6 to support exploratory backward proof steps. This involves providing an interpretable method to capture the simplification tactic and its possible arguments. We also provide an interpretable attribute for locally adding theorems to the simplification set.

The simplifier supports conditional rules and uses higher order matching to apply them. It is written following the LCF style and thus is a conservative extension of Isabelle. The user is responsible for configuring the simplifier. This involves providing it with theorems to use as simplification rules. Some rules, such as recursive definitions, are given to the simplifier automatically. However, the machinery does not check for possible non-termination in the set of rules: this is the user's responsibility.

The simplifier also incorporates decision procedures and can be extended with new ones. This makes it a very powerful and useful tool. The interested reader should consult the Isabelle tutorial [78] for further details. We remark that the ability to make use of tactics implemented in Isabelle, such as the simplifier, comes from the interleaving of the proof plan's execution with its construction. This is a salient benefit of our approach-to and representation-of proof plans.

As well as setting up the set of simplification rules, the user can specify how assumptions are dealt with. In particular, assumptions can be applied to the conclusion as well as each other. As mentioned in Chapter 6, the normal application of simplification is not stable over chained results. In particular, if a goal is reduced to a subgoal by simplification, a proof of the subgoal cannot safely be chained to solve the goal by simplification, this is because it may use the chained result as an assumption to rewrite the main goal, which can inadvertently cause the simplifier to behave differently. For safe usage with chained results, simplification must be applied with the explicit option to stop it from using assumptions to rewrite the goal.

Simplification is an essential part of our proof machinery and, as shown in the Chapter 10, can be combined with generalisation and lemma conjecturing to provide a powerful proof technique without requiring any rippling machinery.

9.6 Rippling in IsaPlanner

The guiding motivation for our development of rippling is to provide a generic implementation for Isabelle that supports experimentation. In Chapter 7, we introduced a general formalism that is able to express many varieties of rippling. We also noted various modifications to rippling and choices open to an implementation. In this section, we describe the framework in IsaPlanner that allows us to implement these choices and experimentally compare them.

Following the approach we outlined in Chapter 3, we separate rippling into interpretable information that can be used by proof critics and the functional component that describes the behaviour of the technique. The idea behind dynamic rippling is independent of underlying logic. The general strategy is to rewrite the goal in a measure decreasing manner until it is blocked. The main work that must be done is to manage the annotations and measures. This is done by the contextual information for rippling which also functions as the interface available to proof critics.

The rules to be used by rippling are held by the theory data and can thus be defined in a general, or domain specific manner. We provide Isar attributes to add theorems to the theorem's set of wave rules. Following our suggestion in Chapter 7, by default, we filter out any which are frequently applicable but never helpful to rippling, such as the rewrite $x \Rightarrow x+0$. To allow experimentation with wave rule sets we also allow the user to force the addition of a theorem to this set, ignoring the filter, as well as remove them. We use Isabelle's existing indexed term nets to provide machinery that quickly select rules that match or unify with a given goal.

During rippling, equations are applied using our equational reasoning tactic. We note that the separation of the search function from the actual rewriting in this tactic allows rewriting arbitrary redexes without having to re-perform the search. We can simply provide the matching information directly to the tactic.

Identifying when Rippling is Blocked

We do not enforce the traditional view of rippling as being blocked when no further rules apply. This is because, as noted earlier, the traditional measure would result in pushing all wave fronts inwards even if they were previously rippled out maximally. Thus rippling out would rarely be used to suggest a lemma and as a result many proofs cannot be found.

It might be thought that eagerly checking for fertilisation would easily maintain the traditional story. However, this increases the size of the search space significantly. Furthermore, when rippling-out is blocked but fertilisation is not possible, then rippling-in will continue, despite it usually being desirable to try conjecturing a lemma.

Another approach is to perform two kinds of rippling, one for rippling-outward and another for rippling outward then inward. However this results in the rippling-out search space being explored twice.

To allow experimentation with the various approaches, we give the contextual information for rippling the power to identify when rippling is blocked. An advantage of this is that it allows us to parameterise the rippling reasoning technique in terms of only the contextual information. This makes it easy to experiment with variations to rippling: we only need to modify aspects of the contextual information. The code for the rippling technique remains unchanged.

Contextual Information for Rippling

We further modularise the rippling contextual information to allow different treatments of the skeleton and various rippling measures. The contextual information for rippling holds a list of skeletons. For each skeleton there is list of embeddings and for each embedding there is a list of possible measures. Additionally, in order to identify when rippling is blocked, the contextual information holds a list of applicable methods. These represent the possible ways to apply theorems. Each time a step of rippling is performed the next applicable measure-decreasing rules are computed.

To implement this modularised version of rippling, we provide a signature for working with measures of an embedding. This allows many of the choices in an implementation of rippling to be localised to that module. To manage the possible embeddings of a skeleton we define a functor in terms of the measure module. This allows choices such as the grouping of multiple embeddings to be made largely independent of the measure used. The hierarchy of modules is shown in Figure 9.6. This figure also shows the choices currently available within each module in our implementation of rippling.

The skeleton/embedding management holds the current annotations associated with each skeleton. It provides an update function that is given a new goal and computes the valid embeddings and corresponding measures. Because each skeleton can potentially be used in several ways to annotate a goal, the update function results in a list of possible new associations. This corresponds to a branch in the rippling search space based on different possible annotations. This allows us to implement both the approach to rippling that holds all embeddings in each reasoning state, as well as the one that searches over different annotations, as is done in the traditional implementations of static rippling.

The contextual information manages the possibility of multiple skeletons. Each skeleton can annotate a goal differently, and thus when there are several skeletons, to model the traditional approach to rippling, we consider each combination of ways of annotating skeletons. We also implement a version that uses our proposed approach which holds all annotations for a skeleton at once. In this case the update function results in a singleton list when some embedding is possible and in the empty list otherwise.

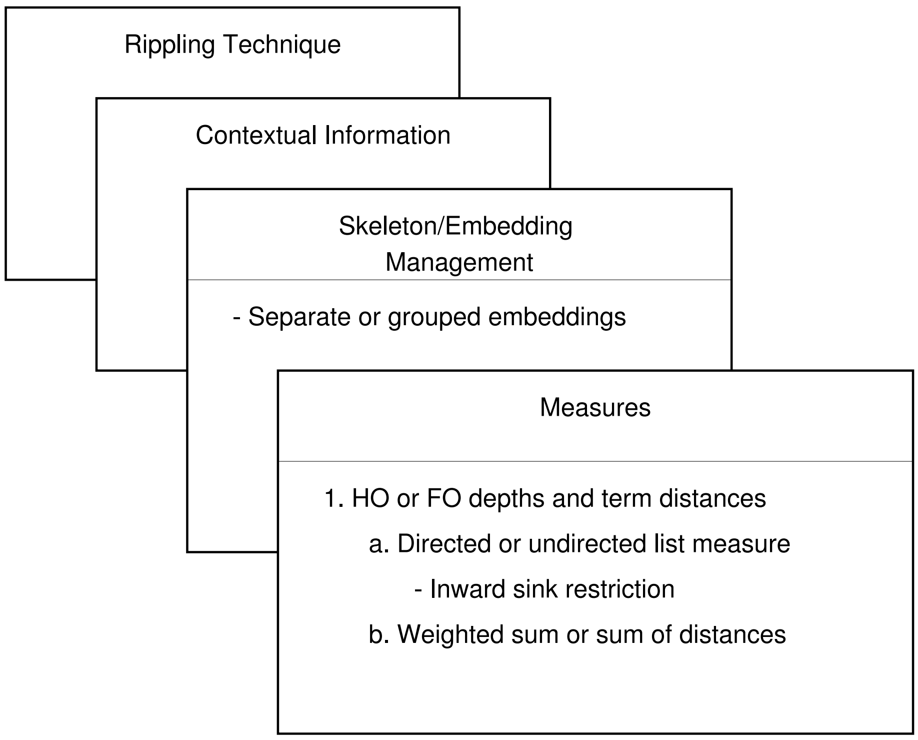

Rippling Technique
Contextual Information
Skeleton/Embedding Management
- Separate or grouped embeddings
Measures
1. HO or FO depths and term distances
a. Directed or undirected list measure
- Inward sink restriction
b. Weighted sum or sum of distances

Figure 9.6: *An illustration of the data hierarchy used by the rippling technique. This shows the available choices to an implementation. See Chapter7 for further details.*

The Rippling Technique

Our version of rippling is written as a reasoning technique which uses the rippling contextual information. The technique is parameterised on a list of theorems which are used as the skeletons. It then has three stages:

1. Setup: Using the given set of skeletons rippling creates an initial list of embeddings and measures. These correspond to the possible rippling annotations.

2. Ripple Steps: Theorems in the wave rule set are used to transform the goal. Note that the order in which the rules are applied is irrelevant as the rewriting process is guided by the rippling measure. After each successful rule application, a new set of annotations is created. When several rules can be applied, or even a rule can be applied in different ways, this creates a branch in the search space.

3. Blocked: The contextual information for rippling identifies each state which it considers blocked. These states have no continuation and can thus be viewed as the states resulting from rippling.

We can express this technique to perform rippling as follows:

```
ripple_step rst = flat (map (update_ripple_states rst) (ripple_methods_of rst))

rippling =
  setup_rippling THEN
  (REPEAT_UNTIL blocked ripple_step)
```

where the function `ripple_methods_of` returns the applicable Isar methods that will be used for rippling and `update_ripple_states` takes a reasoning state and one of these methods which it uses to rewrite the top goal. The function `update_ripple_states` results in the possible valid reasoning states using the contextual information's update function. These correspond to the possible ways of annotating the result of applying the method. Conceptually the `ripple_step` function can be summarised as performing a single step of rewriting and results in new states which correspond to the possible measure decreasing annotations. Rippling is then defined as the repeated application of these ripple steps. This repetition stops when a reasoning state's contextual information considers rippling to be blocked. This is identified by the `blocked` function.

9.7 Caching Exploration

One of the most significant factors that make modern first order provers efficient is their use of caching intermediate results. It is thus surprising to find this rarely mentioned or implemented in the domain of inductive theorem proving. Furthermore, in practice we find that caching proof attempts makes a significant improvement on the speed of the prover.

In particular, we can provide a generic notion for caching the states explored during exploration. We do this by providing a technique function `cache` that takes as an argument a technique that performs some exploration and results in a technique that performs the same exploration, but prunes states that result in the same open goals. This is useful for avoiding symmetries in the search space and also helps avoid some non-terminating branches of the search space.

This is implemented using the map function over the given technique with an extra parameter that is a reference to a term net that holds the cached proof states:

```
CACHE r = MAP (cache_update (ref empty_cache)) r
```

where the `empty_cache` is an empty term net and `ref` creates a reference to it. To ensure that we compare goals in a uniform manner all open goals are composed together using a canonical ordering on terms. Using a reference variable for the cache allows it to be shared across branches in the search space. We then map the function `cache_update` which adds the proof state to the cache. If the goal is already in the cache then the reasoning state is modified to have no continuation. This effectively prunes the search space. Because caching simply involves applying the cache functional, it is easy to integrate with existing techniques such as rippling.

This operation only preserves relative completeness of the technique it is applied to if that technique behaves in the same way on the same proof state in different or-branches. This is true for repeated applications of simple techniques. However, for more complex ones like rippling, we also need to account for the non-logical information, such as the measure. In particular, to preserve relative completeness we need make sure that search is continued from the reasoning state with the subsuming constraint. For rippling that is the one with the worst measure.

We note that this caching technique shows a practical application of the `MAP` function that is unique to our approach to proof planning. In other systems, such as λ*Clam* and Omega, implementing caching such as this would require modification to the proof planner and even then it cannot easily be associated to the application of a specific method. This shows that having a rich language for combining techniques offers flexibility in writing them that can lead to simple implementations of otherwise complex behaviour.

9.8 Fertilisation

As described in Chapter 7, the term *fertilisation* describes the application of the inductive hypothesis. When the goal is an instance of the induction hypothesis it can be solved directly by resolution using Isabelle/Isar's `rule` method. This is called *strong fertilisation*. When the induction hypothesis is an equation and strong fertilisation is not applicable, it is often still possible to perform substitution using the hypothesis. This is called *weak fertilisation*. For example, in a proof with induction hypothesis $a+b = b+a$ we arrive at the subgoal $Suc(a+b) = b+Suc\ a$. At this point, we can use the hypothesis to rewrite this to $Suc\ (b+a) = b+Suc\ a$. As has been remarked by Bundy et al. [17], as well as by many others, such steps often introduce common subterms that can be generalised to make fruitful conjectures.

We remark that even when no such common subterm is generalised, it is useful to conjecture the remaining subgoal as a lemma. We can see why this is the case by considering the heuristic that motivates rippling. Namely to reduce our goal to an instance of the induction hypothesis. In particular, consider the schematic example of attempting to prove $l\ a = r\ a$. This gives a step case of the form $l\ (\boxed{c\ \underline{a}}^{\uparrow}) = r\ (\boxed{c\ \underline{a}}^{\uparrow})$. After rippling, weak fertilisation can be possible one of the following ways, where we use l' and l'' to indicate subterms of l, and similarly r' and r'' for subterms of r:

1. Substitute in the left hand side of the goal:
$$\begin{array}{l}l'(\boxed{f\ (\underline{l\ a})}^{\uparrow}) = r'\ (\boxed{g\ (\underline{r''\ a})}^{\uparrow}) \\ \qquad\qquad \Big\downarrow \text{ using the hypothesis from left to right} \\ l'(\boxed{f\ (\underline{r\ a})}^{\uparrow}) = r'\ (\boxed{g\ (\underline{r''\ a})}^{\uparrow})\end{array}$$

2. Substitute in the left hand side of the goal:
$$\begin{array}{l}l'(\boxed{f\ (\underline{r\ a})}^{\uparrow}) = r'\ (\boxed{g\ (\underline{r''\ a})}^{\uparrow}) \\ \qquad\qquad \Big\downarrow \text{ using the hypothesis from right to left} \\ l'(\boxed{f\ (\underline{l\ a})}^{\uparrow}) = r'\ (\boxed{g\ (\underline{r''\ a})}^{\uparrow})\end{array}$$

3. Substitute in the right hand side of the goal:
$$\begin{array}{l}l'(\boxed{f\ (\underline{l''\ a})}^{\uparrow}) = r'\ (\boxed{g\ (\underline{r\ a})}^{\uparrow}) \\ \qquad\qquad \Big\downarrow \text{ using the hypothesis from right to left} \\ l'(\boxed{f\ (\underline{l''\ a})}^{\uparrow}) = r'\ (\boxed{g\ (\underline{l\ a})}^{\uparrow})\end{array}$$

4. Substitute in the right hand side of the goal:
$$\begin{array}{l}l'(\boxed{f\ (\underline{l''\ a})}^{\uparrow}) = r'\ (\boxed{g\ (\underline{l\ a})}^{\uparrow}) \\ \qquad\qquad \Big\downarrow \text{ using the hypothesis from left to right} \\ l'(\boxed{f\ (\underline{l''\ a})}^{\uparrow}) = r'\ (\boxed{g\ (\underline{r\ a})}^{\uparrow})\end{array}$$

We use f and g to indicate some intermediate term structure that blocked rippling.

The third and fourth cases are symmetrical to the first and second, thus it suffices to examine only the first and second cases. In the first case, if we had the resulting fertilised goal as a wave rule, then instead of rippling getting blocked, it would be able to ripple-out the right hand side as follows:

$$\begin{array}{c} l'\ (\boxed{f\ (\underline{l\ a})}^{\uparrow}) = r'\ (\boxed{g\ (\underline{r''\ a})}^{\uparrow}) \\ \downarrow \\ l'\ (\boxed{f\ (\underline{l\ a})}^{\uparrow}) = l'\ (\boxed{f\ (\underline{r\ a})}^{\uparrow}) \end{array}$$

This is clearly a rippling out step as we now have a complete version of the right hand side within a wave front. Furthermore, if the function $\lambda x.\ l'(fx)$ is injective, then this goal can be directly reduced to the induction hypothesis $l\ a = r\ a$, allowing strong fertilisation. Even if it is not a injective function, because we are working in an extensional framework, we can then apply weak fertilisation to reduce the subgoal to an instance of reflexivity. In this sense, for the first case of weak fertilisation, the conjecture can be seen as a suggested ripple rule. We can see from the rippling heuristic why it is beneficial.

Interestingly, such an argument cannot be made for the second case of weak fertilisation. We illustrate this using a concrete example from the proof of $a+0=0$. The step case ripples to $Suc\ (a+0)=0$. At this point three of the four possible weak fertilisations are applicable. The example corresponding to the second case would be to rewrite the goal to $Suc\ (a+(a+0))=0$. Even if this was given as a wave rule to rippling before hand, it would still not help the proof attempt. In this way we can see that the second case of weak fertilisation does not have an interpretation in terms of a benefit to rippling. Furthermore, in practice we have found that such generalisations do not help the proof attempt. This is the motivation for the heuristic that weak fertilisation should only be applied in the first way (and by symmetry in the third).

9.9 Conjecturing Lemmas

We now introduce the motivation for, and machinery to manage, the conjecturing of lemmas. From a logical perspective, the conjecturing of an intermediate lemma, A, can be seen as an instance of the cut rule:

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

In an inductive theory, the failure of cut-elimination [62], means that automatic provers must sometimes conjecture lemmas. Similarly to the case of choosing an induction scheme, this requires the prover to employ heuristics in order to manage the infinite branching in the search space.

We observe that the process of conjecturing a lemmas is closely related to its subsequent application. The two need to be closely related. For example, consider the goal as $a+(b+c) = a+(c+b)$ which, in an extensional logic such that used by Isabelle, can be generalised to the commutativity of addition ($x+y = y+x$). However, this commutativity rule can be applied in several ways, for instance it could be used to rewrite the original goal to $(b+c)+a = a+(c+b)$. However, this would not prove the goal. For this example, the generalisation of the subgoal follows from the argument congruence rule $?x = ?y \Longrightarrow ?f\ ?x = ?f\ ?y$. Because the generalisation is made using argument congruence, once the generalised lemma is proved, it should be applied using argument congruence, not substitution.

The conjecturing and application of lemmas can be specific to a logic. Thus the process needs to be parameterised in terms of the logic-dependent characteristics. We enable this by defining the process of lemma conjecturing in terms of a logic-independent signature which contains:

- a type, `lemmainfo` for holding information regarding how the lemma was conjectured. This can then be used to apply the lemma in the way that corresponds to the way it was conjectured.
- a function `conjecture` which is given a subgoal term and results in a new term to be conjectured as well as an an object of type `lemmainfo`.
- a function `lemma_dmeth` which creates an interpretable method to apply a proved lemma using an object of type `lemmainfo`.

In practice, the conjecturing of lemmas is frequently needed. Furthermore, this is one of the hardest aspect of automated inductive theorem proving. Initial approaches have to the conjecturing of lemmas has been described by Ireland et al. His approach uses proof critics for rippling that are designed to patch failed proof attempt [52]. In her PhD thesis, Kraan has examined various forms of generalisation [60]. Maclean has extended this study in his Masters thesis [65]. An interesting observation from this work is that the employing many kinds of lemma speculation and generalisation do not necessarily benefit the proof attempt. In particular, considering many kinds of lemma speculation can significantly increase the size of the search space. Common subterm generalisation is one of the most successful approaches. Thus we have focused on implementing this for our inductive prover. We also combine this with the argument congruence generalisation described above. Exploring other kinds of lemma speculation and generalisation is left as further work.

9.10 Common Subterm Generalisation

As mentioned above, one of the simplest kinds of generalisations is to replace common subterms with fresh free variables. It was initially proposed by Boyer and Moore and implemented in their original prover and more recently in ACL2. We have adapted this for a higher order setting. We observe that making common subterm generalisations can involve choice and thus provide machinery to navigate the possibilities. This provides a framework for making such generalisations.

Higher Order Generalisations

The idea behind common subterm generalisations is to conjecture a lemma that has variables at the locations where the same subterm occurs in the original goal. In a higher order setting this is complicated by the existence of bound variables and by terms which are syntactically different but which can be normalised to become identical. Thus, searching for common subterms needs to respect term-convertibility. The inter-convertibility of HOAS terms can be addressed by noting that convertible terms denote the same object. Thus terms can be kept in β-η-contracted form to arrive at a canonical representation. In particular, this allows us to ignore subterms that will be removed or modified by β-contraction. We can then use α-conversion as our equivalence check. When considering bound variables that are bound outside of the subterm we are examining, we simply require them to correspond to the same binder.

The motivation behind this adaptation is simply that we want to replace common subterms with variables and result in a generalised term. A proof of the generalised result should be able to be instantiated to prove the original goal. This motivates us to consider generalisations that are syntactically identical to our original term after the appropriate instantiation, i.e which do not require any further β-η-conversion. Without this restriction the set of possible generalisations is significantly larger as it would consider terms which contain subterms that would be removed by β-reduction. Our restriction is basically to conjecture generalisations in β-η-normal form.

Choice in Subterm Selection

Given this simple adaptation of the notion of a common subterm generalisation which makes it suitable for higher order settings, we can consider the choices available during generalisation. In particular, we have to consider how much to generalise. An over-generalisation will result in a conjecture that is not true. However, making an under-generalisation will result in a conjecture that cannot be proved without further generalisation and can lead to significant increases in the size of the search space.

Furthermore, there are often exclusive choices in generalisation. In particular, there are tradeoffs between the number of occurrences of a subterm and the size of the subterm. For example, consider the term $R(f\ (g\ x), h\ (g\ x), f\ (g\ x))$. The subterm $g\ x$ occurs three times, but the larger subterm $f\ (g\ x)$ occurs twice. We must also decide on how many of the common subterms should be generalised.

We were unable to think of theoretical reasons why one approach would be better than another. Thus we developed a general framework for experimenting so that we could empirically examine the result of different approaches.

The basic algorithm behind our machinery is to start off by considering all the leaves of the term tree and grouping them into those that are identical. We then move up the term tree and create the set of new α-equivalent groups. These correspond to common subterms. We use an implementation of Huet's zippers [47] to provide a practical way to move around terms while preserving information about the context. This is needed in order to be able to move up the term tree incrementally. In this way we can traverse the space of possible common subterms. We then introduce filters to remove those that we are not interested in.

One theoretically motivated heuristic that we add to this process is to filter out generalisations of higher-order type. Such subterms would be generalised to functions. However, a function does not offer new variables for induction. Thus we restrict the introduced generalisations to be of an inductive type. One remark that can be made of this is that introducing new function symbols might still be useful as it will introduce new sinks. However, in practice it tends to result in over generalisation.

A motivating example for common subterm generalisation is the proof of the distributivity of addition over multiplication, which results in the following subgoal after weak fertilisation:

$$(c + b * c) + a * c = c + (b * c + a * c)$$

The correct lemma to generalise is the associativity of addition ($(x + y) + z = x + (y + z)$), where the subterms $b + c$ and $a * c$ are generalised.

We found that making the maximal generalisation which has at more than one occurrence was generally the best strategy. Furthermore, we found that making the most possible generalisations at once was also beneficial. At present we do not take care which order these are performed in as conflicting possible generalisations occur rarely. A more refined heuristic might be found by considering a domain where this happens more frequently. This is further work. We note that in the examples we examined, making the maximal number of smallest generalisations would have resulted in the same behaviour. Investigating which strategy is better requires examining other domains and is also left as further work.

9.11 Caching Conjectures

As well as caching proof states during exploration, we cache the result of proof attempts to support the following efficiency heuristics:

1. If a conjecture is proved to be false, then the prover should not consider making this conjecture again. Additionally, the any conjecture, of which the false one is an instance of, should also be avoided.

2. If the search space for the proof of a conjecture is exhausted, then it seems reasonable (and is useful in practice) to avoid making the same conjecture at a later point in proof planning.

3. Once a lemma is successfully proved, if an instance of it is later conjectured, we should re-use the found proof rather than re-proving it.

The cache of proof attempts is stored as contextual information. To allow this to be shared between or-branches in the search space, we follow the approach set out in Chapter 3. In particular, the contextual information holds a reference which allows different branches in the search space to share the cache.

These heuristics are integrated with the lemma speculation and generalisation critic. In particular, whenever a conjecture is made we perform a lookup in the cache. If it has previously been proved, we reuse the existing proof. If it has previously been shown to be false or failed to be proved, then we avoid making this conjecture. Whenever a conjecture is successfully proved we add it to the cache. Similarly, whenever a conjecture fails to be proved, or when a counter example is found, we also note this in the cache. We remark that using a global cache of proved lemmas is difficult in systems such as λ*Clam* where backtracking removes such derived information.

9.11.1 Subspaces for Efficient Lemma Search

In Chapter 3, we introduced the `ENDSPACE` function for techniques. This applies a function when a particular part of the search space is exhausted. We now extend this notion to allow the search space to also be affected as it is being explored. The motivation comes from the following additional heuristics for the conjecturing of lemmas:

- if a conjecture is proved to be false, then the search space of possible alternative proofs should be pruned.
- when a lemma is successfully proved, we do not search for alternative proofs.

This can be seen as another special case of the `MAP` function on techniques. In particular, like the caching technique, it is a map operation over the search space where some information is shared across or-branches. The information we share is about the the conjecture that has been made. In particular, if it has been proved or shown to be false. In either of these cases, the map function modifies alternative branches in the search space to prune further search.

9.11.2 Avoiding Loops

A common problem in many inductive theorem provers, such as ACL2, Clam and Lambda Clam among many others, is that they fail to terminate for many proof attempts. This is problematic is the prover requires to backtrack in order to find the proof. Furthermore, from the perspective of an interactive user this can make the systems extremely painful to use.

We found it essential for the behaviour and robustness of our proof technique to avoid non-terminating branches of the search space. We do this by employing the embedding machinery described in Chapter 7 also used for rippling to check if a previous goal of which the current state is a subgoal of, embeds into the current subgoal. We slightly broaden this notion by adding the proviso that the current subgoal also does not contain any constants that do not occur in the main goal. From a theoretical point of view, Kruskal's theorem shows that using the embedding check to prune the search space will make it finite [63]. In particular, it states that there is no infinite sequence of trees without an earlier tree embedding into a latter one.

This is implemented by providing a collection of goals that we have seen so far, and each time a new goal should be considered, we check if one of the previous goals embeds into the new goal and that the new goal contains no new constants. At present we hold the previous goals as a list which makes the loop-checking become increasingly slow as the number of seen goals increases. Further work includes the development of an embedding net to simultaneously check against all the previously examined goals.

We note that we can also provide a version of this check for proof exploration such as rippling. This can be implemented easily using the `FOLD` function of our reasoning technique language. In particular, we fold the cache checking function over each path in the search space. The cache checking function holds the goals we have seen so far and is used to check that a new goal is not embedded into by an earlier one. This provides a much more effective way of avoiding non-termination for exploration than using the caching described earlier. However, because of the lack of a way to check against several goals simultaneously, it also incurs a much higher cost in time as the proof depth increases.

Within our inductive prover we use the loop-checking to avoid loops in conjectures. This simple mechanism is the essential tool that allows our inductive prover to terminate when applied to non-theorems. It is also what allows false conjectures, or unsafe branches of the search space to be backtracked over instead of falling into an infinite loop of conjecturing.

9.12 Related Work

We now related this inductive theorem proving technique to other systems.

The Boyer-Moore Prover and ACL2

Perhaps the best known prover is the Boyer-Moore prover [12], the latest version of which is ACL2 [57]. These systems use recursion analysis for the selection of induction schemes and then employ a carefully designed simplification tool followed by a version common subterm generalisation. ACL2 also incorporates various decision procedures and many other heuristics for unfolding definitions.

Recursion analysis enables it to prove theorems which require more complex induction schemes. We note that incorporating a more powerful system to select induction schemes in IsaPlanner is a simple extension of our technique and might significantly improve its performance on these examples.

Another key difference in the strategies employed is that ACL2 and the Boyer-Moore prover simplify the step cases of inductive proofs where we employ rippling. Rippling supports the incorporation of specialised proof critics. However, we have not implemented these, so we cannot yet analyse the importance of these in practice. However, rippling does allow rules to be used in both directions while ensuring termination. In terms of underlying logic, our system is generic, where ACL2 is closely tied to their less expressive logic.

Perhaps the most notable difference in using ACL2 is that it rarely terminates for non-theorems or for problems which it cannot prove. The machinery to avoid loops in IsaPlanner allows it to behave more robustly. Another difference is that ACL2 avoids any explicit notion of backtracking or representation of the search space. Modifying the proof machinery in ACL2 is difficult as it is large and complex. In contrast, the inductive theorem proving technique in IsaPlanner is relatively simple to modify and very modular in design. An interesting difference in the generalisation and conjecturing of lemmas is that ACL2 makes generalisation inline, where IsaPlanner separates them as distinct results that can be reused.

Boyer-Moore Automation for HOL

Boulton has re-implemented the heuristics from the original Boyer-Moore prover within the HOL system [10]. However, these are still only applicable to the first-order problems. The implementation follows the design of the Boyer-Moore prover whcih uses a waterfall structure and provides a single top level tactic that combines induction and simplification. The main feature of this re-implementation is that, like our system, the steps are justified in terms of the underlying proof system.

Clam and λ*Clam*

The inductive prover we have developed is based on that implemented in the Clam and λ*Clam* systems. An important difference is within rippling, where we use a more expressive mechanism for annotation and provide a number of efficiency measures. Additionally, we make use of Isabelle's induction and simplification tactics as well as provide caching for lemma speculation.

Boulton and Slind [9] developed an interface between Clam and HOL. Unlike our approach which tries to take advantage of the tactics in Isabelle, their interface did not use the tactics developed in HOL as part of proof planning. Additionally, problems were limited to being first order, whereas our approach is able to derive proof plans for higher order theorems.

INKA and NuPrl

A general notion of annotated rewriting has been developed by Hutter [48] and extended to the setting of a higher order logic by Hutter and Kohlhase [49]. They develop a novel calculus which contains annotations. This is a mixture between dynamic and static rippling as after each rewrite skeleton preservation still needs to be checked, but the wave rules can be generated beforehand.

A proof method that combines logical proof search and static rippling has been implemented for the NuPrl system by Pietntka and Kreitz [87]. Their implementation is as a tactic without proof critics and focuses on the incremental instantiation of meta variables. They employ a different measure based on the sum of the distances between wave fronts and sinks.

In Summary

A general salient feature of the technique presented in this chapter is that the proof plan's execution is interleaved with the proof planning attempt. This makes the final derived theorem sound with respect to only Isabelle's logical kernel.

This is in contrast to the implementation in λ*Clam* which does not have any object level verification of the proof plans. The logical kernel is also significantly smaller than the trusted code in ACL2. As such, the guarantee of soundness is closer to that provided by the NuPrl, the HOL/Clam combination and the Boyer-Moore perover's reimplementation in HOL. With respect to these systems, as well as λ*Clam* our tactic provides a much more expressive framework for experimentation proof planning techniques such as rippling. The implemented efficiency measures also make it more significantly efficient than the Clam and λ*Clam* systems, as detailed in Chapter 10

9.13 Evaluation and Further Work

We now present a brief sketch of the kinds of problems that our inductive strategy works well for, and those for which it usually fails. This gives a direction for further work.

Success

As shown in chapter 10, our inductive proof techniques out perform λ*Clam* in terms of speed and are roughly equivalent in terms of power. In particular, our techniques work well on problems that are equational, concern recursive functions, and contain recursively defined variables.

Failure

We have observed a number of situations when our inductive proof strategy fails:

Conjecturing the wrong lemma:

This typically happens when an over generalisation is made. One of the reasons for such over generalisations is that we separate the induction hypothesis from the goal and thus when conjecture is made before weak fertilisation, it does not include the induction hypothesis as an assumption. This results in conjecturing a lemma that is too general. In many situations, the conjecture is true only in the context of the assumption. Furthermore, in some situations a common subterm generalisation needs to be made that includes a subterm in the induction hypothesis.

Further work thus includes developing machinery to treat assumptions more effectively during inductive proof. Another area of further work would be to use counter example finding tools to prune the space of possible conjectures. As well as over generalisation, it is also common for theorems to need more sophisticated generalisations. Developing such machinery and investigating how to use it without blowing up the search space is also an interesting area for further work.

Missing the right induction scheme:

Our machinery for selecting induction schemes only considers those that come from the recursive types of variables in the conjecture. Whenever another induction scheme is needed, it must be supplied by the user. Developments such as Gow's [44] offer the possibility of proving a custom induction scheme during the proof attempt. However, further work is needed to clarify what effect they have on the search space.

Needing further case analysis:

When functions are defined using if-statements or case-statements, it is common for a case split to be needed in proofs concerning them. However, at present the only time case analysis is performed is by using induction. When induction does not split a variable in the appropriate way then further case analysis is needed. However naively introducing case analysis creates a huge blow-up in the search space. Ways to include selected case splitting would be a useful area of further research.

Too large a search space:

The main branches points in the search space of our proof technique are within rippling and during the selection and application of an induction scheme. The primary reason for rippling to cause branches is that and-choices in rippling, such as the possibility to rippling the left hand side and the right hand side of an equation, are currently considered as or-choices. Improving the way rippling examines rewriting could significantly improve its behaviour. Using more sophisticated mechanisms for the selection of induction schemes could also improve the size of the search space. Another approach would be to experiment with other search strategies, such as best first search. This would allow the use of heuristic approaches to the exploration of the search space and could avoid getting stuck on a bad path which sometimes happens during depth first search.

9.14 Conclusions

We have presented a technique to prove theorems by induction. It employs rippling, simplification and the conjecturing of lemmas. We also incorporate various caching and efficiency measures as well as several heuristics. We note that these techniques make profitable use of the `FOLD` and `MAP` functions in our technique language. This shows that the extra expressivity of our technique language is useful in practice. More generally we find that writing techniques in terms of operations on the search space is convenient. We see in Chapter 10 that these are also empirically important for the efficiency of our inductive prover.

The contextual information is held in one store. This means that each kind of contextual information must have a default value. Similarly to the construction of proof plans, a problem with this is that it does not allow us to make use of ML's type checking to ensure that contextual information exists. Thus we cannot detect many errors at type-checking time. Trying to make more use of type-checking for the writing techniques is left as further work.

The induction technique selects and applies an induction scheme based on the inductively defined variables in the goal. Although there are various ways to select the variable for induction, such as ripple analysis [17], we found that search backtracks quickly enough for the choice of variable to be largely insignificant in the domains we examined. This is partially due to the caching mechanism that allow proof planning to use a significant portion of the failed proof attempt. For example, when proving $i^{(j+k)} = i^j \cdot i^k$ in Peano arithmetic, wrongly trying induction on i results in the proof of 3 of the 4 needed lemmas, and the only additional lemma to prove is the trivial theorem $x + 0 = x$.

Chapter 10

Experiments with Rippling and our Inductive Prover

In this chapter we perform experiments with our rippling machinery and techniques for inductive theorem proving. These highlight the more effective varieties of rippling. The experiments also show the importance of the representation of definitions on the ability to prove theorems. We also compare our techniques with the λ*Clam* system to get a more general assessment of our inductive prover's utility. This shows that our work provides a powerful and efficient inductive theorem prover and more generally that our proof planning framework provides a practical tool for experimental research.

10.1 Introduction

In this chapter, we describe experiments with our inductive theorem prover and the rippling machinery used within it. The general aim is to illustrate the utility of our proof planning framework as a tool for performing experiments with proof planning techniques. Our evaluation also provides empirical evidence as to which variety of rippling works best in practice. It also highlights the importance of the representation when experimenting with techniques for inductive proof.

Firstly, we evaluate the choices for an implementation of rippling within the context of our inductive theorem prover. In particular, we consider various measures for rippling. We examine the efficiency and power over different formalisations of Peano arithmetic that contain no extra configuration or proved lemmas. This reflects the situation when a user starts a fresh theory development with new definitions and recursive types. These experiments are used to select the version of rippling that is the most effective, which we then examine further in the following evaluations.

Our second evaluation is a comparison with the λ*Clam* system in the domain of ordinal arithmetic. This gives a comparative study to analyse the relative efficiency of our prover. We take care to use the same definitions in both systems.

10.2 Theories of Peano Arithmetic and Varieties of Rippling

We now describe an experiment which compares different versions of rippling on problems in a variety of formalisations of Peano arithmetic. The goal is to ascertain which version of rippling performs most effectively in terms of the number of theorems proved and how quickly it terminates.

We first introduce the methodology, background theories and problem set. We then describe the variations of rippling which we experiment with. Finally, we present the results and analyse them.

10.2.1 Methodology

The goal of these experiments is to find which version of rippling is best for the automation of inductive proof. We carry out our experiments in the domain of Peano arithmetic, although the methodology could easily be employed for other domains. A secondary goal of this experiment is to examine the effect of the formalisation on the ease of automation.

Non-theorems

We consider the performance on non-theorems to be important because it is common for users to make errors in definitions or conjectures which can cause them to spend significant effort trying to prove non-theorems.

To create a test set of non-theorems we analysed errors, including copy and paste errors, that we made in theory development. We observed that these can be reproduced by systematically transforming theorems into non-theorems. In particular, we can modify variables and constants to create terms that are similar to existing theorems but which not true in the theory. For example, from the theorem $a + 0 = a$ in Peano arithmetic, we get the non-theorems $a + 0 = 0$ and $a * 0 = a$. We also perform variable renaming, thus from the distributivity of addition over multiplication, we get non-theorems such as $a*(b+c) = c*b+c*a$ and $a*(b+c) = c*b+c*a$.

This provides us with a large set of non-theorems many of which occurred in practice and many of which we believe could feasibly be made in interactive developments. However, we note that these may be biased due to the kinds of errors we make within the domains examined. A more systematic and wider study of non-theorems would be of interest and could help guide the development of proof techniques.

Measures

To measure the quality of a version of rippling, on a problem in a given theory, we consider the time taken and whether or not the conjecture was proved. We also distinguish non-theorem from theorems which gives us one of the following measurements for each problem:

- the time to prove a theorem, or
- the time to give up when a theorem cannot be proved, or
- the time taken to give up on a non-theorem.

We put an upper limit of five seconds on the time. This is in order to get a concrete value for cases when the proof technique fails to terminate. This also reflects the behaviour of users in an interactive setting. They are likely to cancel proof attempts that take too long. The timings are obtained from a standard 2GHz Intel PC with 512MB of RAM, using the development snapshot of Isabelle from the August 15 2005 with PolyML 4.1.3.

Techniques will often give-up when trying to prove a theorem, or non-theorem, before the time-limit when the search space is exhausted. The speed of termination for cases when proofs are not found is important as quick failure is a better result for the user than non-termination. This motivates the measurement of time for failure.

The separation of timings for failed proof attempts between theorem and non-theorems is because we expect counter-example finding tools to benefit the identification of non-theorems. However such tools would not help when considering a theorem. Thus, from this separation, we also get an idea of which timings could be improved by employing counter example finding tools.

Background Theory and the Setup of Proof Tools

In this experiment we do not provide any background lemmas. This reflects the initial state of a proof development and is thus a good indication of the level of automation that is provided in a new theory. In many evaluations in the literature, techniques have only been evaluated on a single formalisation of the domain. However, because formalisation offers the user many choices in the way they define functions, which can affect the ease of automatic proof, we believe that a more systematic study is needed. For this reason we consider many formalisations of the same theory. In particular, we investigate the hypothesis that the formalisation has an important effect on the ease of automation.

Closely related to the provision of lemmas is the setup of the existing proof tools. When such machinery has a complex configuration, it poses a problem for evaluation: which setup should be considered? To clarify the evaluation of our inductive theorem prover, we evaluate it in theories with only Isabelle's default setup of other proof tools which gives all recursive definitions to the simplifier. Similarly, these definitions are also given to the rippling machinery.

This setup of background theories, with no lemmas and with the default setup of proof tools, is common in theory development. It is exactly the configuration when a user starts a new theory in Isabelle. This approach allows us to gauge the effectiveness of our rippling machinery and inductive theorem prover on a new theory. Thus the results we get are indicative of using our inductive prover in real formalisations.

10.2.2 Primitive Recursive Theories of Peano Arithmetic

Peano arithmetic describes the natural numbers in terms of a zero constant and a successor function, which we characterise using the following datatype:

$$\begin{aligned} nat &= 0 \\ &\mid \; Suc\ nat \end{aligned}$$

We then define functions for addition, multiplication and exponentiation. However, in the definition of these functions we are faced with choices. For now we only consider primitive recursive definitions as this is the easiest way to define such functions in Isabelle. Even just considering this definitional mechanism addition can be defined in four ways:

Definition 10.2.1:

$$0 + y = y$$
$$(Suc\ x) + y = Suc\ (x + y)$$

Definition 10.2.2:

$$x + 0 = y$$
$$x + (Suc\ y) = Suc\ (x + y)$$

Definition 10.2.3:

$$0 + y = y$$
$$(Suc\ x) + y = x + (Suc\ y)$$

Definition 10.2.4:

$$x + 0 = y$$
$$x + (Suc\ y) = (Suc\ x) + y$$

Although these definitions can all be proved to be equivalent they do not provide the same behaviour for the automation of proof. For example, using definitions 10.2.1 and 10.2.2, the commutativity of addition can be proved using only rippling-out, however using definitions 10.2.3 and 10.2.4 requires rippling-in. These differences in the success of a proof technique within different formalisations of the same theory highlight the importance of evaluating techniques over a variety of formalisations.

Multiplication can also be defined in four ways, given in appendix A. Exponentiation can also be defined in different ways, although they are not all equivalent. In particular, the treatment of 0^0 as either 0 or 1 is an exclusive choice which results in different theories. Following the formalisation used in Isabelle, we consider only the case when $x^0 = 1$. The exact definition is given in appendix A.

10.2.3 The Problem Set

Even just considering the above choices for definitions, we have 32 formalisations of Peano arithmetic. Within each of these, we consider theorems from the literature and those that have been proved in Isabelle's existing formalisation of Peano arithmetic, such as:

$$
\begin{aligned}
a+b &= b+a \\
b+c+a &= b+(c+a) \\
k*(m+n) &= (k*m)+(k*n) \\
(m*n)*k &= m*(n*k) \\
x*(y*z) &= y*(x*z) \\
i^{(j+k)} &= i^j * i^k \\
i^{(j*k)} &= (i^j)^k
\end{aligned}
$$

For these theorems, we consider both left and right hand side cases of distributivity rules. For instance, we examine both $k*(m+n) = (k*m)+(k*n)$ and $(m+n)*k = (m*k)+(n*k)$. This is to avoid biasing the problem set towards one theory, which may find one version easier to prove than another. Thus, when we consider the effect of the formalisation, we initially only exmaine results from this set of problems.

When testing techniques, we also consider various specialised versions of the above theorems, such as:

$$
\begin{aligned}
0+0+a &= a \\
(m+Suc\ 0 = n+Suc\ 0) &= (m=n) \\
a+0+a &= a+a \\
(m*Suc\ b)*k &= m*k+(m*b)*k \\
i^{(j+k+l)} &= i^j * i^k * i^l
\end{aligned}
$$

It is interesting to test such special cases of theorems because they can be more difficult to solve as they require the correct generalisation to be made automatically. In practice, such theorems can also arise as subgoals of other lemmas.

As mentioned earlier, we also test non-theorems, such as:

$$
\begin{aligned}
0+a &= 0 \\
m*n &= m*m \\
(m+n)*k &= (m+k)*(n+k)
\end{aligned}
$$

The full list of theorems and non-theorems is given in appendix A.

Each version of Peano arithmetic provides a different context for the proof attempt, thereby making each conjecture a different problem for each version of the theory. This results in a total of 2944 problems to be tried with each technique. Although this may sound like a large problem set, we note that these are all within the domain of equational theorems without premises. In effect, this is still a relatively small domain. In order to provide a wider picture of the behaviour of our techniques we suggest experimentation with other definitions, constants, and conjectures as future work.

10.2.4 The Varieties of Rippling

We consider the following choices, introduced in chapter 7, for an implementation of rippling:

- Whether inward wave-fronts are restricted to occurring over sinks. The technique name starts with `R` when the restriction on sinks is enforced, and with `U` when inward wave-fronts are unrestricted.
- Whether a higher-order or first-order measure of depth is used. The second letter of the technique name will be `H` and `F` respectively.
- Whether adjacent wave fronts are allows to have different directions. Technique names will end with `C`, indicating that only compound wave fronts are allows, and `S` when wave-fronts are split to allow different directions.

In addition to the 8 varieties of rippling that arise from these choices, we also consider a measure, introduced in chapter 7, that counts the sum of the distances between wave fronts and the top of the term tree or nearest sink. We will call this version of rippling `dsum`. The versions that use the traditional list measure and which have the above choices are written with three letters indicating the exact variation.

10.2.5 Results and Analysis of the Techniques

A summary table showing the results for each technique in terms of the timings and number of theorem proved is given in Figure 10.1. This graph shows the average timing for the different varieties of rippling on theorems and non-theorems. The number of theorems proved is shown underneath the name of the technique. The techniques are ordered from left to right by the

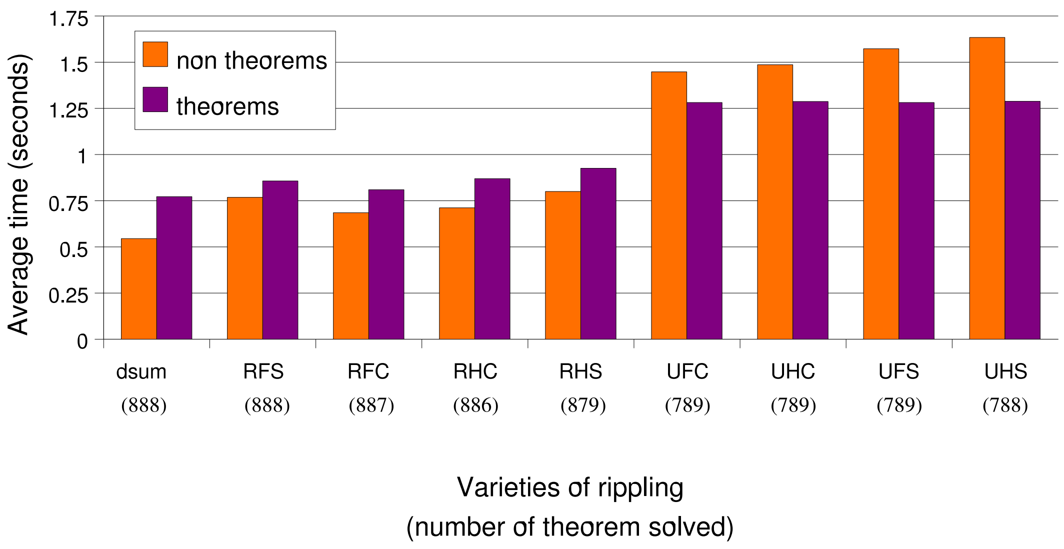


Figure 10.1: *The number of theorems proved and timings for the tested varieties of rippling showing results on theorems and non-theorems.*

number of theorem proved then by the average timing. We now consider different aspects of these results in more detail.

We do not separate the timings for proofs of specialised-case theorems from the general ones as the they are have approximately the same relationship.

The Best Variety of Rippling

A surprising result is that the simple heuristic measure, introduced in chapter 7, that counts the distance from each wave front to the nearest sink, outperforms all the varieties of rippling that use the list measure. In terms of power, it performs equally best with the RFS variety but in terms of timing, it outperforms all other approaches. The reason for this is that the `dsum` version of rippling has a smaller search space. What is surprising is that this search space does not exclude any proofs that are within the varieties based on the list measure. This is a strong indication that further work on the measure may lead to significant improvements in inductive theorem proving.

The Inward Sink Restriction

We observe from the results that when the restriction on inward wave fronts occuring over sinks is enforced, indicated by the technique name starting with “R” instead of “U”, the number of theorems proved increases and the timing for proof attempts decreases. This shows that the sink restriction provides an effective means to decrease the search space without losing proofs. All techniques using the unrestricted version were slower and found fewer proofs.

From analysing the specific cases when techniques were able to prove results we note that the unrestricted versions did not find any proofs that were missed by the restricted versions. Part of the reason for this is that if the a technique is too slow, then the timeout will be reached more frequently causing the technique to be considered to have failed. However, part of the reason is also that different conjectures are made, which combined with depth first search can lead to non-terminating branches in the search space. This causes the unrestricted version of the technique to find fewer solutions.

This highlights an interesting feature of search space pruning: when the search strategy will not cover the whole of the search space, either because of a timeout or incompleteness, pruning the search space can result in introducing proofs. In the above experiments we used depth first search, which is incomplete as well as a relatively short timeout. This motivates experimentation with other search strategies, and further approaches to search space pruning, which is left as an interesting avenue for further work.

First-order and Higher-order Measures of Depth

To view the effect of the different measurements of term depth, we re-order the entries in the results bar chart so that each version with a first-order measure is next to the otherwise equivalent variety with higher-order measure. This is shown in Figure 10.2.

This bar chart shows that our proposed measure for height in the term tree improves the effectiveness of rippling: the varieties that use our first-order height measure are consistently more powerful and quicker than those which use the higher-order measure. The speed improvement is because the first order measure offers fewer ways to annotate a goal, and thus gives a smaller search space. Because no proofs are lost, we conclude that using the first-order height measure is a useful pruning of the search space.

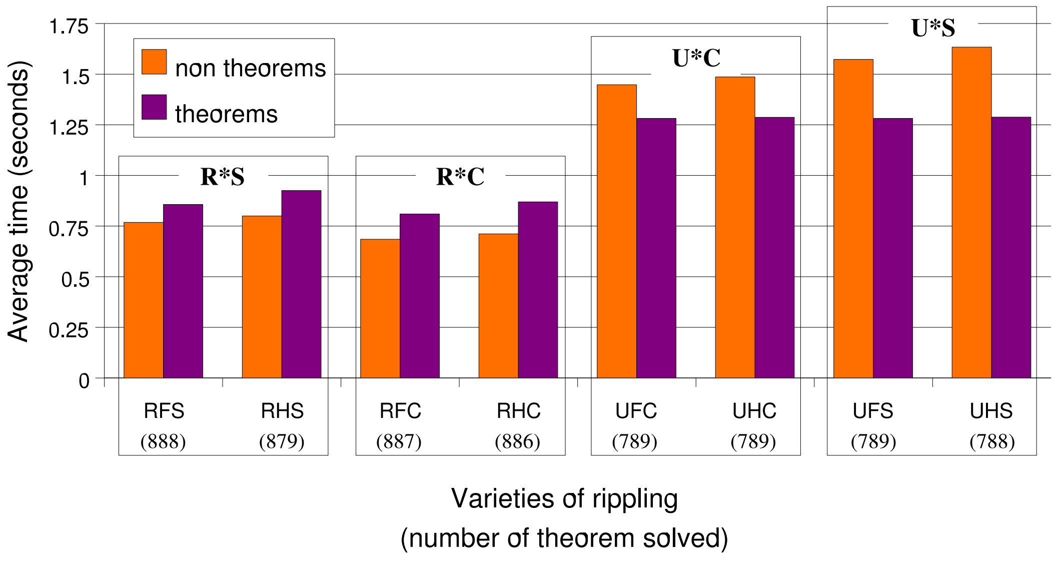


Figure 10.2: *The number of theorems proved and timings for list based techniques grouped into pairs where the only difference is the use of first-order and higher-order measures for depth in the term tree.*

Compound and Split Wave Fronts

We present a view of the effect of the compound wave front restriction in Figure 10.3. In the bar chart, the techniques have been reordered so that each variety with the restriction is next to the otherwise equivalent version without it.

Interestingly, these results show that although the compound wave front restriction improves the timings in all cases, there is a proof that is lost through disallowing adjacent wave fronts to have different direction. In particular, this happens when proving $i^{(j+k)} = i^j * i^k$. With the technique `RFC`, the proof attempt times out, but with `RFS` the problem is proved within 4 seconds. This shows that there are some proofs that with a limited time can be found by using split wave fronts but which using compound wave fronts prunes. However, we note that given more time, the technique `RFC` does find a proof.

Hierarchy of Technique Power

The charts presented earlier do not show if there are any proofs found by `dsum` not found by `RFS` and visa-versa. It could be that the techniques solve different classes of problems. However, examining the specific results in more detail showed that no such cases arise. Thus, the set of

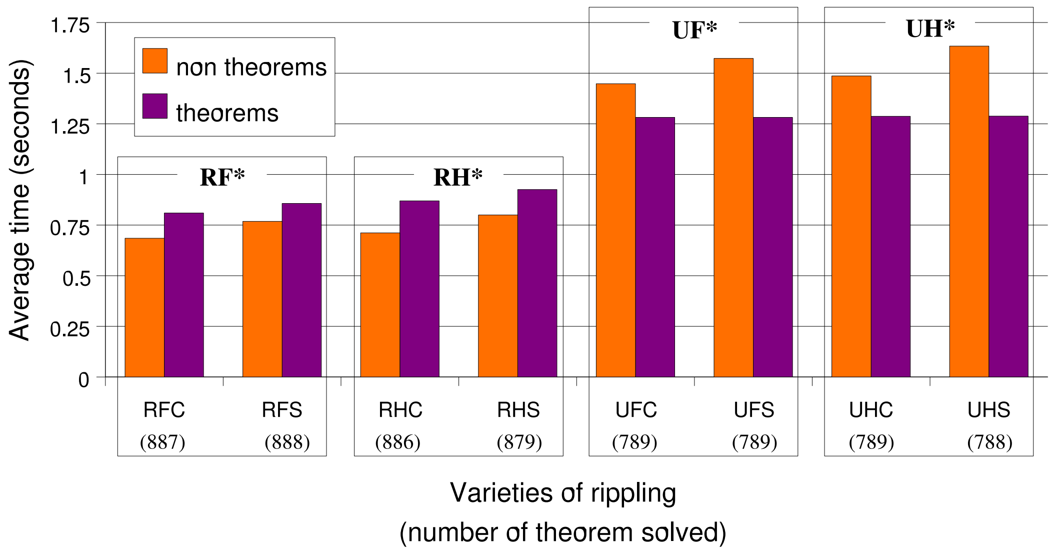


Figure 10.3: *The number of theorems proved and timings for list based techniques grouped into pairs where the only difference is the use of the restriction to compound wave fronts.*

proofs found by `dsum` is identical to `RFS`. Furthermore, we examined the results to determine if any weaker technique ever found a proof for a theorem that was not found by a more powerful technique. This revealed that the power of the techniques shown in Figure 10.1 is strict in the sense that the more powerful techniques can prove everything than the weaker ones can. There was one one exception to this. The theorem $x*y*z = x*z*y$ can be solved by `RHS` within the time limit, but not by `RHC`. This is another example of where the the compound wave front restriction prunes a proof from the search space. However, like the earlier example, the `RHC` technique does find a proof if given more time. Apart from this special case, the results show that the weaker varieties of rippling do not solve problems that more powerful one cannot.

Timings for Non-theorems

We note that average time for a proof attempt of a theorem is not significantly different to that for a non-theorem. For the less restricted varieties of rippling the average time to tackle a non-theorem is worse than a theorem, but for the more restricted varieties the inverse is the case. However, this is likely to be a result of the problem set as both many non-theorems as well as theorems reach the timeout. Thus the averages are reflective of the proportion of easy to difficult cases. However, we can observe that the inward-sink restriction improves the time for tackling non-theorems more than theorems. This is because for non-theorems the whole of the search space must be exhausted before the technique will give up, but when a proof is found the rest of the search space is not explored. Thus any modification to the proof technique that exponentially cuts down the size of the search space will on average speed up the timing for proof attempts of non-theorems more than theorems.

10.2.6 Results and Analysis of the Formalisation

The second variable which we examine is the effect of the formalisation on the difficulty of the problem set. As mentioned earlier, for this analysis we only consider the set of general theorems that were picked to be fair to the different formalisations. If we do not do this, then the evaluation is of the problem set rather than the formalisation.

We consider the difficulty of a formalisation with respect to the number of theorems proved within it and the time taken. We note that the formalisations that were more difficult for one technique were also more difficult for the others. In this sense we found that the formalisations were fair to all theories. This was expected as the techniques are fairly similar. This allows us to define the difficulty of a formalisation in terms of the average number of techniques that can solve a problem and the average time taken.

The difference in difficulty between formalisations

Given this measure of difficulty, the different formalisations of Peano arithmetic had significant differences in difficulty. In terms of the number of theorems proved, problems in the easiest formalisation were solved by an average of 84% of techniques. In contrast to this, within the most difficult formalisation problems were solved by an average of 54% of techniques. The best average time was 0.76 seconds per problem compared to a worst case of 1.5 seconds. From these statistics we conclude that the formalisation has a significant effect of the difficulty of proving theorems. Thus it is important for evaluating techniques to consider this issue.

Effective formalisations

We remark that our techniques were generally more effective, in terms of power and speed when addition is defined so that the successor moves outwards rather than changing argument: definitions 10.2.1 and 10.2.2 produce more proofs and generally result in smaller search spaces than definitions 10.2.3 and 10.2.4.

The realtionship between ease of proof and speed of proof

Interestingly, the worst case for timings was not the theory in which least theorems could be proved. This shows that there is a difference between some theories which are difficult, with respect to our proof techniques, and others which are easier but require more work in the proof. This is because, in some more difficult theories, techniques will give up on theorems more quickly than they can prove them in the easier theories. The result of this is that we cannot consider just measuring the time for proof attempts as this does not reflect the ability of a technique to prove theorems within the domain.

10.3 A Brief Study in Ordinal Arithmetic and Comparison with λ*Clam*

In this section we compare our inductive proof technique, using the `dsum` variety of rippling which was found to be more effective, with λ*Clam* system. We perform this evaluation on the domain of ordinal arithmetic as this was the case study for higher order rippling in λ*Clam*. This involves a development of some ordinal arithmetic in Isabelle which also shows the utility of our inductive prover for formalisation.

Following the methodology we introduced earlier, to distinguish the automation provided by our techniques from that gained by working in the well-developed theories, the tests were carried out in a formalisation without any auxiliary lemmas. All needed lemmas must be conjectured and proved automatically. Also like the previous experiment, we used the default setup of proof tools in Isabelle. The same approach is taken in the formalisation of ordinal arithmetic in λ*Clam*.

10.3.1 A Theory of Ordinal Arithmetic

We now briefly describe our formalisation of ordinal arithmetic in Isabelle which follows that implemented in λ*Clam* by Dennis and Smaill [35]. Ordinal notation is defined using the following datatype:

$$\begin{aligned} ordinal = {} & 0 \\ & \mid \; Suc \;\; ordinal \\ & \mid \; Lim \;\; (nat \to ordinal) \end{aligned}$$

A feature of Isabelle is that the transfinite induction scheme for the ordinal notation is automatically generated by the datatype package [93]. This is then automatically used by the induction technique in IsaPlanner. The injectivity rule for ordinal notion is also automatically proved and is thus be given directly to the rippling machinery.

The arithmetic operations on ordinals are defined using Isabelle's primitive recursive package. For example, addition is defined as follows:

```
primrec
  ord_add_0   : "(x + 0) = (x :: Ord)"
  ord_add_Suc : "x + (Suc y) = Suc (x + y)"
  ord_add_Lim : "x + (Lim f) = Lim (λn. x + (f n))"
```

The other arithmetic operations are defined and named similarly. See appendix B for complete proof script. Using these definitions, the induction and rippling technique is able to derive and produce Isabelle/Isar proof scripts for all the theorems proved in the work of Dennis and Smaill. The theorem that takes longest to prove is the following:

```
theorem "x ^ (y * z) = (x ^ y) ^ z"
proof (induct "z")
  show "x ^ (y * 0) = (x ^ y) ^ 0" by (simp)
next
  fix Ord :: "Ord"
  assume ind_hyp1: "x ^ (y * Ord) = (x ^ y) ^ Ord"
  have "x ^ (y * Ord + y) = x ^ (y * Ord) * x ^ y" by (rule auto_lemma_0)
  hence "x ^ (y * Ord + y) = (x ^ y) ^ Ord * x ^ y" by (subst sym[OF ind_hyp1])
  hence "x ^ (y * Ord + y) = (x ^ y) ^ Suc Ord" by (subst ord_exp_Suc)
  thus "x ^ (y * Suc Ord) = (x ^ y) ^ Suc Ord" by (subst ord_mul_Suc)
next
  fix f :: "nat => Ord"
  assume ind_hyp1: "!!xa. x ^ (y * f xa) = (x ^ y) ^ f xa"
  have "Lim (λn. (x ^ y) ^ f n) = Lim (λn. (x ^ y) ^ f n)" by (simp)
  hence "Lim (λn. x ^ (y * f n)) = Lim (λn. (x ^ y) ^ f n)" by (subst ind_hyp1)
  hence "Lim (λn. x ^ (y * f n)) = (x ^ y) ^ Lim f" by (subst ord_exp_Lim)
  hence "x ^ Lim (λn. y * f n) = (x ^ y) ^ Lim f" by (subst ord_exp_Lim)
  thus "x ^ (y * Lim f) = (x ^ y) ^ Lim f" by (subst ord_mul_Lim)
qed
```

In this proof script, the subst steps refer to the use of our fine grained equational reasoning tactic described in chapter 8. The names `ord_exp_Suc`, `ord_exp_Lim`, `ord_mul_Suc` and `ord_mul_Lim` are of the defining equations in the recursive definitions for exponentiation and multiplication. Note that, for the above proof, the following needed lemmas are all automatically conjectured and proved:

```
lemma auto_lemma_5: "g0 + (g2 + g1) = g0 + g2 + g1"
lemma auto_lemma_4: "g1 * g2 + g1 * g0 = g1 * (g2 + g0)"
lemma auto_lemma_3: "g1 * g0 * x = g1 * (g0 * x)"
lemma auto_lemma_1: "0 + g1  = g1"
lemma auto_lemma_0: "x ^ (g0 + y) = x ^ g0 * x ^ y"
```

In these lemmas the variable names starting with g indicate that they were created from a common-subterm generalisation. Proofs such as this one show the utility of using Isar as the language for proof plans. It allows the proof plans to be presented and examined. Furthermore, the Isar script representation of the plans can be copy-and-pasted directly into theory developments. The main difference between the proofs automatically generated by IsaPlanner and the user-generated ones is that user's scripts tends to prove the lemmas and add them to the simplification machinery. This typical approach results in a series of proofs which just apply induction then simplification. For example, the hand written proof script for the above theorem is as follows:

```
lemma assoc_add: "x + (y + z) = (x + y) + z"
by (induct z, simp_all)

lemma distr: "x * y + x * z = x * (y + z)"
by (induct z, simp_all add: assoc_add)

lemma assoc_mult: "(x * y) * (z :: Ord) = x * (y * z)"
by (induct z, simp_all add: distr)

lemma add_zero2[simp]: "0 + x = x"
by (induct x, simp_all)

lemma exp_add: "x ^ (y + z) = (x ^ y) * (x ^ z)"
by (induct z, simp_all add: assoc_mult)
```

theorem `exp_mult_exp: "`x ^ $(y * z) = (x$ ^ $y)$ ^ z`"`
by `(induct z, simp_all add: exp_add)`

In comparison, the automatic rippling-based proof scripts are more verbose and explicitly show where lemmas are used. They also require the user to provide some lemmas to the simplification machinery by hand. For example, users usually do not provide associativity or commutativity rules to the default simplification set. However, when re-ordering arguments in an associative-commutative operator is needed they must then explicitly give the simplifier the needed rule(s). When theorems can be proved by our inductive prover, such interaction is not needed because the rules are conjectured and proved as they are needed. Furthermore, our caching of proof attempts also frequently allows the prover to avoid the need to reprove lemmas within the same proof attempt.

The proof planning machinery gives the user the choice of copying the generated proof plans into their proof script to get more verbose scripts, or hiding the details within a single command. Either way, the automation is significantly greater than that already present in Isabelle as the user is can directly prove the more difficult theorems without first configuring the simplifier.

10.3.2 Comparison with λ*Clam*

As a comparison with λ*Clam* we observe that:

- λ*Clam* has specialised methods for various domains, such as non-standard analysis [67], which provide it with the ability to prove some theorems not provable by IsaPlanner's default rippling machinery.
- IsaPlanner makes use of Isabelle's tactics such as the simplifier which is user configurable and can be used to provide conditional rewriting for the base cases of inductive proofs. This provides IsaPlanner with automation not implemented in λ*Clam*.
- IsaPlanner executes the proof plan, ensuring soundness of the result, where λ*Clam* is currently not interfaced to an object level theorem prover.
- Higher order rippling in IsaPlanner appears to be significantly faster than in λ*Clam*. Simple theorems are solved in almost equivalent time but those with more complex proofs involving lemmas are much quicker to plan *and* prove in IsaPlanner. For example, the

ordinal theorem $x^{(y \cdot z)} = (x^y)^z$ takes over five minutes in λ*Clam* compared to 2 seconds in IsaPlanner. We believe that this is largely due to our caching of intermediate results.

- The resulting proof plans from IsaPlanner are readable and clear whereas those produced by λ*Clam* are difficult to read. For example, at present the proof plan generated by λ*Clam* for the associativity of addition in Peano arithmetic is 12 pages long (without any line breaks). The proof script generated by IsaPlanner is one page long and in the Isar style.
- Upon failure to prove a theorem, λ*Clam* does not give any helpful results, whereas IsaPlanner is able to provide the user with proofs for useful auxiliary lemmas. For example, upon trying to prove $x^{(y \cdot z)} = (x^y)^z$ in Peano arithmetic, IsaPlanner conjectures and proves 13 lemmas, including the associativity and distributivity rules for multiplication.

We remark that some of the automatically conjectured and proved lemmas can be obtained by simplification from previously generated ones. This shows a certain amount of redundancy in the generated lemmas. In future work, we intend to prune these and identify those which are of obvious use to the simplifier.

10.4 Conclusions

We have performed a novel study into the effectiveness of varieties of rippling. This has highlighted that an alternative measure, introduced in chapter 7, outperforms all traditional varieties based on list-measures. We performed this study over a range of formalisations of Peano arithmetic. This has shown the significant effect of the exact definitions used on the difficulty of proof for our inductive techniques.

We have also compared our inductive theorem proving technique with the λ*Clam* system. This has shown them to be roughly equivalent in terms of proving power if enough time is provided. However, it has also shown that our system is significantly quicker than λ*Clam*. These experiments were relatively straightforward to carry out. This shows that our framework can provide a useful tool for proof planning research.

There are many more experiments that can be performed. Of particular interest would be a more detailed examination of other choices in the implementation of rippling. For instance a case study into the effect of search strategies, which would be difficult in other systems, would be relatively easy in our framework. This is due to the flexible approach to search that we introduced in chapter 5. These are left as further work and we describe some of them in more detail in the following chapter.

Chapter 11

Conclusions and Further Work

We now summarise the work in this thesis, highlighting the main contributions and areas of further work.

11.1 Concluding Remarks

In this thesis we have introduced a new approach to proof planning that represents choices in the search space explicitly. This allows the language for writing techniques to be extensible, where traditional approaches to proof planning have had a fixed language. Another important characteristic of our approach is that it allows technique constructors to be defined in terms of manipulations on the search space. This enables a concise expression of search strategies and techniques that locally specify which search strategy should be used. We have used these features to support efficiency measures which are expressed as simple manipulations of existing techniques.

Our approach to writing techniques is sufficiently expressive to encode proof critics that manipulate the proof plan. Thus it provides a unified language for expressing different kinds of patterns of reasoning. In this respect it provides both a more expressive and simpler language for writing techniques than that used in other proof planning systems.

To show that these features are useful in practice, we provide an implementation of our proof planning framework for the Isabelle proof assistant. We do this by representing proof plans as Isar proof scripts. This allows techniques written in our system to produce human readable as well as fully formal proofs. The success obtained in using this representation shows that it is feasible to write proof planning techniques that produce fully formal proofs.

Because this formality is obtained by executing the proof plan as it is constructed, it allows proof tools in Isabelle to be used during proof planning in an exploratory way. For example, we have made particular use of Isabelle's simplifier, induction tactics, as well as the extended support for fine grained control of equational reasoning

The second branch of work in this thesis is the development of a rich framework for rippling that is suitable for higher order logics. This provides a unified view of various approaches to the technique and makes the proof of its termination a trivial property of its measure. This is in contrast to the rather complex proofs presented by Basin and Walsh [5]. Our analysis also exposes many choices available to the implementation of rippling and shows how they can be examined within our framework.

Within our proof planning machinery, we develop an inductive theorem prover based on our approach to rippling. This allows us to simultaneously evaluate the variations to rippling as well as the suitability of our proof planning framework for encoding complex proof techniques. All the work was done following the Isabelle methodology to implementing generic proof tools. We found this approach to writing generic proof planning tools satisfactory.

To empirically answer the questions raised about the implementation of rippling and to examine the effectiveness of our inductive theorem prover, we carried out experiments in the domains of Peano arithmetic and ordinal arithmetic. These experiments identify the more effective versions of rippling for inductive theorem proving. By comparing our prover with the λ*Clam* system we are able to show that it results in a powerful and efficient inductive proof tool. This also improves the level of automation in Isabelle. The ability to develop effective proof machinery and perform novel experiments shows that our framework provides a practically useful environment for research in proof planning. This work has also allowed us to draw observations to guide further research in proof planning and rippling.

11.2 Further Work

Throughout the thesis we have pointed our various areas of further work. We now summarise the ones we consider particularly important.

Proof Plans

Although the representation of proof plans as Isar proof scripts provides benefits in terms of the intelligibility of the generated proofs and supports the manipulation of the proof plans, we found the representation lacked support for managing meta variables. The inter-proof-script dependencies are implicit which makes a general mechanism for the manipulation of such scripts impossible. Making proof script dependencies explicit be an important step towards the provision of general purpose proof plan transformations. Additionally, incorporation of tools to manage names of parameters and intermediate results would greatly improve the readability of the generated proof scripts.

Improvement for Writing Techniques

We note that the coarse level encapsulation of Isar commands and of contextual information meant that many errors only become apparent at runtime. It would be interesting and practically useful if we could make better use of type checking to catch such errors during the writing of techniques. One such avenue for example, would be to make use of record types to hold contextual information. This would force techniques to declare explicitly the kinds of contextual information they use at the time of writing. It would thus remove the need for a default value for each kind of contextual information and allow type checking to ensure that technique which uses a kind of contextual information cannot be used in a context where it does not exist.

Rippling

The framework we have developed for experimenting with rippling opens up wide array of possible experiments. Of particular interest would be a comparison of simplification with rippling for inductive proof. Such an experiment should be done within the context of a theory development to enable a real analysis of its effectiveness for actually task of formalisation.

We believe that the main way to further improve the efficiency of rippling would be to improve the underlying rewriting machinery to enable to identify when applications of rules are independent. This would allow it to avoid symmetries in the search space which are the main source of inefficiency in our current implementation.

Further Proof Planning Techniques

Many other proof planning techniques have been proposed in the literature and implemented in different systems, such as the derivation of induction schemes during rippling suggested by Gow [44]. Adapting these various techniques to our framework would help identify the relative merits of our proposed approach. It would also require the development of improved machinery. The work of Gow for example, would require support for meta variables in proof plans.

Techniques in the Omega system would be particularly interesting, as we have already shown that the inductive proof techniques can effectively be ported from λ*Clam*. This would help clarify the relationship between encoding techniques in IsaPlanner with Omega.

Additionally, further proof critics could be developed for new domains. For example, it would be interesting to see if proof planning technique can be written to improve the automation in the verification of security protocols. These proof exhibit significant common structure and thus it looks like a fruitful area of research.

Comparing Rippling with Simplification

It seems natural to compare rippling and simplification as both can be used to automate the step case of inductive proofs. However, simply employing simplification instead of rippling and repeating the experiments detailed in this chapter would give misleading results. In particular, it would ignore the simplicity of configuring rippling. Furthermore, at present, only the lemma speculation and generalisation proof critic have been implemented. One of the salient features of rippling its that the annotations can be used to provide a middle out reasoning approach to lemma conjecturing, as suggested by Ireland [52]. As such to get an informative comparison, we would argue that a case study in formalising a new theory should be performed in which proofs are performed using both rippling and simplification. This would give the developer a clear insight into their relative utility. Furthermore, we would also suggest that additional proof critics should be developed for our version(s) of rippling. At present this cannot be done because of the inability of Isar to use meta-variables.

Comparison Inductive Theorem Provers

To compare our inductive theorem prover with other systems would require developing equivalent formalisations in the various systems. In particular, it would be interesting to compare ACL2, Quodlibet [2], Clam, λ*Clam*, INKA, Otter-λ [6, 7], and IsaPlanner. Performing such a comparison would require careful analysis of how each system was setup. When a theory can be more naturally expressed within one system, for example by making use of the more expressive logic, then both formalisations should be examined. Such a comparison would reflect the efficiency of the implementations and thus may obscure the effectiveness of the underlying approaches. However, it would still help to contrast the current state of the art in inductive theorem proving. To survey different theoretical approaches in a fair way, we would suggest comparing them within a similar framework, for example by implementing them within Isa-Planner.

Appendix A

Formalisations of Peano Arithmetic

```
theory N imports Main IsaP begin
datatype N = zero  ("0")
           | suc N ("Suc _" [100] 100)
declare N.inject[wrule]

consts "plus" :: "[N, N] => N" (infixl "+" 65)
consts "mult" :: "[N, N] => N" (infixl "*" 70)
consts "exp"  :: "[N, N] => N" (infixr "^" 80)
end

theory a1 imports N begin
primrec
  add_0[wrule]    :  "0 + y = y"
  add_suc[wrule]  :  "Suc x + y = Suc (x + y)"
end

theory a2 imports N begin
primrec
  add_0[wrule]    :  "x + 0 = x"
  add_suc[wrule]  :  "x + (Suc y) = Suc (x + y)"
end

theory a3 imports N begin
primrec
  add_0[wrule]    :  "0 + y = y"
  add_suc[wrule]  :  "Suc x + y =  x + (Suc y)"
end

theory a4 imports N begin
primrec
  add_0[wrule]    :  "x + 0 = (x :: N)"
  add_suc[wrule]  :  "x + (Suc y) =  (Suc x) + y"
end

theory m1 imports N begin
primrec
  mult_0[wrule]    :  "(x * 0) = (0 :: N)"
  mult_suc[wrule]  :  "x * (Suc y) = x + (x * y)"
end

theory m2 imports N begin
primrec
  mult_0[wrule]    :  "(x * 0) = (0 :: N)"
  mult_suc[wrule]  :  "x * (Suc y) = (x * y) + x"
end

theory m3 imports N begin
primrec
  mult_0[wrule]    :  "(0 * y) = (0 :: N)"
  mult_suc[wrule]  :  "(Suc x) * y = y + (x * y)"
end

theory m4 imports N begin
primrec
  mult_0[wrule]    :  "(0 * y) = (0 :: N)"
  mult_suc[wrule]  :  "(Suc x) * y = (x * y) + y"
end

theory e1 imports N begin
primrec
  exp_0[wrule]   : "x ^ 0 = Suc 0"
  exp_suc[wrule] : "x ^ (Suc y) = x * (x ^ y)"
end
```

```
theory e2 imports N begin
primrec
  exp_0[wrule]   : "x ^ 0 = Suc 0"
  exp_suc[wrule] : "x ^ (Suc y) = (x ^ y) * x"
end
```

General theorems:

```
theorem add_0_left: "0 + a = a"
theorem add_0_right: "a + 0 = a"
theorem add_Suc_right_left: "a + (Suc b) = (Suc a) + b"
theorem add_Suc_left_right: "(Suc a) + b = a + (Suc b)"
theorem add_Suc_right: "a + (Suc b) = Suc (a + b)"
theorem add_Suc_left: "(Suc a) + b = Suc (a + b)"
theorem add_commute: "a + b = b + a"
theorem add_assoc: "b + c + a = b + (c + a)"
theorem add_left_commute: "a + (b + c) = b + (a + c)"
theorem add_right_commute: "(a + b) + c = (a + c) + b"
theorem add_left_cancel: "(k + m = k + n) = (m = n)"
theorem add_right_cancel: "(m + k = n + k) = (m = n)"
theorem mult_0_left: "0 * m = 0"
theorem mult_0_right: "m * 0 = 0"
theorem mult_suc_left: "Suc m * b = b + (m * b)"
theorem mult_suc_right: "m * Suc b = m + (m * b)"
theorem mult_suc_left2: "Suc m * b = (m * b)+ b"
theorem mult_suc_right2: "m * Suc b = (m * b) + m"
theorem mult_1_left: "(Suc 0) * n = n"
theorem mult_1_right: "n * (Suc 0) = n"
theorem mult_commute: "m * n = n * m"
theorem add_mult_dist_right: "(m + n) * k = (m * k) + (n * k)"
theorem add_mult_dist_left: "k * (m + n) = (k * m) + (k * n)"
theorem mult_assoc: "(m * n) * k = m * (n * k)"
theorem mult_left_commute: "x * (y * z) = y * (x * z)"
theorem mult_right_commute: "(x * y) * z = (x * z) * y"
theorem power_squared: "x ^ (Suc (Suc 0)) = x * x"
theorem power_1: "x ^ (Suc 0) = x"
theorem power_add: "i ^ (j + k) = i ^ j * i ^ k"
theorem power_mult: "i ^ (j * k) = (i ^ j) ^ k"
```

Special case theorems:

```
theorem "0 + 0 + a = a"
theorem "a + 0 + a = a + a"
theorem "Suc a + (Suc b) = Suc (Suc (a + b))"
theorem "a + a + b = b + (a + a)"
theorem "d + b + c + a = b + (c + (a + d))"
theorem "(Suc 0 + m = Suc 0 + n) = (m = n)"
theorem "(m + Suc 0 = n + Suc 0) = (m = n)"
theorem "0 * (n + m)= 0"
theorem "0 * m * k = 0"
theorem "(n + m) * 0 = 0"
theorem "(m * Suc b) * k = m * k + (m * b) * k"
theorem "i ^ (j + k + l) = i ^ j * i ^ k * i ^ l"
```

Non-theorems:

```
theorem "0 + a = 0" oops
theorem "a + 0 = 0" oops
theorem "a + b = a + a" oops
theorem "a + (Suc b) = Suc (Suc a + b)" oops
theorem "b + c + a = b + (c + c)" oops
theorem "b + c + a = b + (c + b)" oops
theorem "b + c + a = c + (c + a)" oops
theorem "(k + m = k + n) = (k = n)" oops
theorem "(m + k = n + k) = (k = n)" oops
theorem "0 * m = m" oops
theorem "m * 0 = m" oops
theorem "m * suc b = b + (m * b)" oops
theorem "m * suc b = m * (m + b)" oops
theorem "m * suc b = b * (m + b)" oops
theorem "(Suc 0) * n = Suc 0" oops
theorem "(Suc 0) * n = 0" oops
theorem "(Suc 0) * n = n + n" oops
theorem "n * (Suc 0) = Suc 0" oops
theorem "n * (Suc 0) = 0" oops
theorem "n * (Suc 0) = n + n" oops
theorem "m * n = m * m" oops
theorem "m * n = n + m" oops
theorem "(m + n) * k = (m * k) * (n * k)" oops
theorem "(m + n) * k = (m + k) + (n * k)" oops
theorem "(m + n) * k = (m * k) + (n + k)" oops
theorem "(m + n) * k = (m * n) + (n * k)" oops
theorem "(m + n) * k = (m + k) * (n + k)" oops
theorem "k * (m + n) = (k * m) * (k * n)" oops
theorem "k * (m + n) = (k + m) + (k * n)" oops
theorem "k * (m + n) = (k * m) + (k + n)" oops
theorem "k * (m + n) = (k * n) + (k * n)" oops
theorem "k * (m + n) = (k + m) * (k + n)" oops
theorem "(m * n) * k = m + (n + k)" oops
theorem "(m * n) * k = m * (m * k)" oops
theorem "(m * n) * k = m + (n * k)" oops
theorem "x * (y * z) = y * (y * z)" oops
theorem "x * (y * z) = z * (x * z)" oops
theorem "(x * y) * z = (x * y) * y" oops
theorem "(x * y) * z = (y * z) * y" oops
theorem "x ^ (Suc  0) = x * x" oops
theorem "x ^ (Suc (Suc 0)) = x + x" oops
theorem "x ^ (Suc (Suc 0)) = x * (Suc x)" oops
theorem "x ^ (Suc 0) = 0" oops
theorem "x ^ (Suc 0) = Suc 0" oops
theorem "x ^ 0 = x" oops
theorem "i ^ (j + k) = i ^ j + i ^ k" oops
theorem "i ^ (j + k) = i ^ (j ^ k)" oops
theorem "i ^ (j + k) = i ^ j * j ^ k" oops
theorem "i ^ (j + k) = j ^ k * i ^ k" oops
theorem "i ^ (j + k) = (i ^ j) ^ k" oops
theorem "i ^ (j * k) = (i * j) ^ k" oops
theorem "i ^ (j * k) = (i ^ j) * k" oops
theorem "i ^ (j * k) = (i + j) ^ k" oops
theorem "i ^ (j * k) = (i + j) * k" oops
```

Appendix B

A Formalisation of Ordinal Arithmetic

```
datatype Ord =
    Zero_Ord              ("0")
  | Suc_Ord Ord           ("Suc _" [90] 90)
  | Lim_Ord "nat ⇒ Ord" ("Lim _" [90] 90)

declare Ord.inject[wrule]

consts "plus" :: "[Ord, Ord] => Ord" (infixl "+" 65)
consts "mult" :: "[Ord, Ord] => Ord" (infixl "*" 70)
consts "exp"  :: "[Ord, Ord] => Ord" (infixr "^" 80)

primrec
  ord_add_0     [wrule]: "(x + 0) = x"
  ord_add_Sc    [wrule]: "x + (Suc y) = Suc (x + y)"
  ord_add_Lim   [wrule]: "x + (Lim f) = Lim (λn. x + (f n))"

primrec
  ord_mul_0     [wrule]: "x * 0 = 0"
  ord_mul_Sc    [wrule]: "x * (Suc y) = (x * y) + x"
  ord_mul_Lim   [wrule]: "x * (Lim f) = Lim (λn. x * (f n))"

primrec
  ord_exp_0     [wrule]: "x ^ 0 = Suc 0"
  ord_exp_Sc    [wrule]: "x ^ (Suc y) = (x ^ y) * x"
  ord_exp_Lim   [wrule]: "x ^ (Lim f) = Lim (λn. x ^ (f n))"
```

Bibliography

[1] D. Aspinall and T. Kleymann. *Proof General Manual.* University of Edinburgh, proofgeneral-3.5 edition, 2004.

[2] J. Avenhaus, U. Kühler, T. Schmidt-Samoa, and C. P. Wirth. How to prove inductive theorems? QuodLibet! In Franz Baader, editor, *Proceedings of the 19th International Conference on Automated Deduction (CADE-19)*, number 2741 in Lecture Notes in Artificial Intelligence, pages 328–333. Springer, 2003.

[3] C. Ballarin. Locales and locale expressions in Isabelle/Isar. In *TYPES*, pages 34–50, 2003.

[4] B. Barras, S. Boutin, C. Cornes, J. Courant, J.C. Filliatre, E. Giménez, H. Herbelin, G. Huet, C. Muñoz, C. Murthy, C. Parent, C. Paulin, A. Saïbi, and B. Werner. The Coq Proof Assistant Reference Manual – Version 7.2. Technical Report 0255, Institut National de Recherche en Informatique et en Automatique, February 2002.

[5] D. A. Basin and T. Walsh. A calculus for and termination of rippling. *Journal of Automated Reasoning*, 16(1-2):147–180, 1996.

[6] M. Beeson. A second-order theorem prover applied to circumscription. In R. Goré, A. Leitsch, and T. Nipkow, editors, *IJCAR*, volume 2083 of *Lecture Notes in Computer Science*, pages 318–324. Springer, 2001.

[7] M. Beeson. Lambda logic. In D. A. Basin and M. Rusinowitch, editors, *IJCAR*, volume 3097 of *Lecture Notes in Computer Science*, pages 460–474. Springer, 2004.

[8] C. Benzmüller, L. Cheikhrouhou, D. Fehrer, A. Fiedler, X. Huang, M. Kerber, K. Kohlhase, A. Meier, E. Melis, W. Schaarschmidt, J. Siekmann, and V. Sorge. Ωmega: Towards a mathematical assistant. In W. McCune, editor, *14th International Conference on Automated Deduction*, pages 252–255. Springer-Verlag, 1997.

[9] R. Boulton, K. Slind, A. Bundy, and M. Gordon. An interface between CLAM and HOL. In *TPHOLs'98*, volume 1479 of *LNAI*, pages 87–104, 1998.

[10] R. J. Boulton. Boyer-Moore automation for the HOL system. In L. J. M. Claesen and M. J. C. Gordon, editors, *Higher Order Logic Theorem Proving and its Applications: Proceedings of the IFIP TC10/WG10.2 Workshop*, volume A-20 of *IFIP Transactions*, pages 133–142, Leuven, Belgium, September 1992. North-Holland/Elsevier.

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

[12] R. S. Boyer and J. S. Moore. *A Computational Logic Handbook, (Perspectives in Computing, Vol 23)*. Academic Press Inc, 1988.

[13] J. Brotherston. Cyclic proofs for first-order logic with inductive definitions. In *Proceedings of TABLEAUX 2005*, Lecture Notes in Artificial Intelligence, page 15. Springer, 2005. To appear.

[14] A. Bundy. The use of explicit plans to guide inductive proofs. In *Conference on Automated Deduction*, pages 111–120, 1988.

[15] A. Bundy. A science of reasoning. In *Computational Logic - Essays in Honor of Alan Robinson*, pages 178–198, 1991.

[16] A. Bundy. Proof planning. In B. Drabble, editor, *Proceedings of the 3rd International Conference on AI Planning Systems, (AIPS) 1996*, pages 261–267, 1996.

[17] A. Bundy, D. Basin, D. Hutter, and A. Ireland. *Rippling: Meta-level Guidance for Mathematical Reasoning*. Springer-Verlag, 2005.

[18] A. Bundy, J. Gow, J. Fleuriot, and L. Dixon. Constructing induction rules for deductive synthesis proofs. In S. Allen, J. Crossley, K.K. Lau, and I Poernomo, editors, *Proceedings of the ETAPS-05 Workshop on Constructive Logic for Automated Software Engineering (CLASE-05), Edinburgh*, pages 4–18. LFCS University of Edinburgh, 2005.

[19] A. Bundy, A. Stevens, F. van Harmelen, A. Ireland, and A. Smaill. Rippling: A heuristic for guiding inductive proofs. *Artificial Intelligence*, 62:185–253, 1993.

[20] A. Bundy, F. van Harmelen, J. Hesketh, and A. Smaill. Experiments with proof plans for induction. *Journal of Automated Reasoning*, 7:303–324, 1991.

[21] A. Bundy, F. van Harmelen, C. Horn, and A. Smaill. The Oyster-Clam system. In *10th International Conference on Automated Deduction*, pages 647–648, 1990.

[22] F. Cantu, A. Bundy, A. Smaill, and D. Basin. Experiments in automating hardware verification using inductive proof planning. In *FMCAD96*, volume 1166 of *LNCS*, pages 94–108, 1996.

[23] C. Castellini. *Automated Reasoning in Quantified Modal and Temporal Logics*. PhD thesis, School of Informatics, University of Edinburgh, 2004.

[24] C. Castellini. *Automated Reasoning in Quantified Modal and Temporal Logics*. PhD thesis, University of Edinburgh, 2005.

[25] C. Castellini and A. Smaill. Tactic-based theorem proving in first-order modal and temporal logics. In *Workshop on Issues in the Design and Experimental Evaluation of Systems for Modal and Temporal Logics in proceedings of IJCAR Workshop 10*, International Joint Conference on Automated Reasoning, page 10, 2001.

[26] E. Charniak, C. K. Riesbeck, and D. V. McDermott. *Artificial Intelligence Programming*. Lawrence Erlbaum Associates, 1980.

[27] L. Cheikhrouhou and V. Sorge. PDS — A Three-Dimensional Data Structure for Proof Plans. In *Proceedings of the International Conference on Artificial and Computational Intelligence for Decision, Control and Automation in Engineering and Industrial Applications (ACIDCA'2000)*, page 6, Monastir, Tunisia, 22–24 March 2000.

[28] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(2):56–68, 1940.

[29] A. Cohen, S. H. Murray, M. Pollet, and V. Sorge. Certifying solutions to permutation group problems. In F. Baader, editor, *Proceedings of the 19th International Conference on Automated Deduction (CADE–19)*, volume 2741 of *LNAI*, pages 258–273, Miami, FL, USA, Jul 28–Aug 2 2003. Springer Verlag, Berlin, Germany.

[30] R. L. Constable, F. A. Stuart, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nuprl Development System*. Prentice-Hall, NJ, 1986.

[31] N. G. de Bruijn. AUTOMATH, a Language for Mathematics. In J. Siekmann and G. Wrightson, editors, *Automation of Reasoning 2: Classical Papers on Computational Logic 1967-1970*, pages 159–200. Springer, Berlin, Heidelberg, 1983.

[32] L. Dennis and J. Brotherston. *User/Programmer Manual for the λClam proof planner*. Division of Informatics, University of Edinburgh, Edinburgh, v3.2.0 edition, 2002.

[33] L. Dennis, I. Green, and A. Smaill. Embeddings as a higher-order representation of annotations for rippling. Submitted to JAR, 2005.

[34] L. A. Dennis. *Proof Planning Coinduction*. PhD thesis, Dept. of Artificial Intelligence, University of Edinburgh, 1999.

[35] L. A. Dennis and A. Smaill. Ordinal arithmetic: A case study for rippling in a higher order domain. In *TPHOLs'01*, volume 2152 of *LNCS*, pages 185–200, 2001.

[36] L. Dixon. Interactive and hierarchical tracing of techniques in IsaPlanner. Workshop on User Interfaces For Theorem Provers, 2005, to be published as a volume of ENTCS.

[37] L. Dixon and J. D. Fleuriot. IsaPlanner: A prototype proof planner in Isabelle. In *Conference on Automated Deduction*, volume 2741 of *LNCS*, pages 279–283, 2003.

[38] L. Dixon and J. D. Fleuriot. Higher order rippling in IsaPlanner. In *Theorem Proving in Higher Order Logics*, volume 3223 of *LNCS*, pages 83–98, 2004.

[39] L. Dixon and J. D. Fleuriot. A proof-centric approach to mathematical assistants. *Journal of Applied Logic: Special Issue on Mathematics Assistance Systems*, 2005. To be published.

[40] The DReaM Group. *The Clam proof planner, user manual and programmer manual (version 2.8.1)*, April 1999. Available from ftp://dream.dai.ed.ac.uk/pub/oyster-clam/manual.ps.gz.

[41] J. D. Fleuriot. *A Combination of Geometry Theorem Proving and Nonstandard Analysis, with Application to Newton's Principia*. Springer-Verlag, 2001.

[42] K. Gödel. Über formal unentscheidbare sätze der principia mathematica und verwandter systeme i. *Monatsh. Math. Phys.*, 38:173–198, 1931. English translation in [99].

[43] M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A theorem proving environment for higher order logic.* Cambridge University Press, 1993.

[44] J. Gow. *The Dynamic Creation of Induction Rules Using Proof Planning*. PhD thesis, School of Informatics, University of Edinburgh, 2004.

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

[46] G. Huet. A unification algorithm for typed lambda-calculus. *Journal of Theoretical Computer Science*, 1(1):27–57, 1975.

[47] G. Huet. The zipper. *Journal of Functional Programming*, 7(5):549–554, 1997.

[48] D. Hutter. Annotated reasoning. *Annals of Mathematics and Artificial Intelligence*, 29(1-4):183–222, 2000.

[49] D. Hutter and M. Kohlhase. A colored version of the lambda-calculus. In *CADE'97*, volume 1249 of *LNCS*, pages 291–305, 1997.

[50] D. Hutter and C. Sengler. INKA: the next generation. In M. A. McRobbie and J. K. Slaney, editors, *13th International Conference on Automated Deduction*, pages 288–292. Springer-Verlag, 1996. Springer Lecture Notes in Artificial Intelligence No. 1104.

[51] A. Ireland. The use of planning critics in mechanizing inductive proofs. In *Logic Programming and Automated Reasoning*, pages 178–189, 1992.

[52] A. Ireland and A. Bundy. Productive use of failure in inductive proof. *Journal of Automated Reasoning*, 16(1–2):79–111, 1996.

[53] Isabelle archive of formal proof. http://afp.sourceforge.net/, 2004.

[54] M. Jackson and H. Lowe. System description: Interactive proof critics in xbarnacle. In D. A. McAllester, editor, *CADE*, volume 1831 of *Lecture Notes in Computer Science*, pages 502–506. Springer, 2000.

[55] P. Janičić and A. Bundy. Strict general setting for building-in decision procedures into theorem provers. In *Proceedings of International Joint Conference on Automated Reasoning*, International Joint Conference on Automated Reasoning, pages 18–23, 2001.

[56] F. Kammüller, M. Wenzel, and L. C. Paulson. Locales - a sectioning concept for isabelle. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, editors, *TPHOLs*, volume 1690 of *Lecture Notes in Computer Science*, pages 149–166. Springer, 1999.

[57] M. Kaufmann and J. Moore. An industrial strength theorem prover for a logic based on Common Lisp. *IEEE Transactions on Software Engineering*, 23(4):203–213, April 1997.

[58] M. Kaufmann and J. Strother Moore. ACL2: An industrial strength version of nqthm. In *Compass'96: Eleventh Annual Conference on Computer Assurance*, page 23, Gaithersburg, Maryland, 1996. National Institute of Standards and Technology.

[59] M. Kerber, M. Kohlhase, and V. Sorge. Integrating computer algebra into proof planning. *Journal of Automated Reasoning*, 21(3):327–355, 1998.

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

[61] I. Kraan, D. Basin, and A. Bundy. Middle-out reasoning for synthesis and induction. *Journal of Automated Reasoning*, 16(1–2):113–145, 1996. Also available from Edinburgh as DAI Research Paper 729.

[62] G. Kreisel. Mathematical logic. In T. Saaty, editor, *Lectures on Modern Mathematics*, volume 3, pages 95–195. J. Wiley & Sons, 1965.

[63] J. B. Kruskal. Well-quasi-ordering, the tree theorem, and vazsonyi's conjecture. *Trans. Amer. Math. Soc.*, 95(2):210–225, 1960.

[64] D. Lacey, J. D. C. Richardson, and A. Smaill. Logic program synthesis in a higher order setting. In *Computational Logic*, volume 1861 of *LNCS*, pages 87–100, 2000.

[65] E. Maclean. Generalisation as a critic to the induction strategy. Master's thesis, Dept. of Artificial Intelligence, University of Edinburgh, 1999.

[66] E. Maclean. *Proof Planning Non-Standard Analysis*. PhD thesis, School of Informatics, University of Edinburgh, 2004.

[67] E. Maclean, J. D. Fleuriot, and A. Smaill. Proof-planning non-standard analysis. In *The 7th International Symposium on AI and Mathematics*, 2002.

[68] A. Manning, A. Ireland, and A. Bundy. Increasing the versatility of heuristic based theorem provers. In A. Voronkov, editor, *Logic Programming and Automated Reasoning: Proc. of the 4th International Conference LPAR'93*, pages 194–204. Springer, Berlin, Heidelberg, 1993.

[69] A. Meier, M. Pollet, and V. Sorge. Comparing Approaches to the Exploration of the Domain of Residue Classes. *Journal of Symbolic Computation*, 34(4):287–306, October 2002.

[70] L. I. Meikle and J. D. Fleuriot. Formalizing Hilbert's Grundlagen in Isabelle/Isar. In *Theorem Proving in Higher Order Logics*, pages 319–334, 2003.

[71] E. Melis. AI-techniques in proof planning. In *European Conference on Artificial Intelligence*, pages 494–498, 1998.

[72] E. Melis. The "limit" domain. In *Artificial Intelligence Planning Systems*, pages 199–207, 1998.

[73] E. Melis and A. Meier. Proof planning with multiple strategies. In *Computational Logic*, pages 644–659, 2000.

[74] D. Miller and G. Nadathur. An overview of λProlog. In R. Bowen, K. & Kowalski, editor, *Proceedings of the Fifth International Logic Programming Conference/ Fifth Symposium on Logic Programming*. MIT Press, 1988.

[75] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences.*, 17(3):348–375, 1978.

[76] R. Monroy. *Planning Proofs of Correctness of CCS Systems*. PhD thesis, Department of Artificial Intelligence, University of Edinburgh, 1998.

[77] T. Nipkow. Order-sorted polymorphism in Isabelle. In G. Huet and G. Plotkin, editors, *Logical Environments*, pages 164–188. Cambridge University Press, 1993.

[78] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[79] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS System Guide*. Computer Science Laboratory, SRI International, Menlo Park, CA, September 1999.

[80] L. C. Paulson. Generic automatic proof tools. In Robert Veroff, editor, *Automated Reasoning and Its Applications*. MIT Press, 1997.

[81] L. C. Paulson. A generic tableau prover and its integration with Isabelle. *Journal of Universal Computer Science*, 5(3), 1999.

[82] L. C. Paulson. *The Isabelle Reference Manual*. Computer Laboratory, University of Cambridge, isabelle2004 edition, 2004.

[83] L. C. Paulson. *Isabelle's Logics*. Computer Laboratory, University of Cambridge, isabelle2004 edition, 2004.

[84] L.C. Paulson. *Isabelle: A generic theorem prover*. Springer-Verlag, 1994.

[85] F. Pfenning and C. Elliot. Higher-order abstract syntax. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 199–208, New York, NY, USA, 1988. ACM Press.

[86] F. Pfenning and C. Schürmann. System description: Twelf — A meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206, Trento, Italy, 1999. Springer-Verlag LNAI 1632.

[87] B. Pientka and C. Kreitz. Automating inductive specification proofs. *Fundamenta Informatica*, 39(1-2):189–209, 1999.

[88] R. Pollack. On extensibility of proof checkers. In Dybjer, Nordstrom, and Smith, editors, *Types for Proofs and Programs: International Workshop TYPES'94, Båstad, June 1994, Selected Papers*, volume 996 of *LNCS*, pages 140–161. Springer-Verlag, 1995.

[89] T. M. Rasmussen. An inductive approach to formalizing notions of number theory proofs. In R. J. Boulton and P. B. Jackson, editors, *TPHOLs 2001: Supplemental Proceedings*, number EDI-INF-RR-0046 in Informatics Report Series, pages 328–336. Division of Informatics, University of Edinburgh, Edinburgh, Scotland, UK, September 2001.

[90] J. Richardson and A. Smaill. Continuations of proof strategies. In Tobias Nipkov Rajeev Gorê, Alexander Leitsch, editor, *Proceedings of International Joint Conference on*

Automated Reasoning, International Joint Conference on Automated Reasoning, pages 130–139, 2001.

[91] P. Rudnicki. An overview of the mizar project. In *1992 Workshop on Types for Proofs and Programs*, pages 311–332. Chalmers University of Technology, Bastad, 1992.

[92] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach.* Prentice-Hall, Englewood Cliffs, NJ, 2nd edition edition, 2003.

[93] K. Slind. Derivation and use of induction schemes in higher-order logic. In *TPHOLs'97*, volume 1275 of *LNCS*, pages 275–290, 1997.

[94] K. Slind and R. Boulton. Iterative dialogues and automated proof. In *The Second International Workshop on Frontiers of Combining Systems (FroCos'98)*, volume 7 of *Studies in Logic and Computation*, pages 317–335, 1998.

[95] K. Slind and M. Norrish. The k combinator as a semantically transparent tagging mechanism. In *TPHOLs'02*, volume CP-2002-211736 of *NASA Conference Proceedings*, pages 139–145, 2002.

[96] A. Smaill and I. Green. Higher-order annotated terms for proof search. In *Theorem Proving in Higher Order Logics*, pages 399–413, 1996.

[97] T.F. Melham. The HOL logic extended with quantification over type variables. In L.J.M. Claesen and M.J.C. Gordon, editors, *International Workshop on Higher Order Logic Theorem Proving and its Applications*, pages 3–18, Leuven, Belgium, 1992. North-Holland.

[98] A. M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society (2)*, 42:230–265, 1936-7.

[99] J. van Heijenoort. *From Frege to Gödel: a source book in Mathematical Logic, 1879-1931.* Harvard University Press, Cambridge, Mass, 1967.

[100] C. Walther. Combining induction axioms by machine. In *Proceedings of IJCAI-93*, pages 95–101. International Joint Conference on Artificial Intelligence, 1993.

[101] M. Wenzel. Type classes and overloading in higher-order logic. In *Theorem Proving in Higher Order Logics*, pages 307–322, 1997.

[102] M. Wenzel. Isar - a generic interpretative approach to readable formal proof documents. In *TPHOLs'99*, volume 1690 of *LNCS*, pages 167–184, 1999.

[103] V. Zammit. *On the Readability of Machine Checkable Formal Proofs*. PhD thesis, University of Kent, March 1999.