Automated Discovery of Inductive Lemmas

Moa Johansson

Doctor of Philosophy

Centre for Intelligent Systems and their Applications

School of Informatics

University of Edinburgh

2009

Abstract

The discovery of unknown lemmas, case-splits and other so called eureka steps are challenging problems for automated theorem proving and have generally been assumed to require user intervention. This thesis is mainly concerned with the automated discovery of inductive lemmas. We have explored two approaches based on failure recovery and theory formation, with the aim of improving automation of first- and higher-order inductive proofs in the *IsaPlanner* system.

We have implemented a *lemma speculation critic* which attempts to find a missing lemma using information from a failed proof-attempt. However, we found few proofs for which this critic was applicable and successful. We have also developed a program for inductive theory formation, which we call *IsaCoSy*.

IsaCoSy was evaluated on different inductive theories about natural numbers, lists and binary trees, and found to successfully produce many relevant theorems and lemmas. Using a background theory produced by IsaCoSy, it was possible for IsaPlanner to automatically prove more new theorems than with lemma speculation.

In addition to the lemma discovery techniques, we also implemented an automated technique for case-analysis. This allows IsaPlanner to deal with proofs involving conditionals, expressed as if- or case-statements.

Acknowledgements

I would like to thank my supervisors; Alan Bundy for sharing his knowledge and experience and Lucas Dixon for his enthusiasm, curiosity and willingness to answer my many questions about Isabelle, IsaPlanner, ML and countless other things.

I am also grateful to all the members of the DReaM group for providing a stimulating and creative research environment. Finally, thank you to Omar Montano Rivas and Andrew Priddle-Higson for proof reading parts of this thesis.

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.

(*Moa Johansson*)

Table of Contents

Chapter 1

Introduction

Discovering unknown lemmas and theorems, generalisations, case-splits and other so called eureka steps, are major challenges for automated theorem proving. It has generally been assumed that lemma discovery requires user intervention. Consequently, most theorem provers rely on the user to supply any additional lemmas that might be needed. Interactive theorem provers, such as Isabelle [65], often come with large theory libraries of previously proved lemmas and theorems that are carefully configured and expected to be useful in future proofs.

Approaches to automated lemma discovery can broadly be divided into *lazy* and *eager* techniques. Techniques following a lazy approach attempt to conjecture suitable lemmas when needed in a particular proof attempt. An example of this is *proof critics*, where information from failed proof attempts is used to speculate missing lemmas [41]. In contrast, eager techniques attempt to discover useful lemmas about available functions in advance, thus producing a richer background theory. For a powerful theorem prover, with sophisticated lazy techniques for finding lemmas when needed, it may suffice with a simple background theory. However, a richer background theory makes it possible to find many harder proofs more efficiently, without further lemma discovery techniques.

We have implemented and evaluated one lazy and one eager technique for lemma discovery, with the aim of improving automation of higher-order inductive proofs in the IsaPlanner system [26, 27, 25]. Firstly, we have implemented a *lemma speculation critic*, which, following the lazy approach, tries to find a missing lemma when an inductive proof attempt fails before the inductive hypothesis can be applied. However, lemma speculation turned out to have some serious limitations, and is rarely applicable.

Following the eager approach, we implemented a program for theory formation in

inductive theories, called *IsaCoSy*, which synthesises theorems from recursive datatype- and function definitions.

An automated technique for case-analysis was also implemented, enabling IsaPlanner to deal with proofs involving conditional statements.

The hypothesis we set out to verify can be summarised as follows:

1. Automated lemma discovery techniques can increase the number of inductive proofs found automatically. Lemma speculation is however rarely applicable and is less useful than theorem synthesis, which can tractably produce interesting and useful lemmas.

2. Automated case-analysis techniques enable more theorems to be proved automatically, in particular, theorems involving conditionals.

1.1 Motivation

Inductive proofs are important when reasoning about repetition, for example, recursively defined datatypes and functions. Reasoning about iteration or recursion in computer programs requires induction, which is sometimes needed in program verification. Program verification, is becoming increasingly important as computers become integrated in a vast variety of safety critical systems such as cars, power stations, medical equipment and aeroplanes.

Automating induction is however a difficult task. First of all, as induction is incomplete, there will exist truths that our automated theorem prover will not be able to prove (see §5 of [9] for a discussion). Furthermore, *cut elimination* is not possible in inductive theories. The cut-rule (equation 1.1) is required in inductive proofs to allow introduction of an intermediate lemma or a generalisation. Informally, the cut-rule states that if we can prove a goal Δ from some context Γ, also using a 'lemma', A, and A can itself be proved from Γ, then it is possible to 'cut out' A from the proof of Δ. The proof of A can essentially be included into the proof of Δ:

$$\frac{\Gamma, A \vdash \Delta \qquad \Gamma \vdash A}{\Gamma \vdash \Delta} \tag{1.1}$$

A consequence of the failure of cut elimination is that inductive proofs will sometimes require lemmas that are not already available and cannot be proved without a

nested application of induction. Furthermore, when used backwards, the cut-rule introduces a potential infinite branching point in the search space, as the 'lemma' A can be any formula. To manage these consequences of the cut-rule, we need good heuristic techniques for deciding which lemma is needed and when. Proof-planning critics for lemma discovery are heuristic techniques designed to address these issues. Critics were introduced by Ireland et al. to automatically discover missing lemmas, generalisations and case-splits from information in failed proof attempts [41]. These critics were implemented in the first-order proof-planning system CLAM 3 [13].

An alternative to proof critics is to use automated theory formation techniques to eagerly attempt to find useful lemmas in advance. In inductive theories, this is an equally challenging problem. Not only do we still need to automatically prove theorems, but the program also needs to invent the conjectures themselves. To our knowledge, there is only one system, MATHsAiD [59], that currently is able to perform theory formation for inductive theories.

1.2 Aims of the Project

The aims of the project were twofold. Firstly, we aimed to extend Ireland's idea of proof-critics, in particular for lemma discovery and case-analysis, to higher-order logic in the IsaPlanner system, thus improving automation. Secondly, we wished to contrast lazy lemma discovery by proof critics with eager lemma discovery, using theorem synthesis to discover a set of useful lemmas in advance.

One of IsaPlanner's main weaknesses was its lack of a case-analysis mechanism, which is crucial in many proofs involving functions with conditional definitions. We therefore wanted to implement a case-analysis technique (see Chapter 5), to extend the set of proofs IsaPlanner can deal with automatically. IsaPlanner did already have a simple *lemma calculation critic*, which proves, as lemmas, generalised versions of any sub-goal remaining after the inductive hypothesis has been applied. This works well for many proofs. We implemented and evaluated a more sophisticated *lemma speculation critic* (see Chapter 6). Results were however not encouraging.

We also aimed to build a program that is able to efficiently synthesise interesting and useful theorems and lemmas from recursively defined functions and datatypes, in different inductive theories (see Chapter 7). Synthesising a good background theory can reduce the need for lemma speculation in an automated theorem prover, and may also reduce the workload of a human user developing a new theory.

1.3 Contributions

We will here summarise the main contributions and results of our research.

- We designed and implemented the *IsaCoSy* program, following a novel approach for inductive theory formation by conjecture synthesis. We applied it to different theories about natural numbers, lists and binary trees, including first- and higher-order functions. Using Isabelle's library theories as a reference, IsaCoSy had high recall of 83-100%, if slightly lower precision of 38-63%. We also showed that adding synthesised theorems to our theorem prover reduced the need for lemma speculation.

- We designed, implemented and evaluated a lemma speculation critic for IsaPlanner, extending previous work by Ireland et al. to higher-order logic. We highlighted limitations of lemma speculation not discovered in previous work.

- We designed, implemented and evaluated a novel technique for case-analysis in IsaPlanner, able to deal with if- and case-statements. We showed that it improves on existing techniques, and that it allows IsaPlanner to prove a significant number of theorems that could not previously be automated.

In addition to the above, we have also improved the IsaPlanner system by adding tools for reasoning efficiently about goals with meta-variables, which includes heuristics for restricting higher-order unification during rewriting and resolution.

1.4 Overview of the Thesis

The remainder of the thesis is organised as follows:

Chapter 2: Literature Survey. We give an overview of literature in the domain, in particular proof-planning and inductive theorem proving. We discuss various techniques for dealing with failure in such systems by, for example, introducing missing lemmas or generalisations. We also give a summary of work in the area of automated theorem discovery and theory formation.

Chapter 3: Background. We present some background material including a brief introduction to higher-order unification and an introduction to rippling, as well as overviews of the Isabelle and IsaPlanner systems.

Chapter 4: Reasoning with Meta-Variables. We describe our new techniques for reasoning with meta-variables, *middle-out rewriting* and *restricted resolution*, which are used by the lemma speculation and case-analysis techniques to restrict higher-order unification. We also introduce the concept of *partial wave-rules*, which are used in middle-out rewriting to help manage meta-variable instantiatations.

Chapter 5: Case-Analysis. Our case-analysis technique for IsaPlanner is described and evaluated. We show that the new case-analysis technique enables IsaPlanner to prove a range of new theorems involving conditionals. We also identify some weaknesses in IsaPlanner and suggest improvements to its reasoning about conditions as further work.

Chapter 6: Lemma Speculation. We present our lemma speculation critic for IsaPlanner, extending existing work to higher-order logic. Lemma speculation is also evaluated and the major weaknesses of the technique are identified and discussed.

Chapter 7: Conjecture Synthesis. The implementation of our conjecture synthesis algorithm as the IsaCoSy program is described. We show how a generative synthesis process can become manageable by deducing constraints from known theorems.

Chapter 8: Evaluation of Conjecture Synthesis. We evaluate IsaCoSy's conjecture synthesis machinery on several different theories about natural numbers, lists and trees, and discuss some related work.

Chapter 9: Further Work. We discuss further work in the area of automated inductive theorem and lemma discovery. In particular, we suggest improvements to our conjecture synthesis algorithm.

Chapter 10: Conclusions. We draw conclusions from our results and discuss to what extent the aims of the project have been met, and whether our hypothesis has been confirmed.

Finally, Appendix A contains definitions of functions used in the thesis while Appendix B contains the experimental results for the case-analysis technique. Appendix C contains experimental results for conjecture synthesis.

All techniques described in chapters 4 - 7 of the thesis have been fully implemented in IsaPlanner. The source code is available from the IsaPlanner web-page[1].

1.5 Summary

Improving automation of inductive theorem proving is a challenging but important problem. The main goal of this project is to investigate techniques for automating the discovery of inductive lemmas. We do so by two different techniques, failure recovery and theory formation. We hypothesise that such techniques can increase the domains for which fully automated inductive proofs can be performed.

[1] http://dream.inf.ed.ac.uk/projects/isaplanner/

Chapter 2

Literature Survey

2.1 Introduction

Our work is implemented within the proof-planning approach for automated theorem proving, which we introduce in §2.2. We also give an overview of available proof-planning systems. Rippling is an important heuristic often used with proof-planning for inductive proofs. It is described in §2.3. In §2.4 we introduce the idea of middle-out reasoning, which allows difficult choices in a proof to be postponed. In §2.5, we survey techniques for dealing with common failures in automated inductive proofs. These include for example finding appropriate generalisations or missing lemmas. Similar techniques, applied to reasoning about computer program correctness, are surveyed in §2.6. As a contrast to the techniques that attempt to add missing lemmas when a proof gets stuck, in §2.7, we survey other approaches that instead attempt to form useful background theories directly from initial definitions.

2.2 Proof Planning

Proof-planning was developed by Alan Bundy, motivated by the observation that human mathematicians often first have a high level plan for how to go about solving a proof and then fill in the exact details [8, 12]. The proof-planning technique is used to guide search in automated theorem proving by exploiting the fact that there exist families of proofs with a similar structure. One such family is proofs by induction. An example of a proof-plan is given in figure 2.1, showing how one may go about trying to find a proof by induction, using the rippling heuristic (see §2.3 and §3.4) for the step-case.

A *tactic* is a function that combines several lower level inference rules in a theorem prover to perform some common task. Proof-planners reason about higher-level declarative descriptions of tactics, which, for example, specify when the tactics are applicable. Following the structure of a high-level proof-plan, such as the one for induction in figure 2.1, the proof-planner assembles a tree of tactics that can be executed by a theorem prover to give a fully formal proof.

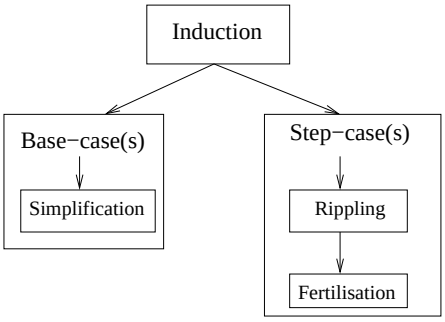

Figure 2.1: Proof-plan for an inductive proof, using the rippling heuristic (see §2.3) to allow the inductive hypothesis to be applied to the step-case goal (called fertilisation).

2.2.1 The CLAM-family

CLAM is a proof-planner written in Prolog developed by Bundy et al. [13]. It originally worked with the Oyster theorem-prover, a Prolog re-implementation of the NuPrl prover [20]. CLAM was later also combined with the HOL prover [4].

CLAM is equipped with a set of *methods* and *methodicals*. Each method has a set of pre-conditions specifying the conditions that must be true for the method to be applicable, and a set of effects that will hold true after the method has been applied. Each method has a corresponding tactic that will be used when the proof-plan is executed. Methodicals can combine several atomic methods into larger compound methods. CLAM was not designed to handle higher-order logic, which motivated the development of the proof-planner λClam [69]. λClam was written in λ-Prolog, which is a higher-order version of the Prolog language. In addition to inductive proofs, λClam has also been applied to proof-planning in non-standard analysis [56], and combined with an object level prover for first-order temporal logic to plan proofs in this domain [15].

The proof-planners in the CLAM-family are no longer actively developed.

2.2.2 INKA

The INKA system is a first-order prover designed for reasoning about program verification developed by Dieter Hutter et al. [38]. It provides support for inductive proofs by rippling. In addition to its automatic reasoning capabilities, INKA also allows for user interaction, where the user can guide the proof at varying levels of detail. INKA is no longer actively maintained, although aspects of INKA are now instead included in the formal verification system VSE [37].

2.2.3 Ωmega

The Ωmega system has been developed by the research group with the same name at the Saarland University [3]. It takes a different, knowledge-based, approach to proof-planning. Ωmega controls proof-planning by a set of high-level control-rules that selects between different proof-planning methods. The control-rules encode heuristic knowledge for a specific domain narrowing down the number of methods that may be applied. Methods then specify the exact conditions under which they are applicable. Within Ωmega, proof-planning has been successfully applied to, amongst other areas, the domain of limit theorems [62].

Erica Melis and Andreas Meier extended proof-planning in Ωmega, adding yet another layer of control, *strategies*, in the MULTI proof-planner [61, 60]. Each strategy represents an instance of a problem solving algorithm, for example rippling. The strategy layer allows the planner to let several proof-planning strategies cooperate when proving a theorem. The planner uses a black-board architecture where strategies advertise their applicability to the current goals. Control knowledge is used to determine which strategy should be invoked.

2.2.4 IsaPlanner

Lucas Dixon originally developed the IsaPlanner system, [26, 25], a proof-planner for the interactive theorem-prover Isabelle [65]. IsaPlanner interleaves proof-planning with the execution of the proof in Isabelle, allowing access to Isabelle's powerful tactics. Proofs are planned through a series of *reasoning states*, each containing the partial plan constructed so far, the next reasoning technique to be applied and contextual information. The contextual information contains knowledge acquired during the planning process, such as annotations for rippling, information about proved lemmas etc.

2.3 Rippling

Rippling is a heuristic used to guide rewriting of the step-case in inductive proofs [10]. Rippling has however been shown to be applicable to other areas by a range of authors, for example summing series [73], equation solving [36], and non-standard analysis [56]. Within the context of proof-planning, rippling has successfully been used for automating proofs in both hardware [14] and software verification [43], as well as in the synthesis of higher-order programs [52]. Common for these domains is that we have some *given* and some *goal* that is to be rewritten in such a way that the differences between the given and goal are reduced. The aim is to arrive at a situation where the goal can be justified by the given. This is called *fertilisation*. If the goal matches the given exactly, the given can be applied directly to conclude the proof, which is referred to as *strong fertilisation*. If the given is an equation, it may be possible to apply it as an extra rewrite rule even if it does not match the goal exactly. This is called *weak fertilisation*.

Rippling guides the rewriting process by identifying and annotating the parts of the goal that are similar to the given and should be preserved, called the *skeleton*, and the parts that are different and need to be moved out of the way before fertilisation, called the *wave-fronts*. A *wave-hole* denotes a sub-term inside a wave-front that is part of the skeleton. Positions corresponding to universally quantified variables in the given, which can be instantiated during fertilisation, are called *sinks*. An example of an annotated step-case goal of an inductive proof is shown below:

Given (inductive hypothesis): $\forall b'.\ a + b' = b' + a$

Goal (step-case goal): $\mathit{Suc}\ a^{\uparrow} + \lfloor b \rfloor = \lfloor b \rfloor + (\mathit{Suc}\ a)^{\uparrow}$

The wave-fronts are represented by shaded boxes. Note that the position of the universally quantified variable b' becomes a sink in the goal, annotated by $\lfloor b \rfloor$. When a goal can be annotated with respect to a skeleton, we sometimes refer to the skeleton as having an *embedding* into the goal.

Rippling can reduce the differences in two ways, the arrows on the wave-fronts indicate if they are to be *rippled-out* ($^{\uparrow}$) or *rippled-in* ($^{\downarrow}$). Rippling-out tries to move the wave-fronts towards the top of the term-tree until the goal contains a sub-term matching the given. Rippling-in attempts to move wave-fronts to a position of a sink, which corresponds to a universally quantified variable in the given. Differences in sinks can instantiate such variables, allowing fertilisation. A wave-front is only allowed to be

rippled-in if, within its wave-hole, there is a sink to eventually unify with, or an outwards directed wave-front to potentially cancel it.

After the initial goal has been annotated rippling proceeds by applying rewrite-rules derived from function definitions, axioms and existing theorems and lemmas. These rules are referred to as *wave-rules*. A *ripple measure* is a well-founded order on annotated terms, based on the positions of wave-fronts in the goal. Each wave-rule application is required to decrease the ripple measure, which ensures the termination of rippling. New goals that cannot be annotated with respect to the skeleton are typically discarded. When no more measure-decreasing rewrites are possible, either fertilisation is possible, or rippling is said to be *blocked*.

The ripple measure allows rippling to apply equations in either direction. This makes rippling more flexible and easier to configure than conventional rewriting when the direction of equational rules must be specified.

A worked example illustrating the different features of rippling in more detail can be found in §3.4.

Dynamic Rippling

In the traditional account of rippling, *static rippling*, as described in [10], terms are annotated at the object level, and rewrite rules annotated prior to rippling. In dynamic rippling on the other hand, annotations are kept separate from the goal, and recomputed after each application of a rewrite-rule. Rewrite-rules themselves are not annotated.

Dynamic rippling was first suggested by Alan Bundy and David Basin[1] as a way to avoid having to compute and store all the possible annotated versions of each rewrite rule. They propose that wave-rules should instead be annotated dynamically, as they are needed. Another disadvantage of static rippling is that its object-level annotations require a special notion of substitution, otherwise illegal annotations may be produced [2]. In dynamic rippling, annotations can be stored separately from the goal, as *term-embeddings*, introduced by Alan Smaill and Ian Green [70]. As a result, no special notion of substitution is needed.

Dennis, Smaill and Green also identified that dynamic rippling is required for rippling in higher-order domains, as the object-level annotations of static rippling are not stable over β-reduction and may introduce incorrect annotations in the presence of meta-variables [70, 23]. Their version of dynamic rippling was implemented in the

[1] Personal communication. University of Edinburgh, Mathematical Reasoning Group, internal Blue Book Notes 919 and 920

λClam system [23]. Lucas Dixon later implemented a considerably faster version in IsaPlanner [27, 25].

2.4 Middle-Out Reasoning

The central idea of middle-out reasoning is to postpone difficult decisions in a proof, instead starting from the middle and expecting the simpler parts of the proof to suggest solutions for more difficult steps. Middle-out reasoning was first suggested by Alan Bundy et al. as a technique for handling Eureka steps, such as finding an existential witness in program synthesis [11].

2.4.1 Middle-Out Reasoning and Tail Recursive Functions

Jane Hesketh implemented middle-out reasoning techniques in the CLAM system for her PhD [30]. She applied middle-out reasoning to the task of generating tail-recursive function definitions from naive ones [31]. The naive definition is taken as a specification for the synthesis of the tail-recursive definition. As mentioned in §2.2.1, the CLAM system works with the Oyster theorem prover. Oyster uses a constructive logic where proof steps correspond to steps in a program. Hesketh's technique attempts to generate the tail recursive definition by wrapping a meta-variable around the naive version together with a new universally quantified variable (which is to become the accumulator). Middle-out reasoning is then used to attempt to instantiate the meta-variable during the subsequent rippling proof.

Hesketh also applied middle-out reasoning to find generalisations for failed inductive proofs. For example, conjectures about tail-recursive functions may need to be generalised to insert a variable in the accumulator position in order for the inductive hypothesis to be applicable. These ideas were further developed by Andrew Ireland et al. into a *generalisation critic* [41], described in §2.5.1.

2.4.2 The Periwinkle system

Ina Kraan implemented middle-out reasoning techniques for synthesis of logic programs from given specifications in the Periwinkle system [51]. The body of the program is here initially represented by a meta-variable and then instantiated during proof-planning. Middle-out reasoning was also used to solve the problem of choosing an

induction scheme during synthesis. The induction scheme corresponds to the recursive structure of the program, which is unknown at this point. The system applies a schematic induction scheme, which is instantiated during the step-case proof. The instantiated induction scheme is then checked against a pre-computed set of known schemes to ensure it is valid.

2.4.3 The Dynamis system

Jeremy Gow further developed the idea of synthesising induction schemes by middle-out reasoning in an extension of λClam, called *Dynamis* [29]. As in Periwinkle, a schematic induction scheme is applied, and the proof of the step-case instantiates the meta-variables. Dynamis can deal with both constructor- and destructor style induction schemes, unlike Periwinkle. After an instantiated rule has been found, Dynamis proves whether or not the instantiated rule is a valid induction rule. The system was evaluated on a set of theorems where common recursion analysis did not suggest a successful induction scheme. It successfully found valid induction schemes for half of the 38 test cases.

2.5 Failure Reasoning in Inductive Theorem Proving

2.5.1 Critics in CLAM

Proof-planning sometimes fails, but a failed proof-plan may still contain useful information about how the proof could be patched. Critics make use of this information to try to find a suitable patch that will allow the proof to continue [39, 41]. Critics are typically attached to a proof-planning method and fired when that method fails in a particular way.

2.5.1.1 Critics for Rippling

Ireland et al. present four critics; induction scheme revision, lemma discovery, generalisation and case-splitting, each triggered by different ways the rippling method might fail [41]. The critics were implemented in version 3 of the CLAM proof-planner. With their help, CLAM was shown to be able to fully automatically prove a range of conjectures that would otherwise fail or require user-intervention.

Induction scheme revision: The induction scheme revision critic can repair proof attempts where it is necessary to use a different induction scheme than the one initially chosen, for example two-step induction instead of the standard one-step scheme. The critic is fired when no more wave-rules apply to the current goal, but it can identify a partial match, where the skeletons of some wave-rule and the goal match, but some wave-front in the rule is missing from the goal. The critic attempts to trace this missing wave-front back to the induction variable to suggest another induction scheme.

Case-splits: In the presence of conditional wave-rules, it might be the case that rippling fails to apply because a condition associated with a wave-rule cannot be proved. The case-splitting critic performs a case-analysis, and suggests a case-split on the associated condition. The resulting goals are then proved separately.

Lemma discovery: There are two methods for lemma discovery. *Lemma calculation* applies when no wave-rules are applicable and one side of an equational conjecture is fully rippled. A potential equational lemma is constructed with the blocked term as the left hand side and the term resulting from weak-fertilisation as the right hand side. The result is then generalised, using common sub-term generalisation, and proved. *Lemma speculation* is applied when weak-fertilisation is not applicable. The right hand side of the lemma is constructed by inserting second-order meta-variables into the skeleton-term of the goal. Several options of where to insert them exist, corresponding to outward, inward or sideways wave-rules. After the schematic lemma has been applied to the blocked goal, the meta-variables are instantiated through *middle-out reasoning* [11, 30], where further applications of wave-rules provide instantiations for the meta-variables of the schematic goal (also shared by the schematic lemma). After each step, an attempt is made to coerce the remaining meta-variables by exploring their possible projections. A counter-example checker is employed to filter out non-theorems, after which any remaining candidate lemmas are passed on to the prover.

Generalisation to introduce accumulator variables: Generalisation may sometimes be needed to strengthen the inductive hypothesis before the proof can go through. The generalisation critic in CLAM 3 deals with one such example: generalisation to introduce accumulator variables. When rippling-in fails to apply due to a missing sink, the generalisation critic is fired. The goal is generalised by introducing second-order meta-variables in positions of potential accumulator vari-

ables. These are then instantiated through subsequent applications of wave-rules followed by attempts to coerce the meta-variables, similarly to lemma speculation. The generalisation critic was later extended to deal with the introduction of multiple accumulator variables and to handle the introduction of auxiliary accumulators in arbitrary positions [40].

The lemma speculation and generalisation critics search for instantiations of the meta-variables by applying wave-rules to the schematic terms constructed. This requires the use of higher-order unification, potentially giving rise to a large number of possible unifiers. The CLAM 3 critics use the rippling annotations to help control higher-order unification by dividing it into sub-tasks. When rippling-in, the wave-holes in the goal and the rule are matched first, before unification is attempted. For rippling-out, the super-terms containing the wave-front are pre-matched. However, in some cases, it is still necessary to backtrack over different possibilities.

2.5.1.2 Generalisation Critics for the Induction Method

For his MSc dissertation Ewen Maclean investigated attaching generalisation critics to the induction method of CLAM, thereby trying to apply generalisations before rippling was attempted [55]. Critics for three types of generalisation are implemented: replacement of minimal common sub-terms, generalising variables apart and replacement of independent sub-terms. These techniques, alongside other generalisation heuristics for inductive proofs are also described in a survey by Birgit Hummel [35].

Attaching generalisation critics to the induction method turns out to be problematic, as the critics often either fail to fire without further look-ahead into the proof-plan, or produce over-generalisations. Instead, Maclean suggests attaching the generalisation critics to the rippling method, allowing them to fire when rippling is blocked. A critic for independent sub-term generalisation was implemented as a ripple-method critic and shown to find more generalisations than the equivalent critic attached to the induction method.

2.5.1.3 Interactive critics

Interactive versions of the rippling critics from CLAM 3 have been implemented in the XBarnacle system, a graphical front-end to CLAM [54, 44, 45]. Critics increase the size of the search space: at the point of failure more than one critic may be applicable

and each critic may suggest several patch schemes. In addition, the patch applied may fail and require backtracking or application of further critics.

Interactive critics also give a skilled user the chance to aid the proof-planner to solve conjectures where the critics in CLAM 3 alone would fail. An example is any proof where a generalisation is required but a lemma needed in the process is missing (see [44] §5). It is also argued that interactive critics give the user a chance to produce a shorter proof by instantiating meta-variables, thus relieving the proof-planner of time-consuming middle-out reasoning.

Michael Jackson explores the use of interactive critics in his PhD thesis [45]. Two approaches are evaluated, *active* and *passive* interactive critics. Active critics are fired when the associated method fails as usual. The user is presented with potential lemmas and generalisations containing meta-variables that can be instantiated or left to the planner. There is also an option to view an explanation of why the proof-planning method failed and why a particular critic was invoked. Motivated by user comments, a passive version was developed, where the user decided when a critic should be applied and to which goal. Evaluation suggested that passive interactive critics were preferred by users.

2.5.1.4 Critics for Patching Faulty Conjectures

Raul Monroy-Borja developed a set of critics for patching faulty conjectures in CLAM 3 for his MSc project [63]. The critics attempt to derive antecedents for the faulty conjecture that will turn it into a theorem, or correct arguments that are in the wrong positions. The *contradictory blocked goals* critic deals with failure in the base-case and is fired when an induction attempt leads to an obvious contradiction, such as $0 \neq 0$. The associated patch attaches the negation of the base-case of the most recent induction attempt as a condition to the original conjecture. If this patch fails, another critic is fired, attempting a patch that adds the whole of the current blocked sub-goal as a new condition for the conjecture.

Sometimes the conjecture may need further refinement after the base-case has been patched. If the hypothesis and the step-case goal match modulo antecedents and it can be shown that the antecedent of the hypothesis logically implies that of the conclusion, a method called *conditional fertilisation* is applied to conclude the proof. Should this fail, a fertilisation critic is fired, looking for a suitable function to replace the antecedent of the conjecture (§3.4 of [63] gives details of how such a function is selected).

For some faulty conjectures the base-case may succeed, with the blockage arising

in the step-case. If a counter-example for the blocked goal can be found, the *partial success critic* is invoked. This critic uses the successful instantiation of the induction-variable from the base-case to construct a condition that is added to the conjecture to make it into a theorem. Another critic is fired if a counter-example can be found for the current sub-goal, with lemma calculation simultaneously being applicable. The critic then attempts to re-arrange the arguments in the conjectured lemma, rejecting combinations for which a counter-example can be found.

2.5.2 Generalisation in INKA

Birgit Hummel implemented a range of generalisation techniques in the INKA system for her PhD. These include replacement of common sub-terms with new variables and replacement of independent sub-terms. Hummel also includes the technique of applying the inductive hypothesis as a rewrite rule as one of the generalisation techniques, which is otherwise commonly refereed to as weak fertilisation. We are unable to review this work in detail as her thesis is written in German. A detailed survey of the techniques can however be found in [35].

2.5.3 Failure reasoning in Ωmega

Andreas Meier developed some methods for failure reasoning in proofs of limit theorems in the MULTI proof-planner, which is part of the Ωmega system [60] (chapter 8). Case-splitting is triggered under similar circumstances as in CLAM. A form of lemma speculation is triggered when unification or matching fails, leaving some residue. The residual term is given to a constraint solver to determine if it is promising to prove the term as a lemma and if so, attempts to provide instantiations of meta-variables. MULTI also supports another form of failure reasoning, called goal-directed backtracking. This allows backtracking to any point in the proof-plan, without undoing all previous work. When a highly desirable strategy is blocked, backtracking is directed towards areas in the proof-plan that are likely to help unblock the strategy. For example, if instantiations for some necessary meta-variable cannot be found due to insufficient constraints, backtracking is directed towards further refining the complex inequalities the constraints are based on.

2.5.4 Critics in IsaPlanner

The first version of the IsaPlanner system only supported one critic for lemma calculation [25] (section 9.9), [27]. If a non-trivial goal remains after weak-fertilisation, a lemma is constructed by applying common sub-term generalisation to the goal. Generalisations are performed such that the largest common sub-terms are replaced by variables, motivated by empirical results. This also cuts down the search space compared to picking the smallest common sub-term. IsaPlanner may also apply argument congruence rules, which allows function symbols appearing on the top-level of both the left- and right hand side of an equation to be dropped.

After generalisation, a separate proof-attempt of the resulting lemma is launched which, if successful, allows the proof to continue. As a consequence of the lack of critics, IsaPlanner was previously unable to prove many of the conjectures provable by CLAM 3.

2.5.5 Generalisation in VeriFun

VeriFun is a semi-automatic verification tool for functional programs with an automatic tactic for inductive proofs in first-order logic, originally developed by Christoph Walther [74]. Markus Alderhold has extended VeriFun with a number of standard generalisation techniques with sophisticated heuristics, for deciding when and how the generalisations are applied, along with a counter-example finder to avoid over-generalisations [1]. The generalisation techniques can either be invoked by the user or automatically when VeriFun's verification tactic has failed. The current goal is generalised to yield a lemma that will help solve the proof. This is similar to the lemma calculation critics of CLAM and IsaPlanner but with a wider selection of generalisation techniques to choose from.

VeriFun supports the following techniques for generalisation:

Selector Elimination: The destructor style induction of VeriFun often leaves many instances of selector terms (such as *head* and *tail* functions on lists). These are replaced by fresh variables.

Common non-variable sub-term generalisation: VeriFun does not commit to generalising the largest or smallest common sub-term, but instead tries to replace a sub-term that occurs in a recursion position of some function. Inducting on

a variable in such a position is beneficial, as subsequent rewrite rules about the function are more likely to apply.

Generalising variables apart: VeriFun attempts to separate occurrences of a variable that occurs both in a recursion position and in a non-recursion position to facilitate an inductive proof attempt.

Inverse Weakening: Destructor style induction will sometimes leave unnecessary conditions which the Inverse Weakening generalisation technique can identify and remove.

Inverse Functionality: If both sides of an equation have the same top-level function symbol, this can sometimes be dropped and replaced by an equality between the arguments (also known as argument congruence). VeriFun combines this type of generalisation with counter-example checking to ensure no over-generalisation is found.

The generalisation heuristics of VeriFun have been evaluated on a range of problems from the CLAM 3 corpus (see [41]) and from verification of sorting algorithms [1]. The generalisation techniques allow many of these problems to be solved while rarely suggesting over-generalisations that have to be discarded by counter-example checking.

2.5.6 Lemma Discovery in RRL

Algorithms for lemma discovery and generalisation of accumulator variables in equational inductive proofs, similar to the critics in CLAM, have also been proposed for the *Rewrite Rule Laboratory* (RRL) system [48, 47].

As with the lemma speculation critic in CLAM (§2.5.1), the technique proposed for RRL is applied in an inductive proof when it is not possible to either rewrite the step-case goal or apply the hypothesis. It first generates a set of equations, by equating sub-terms in the step-case goal with the left- or right hand side of the hypothesis. These equations are then simplified and meta-variables inserted (here referred to as *instantiation schemes*) in positions of non-induction variables[2]. The next step of the algorithm attempts to generate constraints on the instantiations of the meta-variable by applying rewrite rules that remove the term-context surrounding it. These constraints are then used to speculate instantiations.

[2]These are the same as positions of sinks in rippling.

A method for relating conjectures about tail-recursive and non-tail recursive functions by generalisation of an accumulator variable is proposed in [48] and further refined in [47]. This method initially proceeds in the same fashion as the critic in CLAM (§2.5.1), by replacing a constant, assumed to be in the position of an accumulator variable, on one side of an equational goal with a variable. The other side of the goal is transformed by introducing a top-level meta-variable that also takes the new variable as an argument. A new induction is then attempted on this schematic conjecture. As above, a set of constraints is generated to help suggest instantiations. This method can also provide counter-examples for invalid conjectures.

To our knowledge, no experimental results for the above methods have been published, so it is not clear whether these algorithms were ever fully implemented in RRL.

2.5.7 Critics Detecting Divergence

Toby Walsh has implemented critics to detect divergence in inductive proof attempts in the theorem prover SPIKE [72], where unsolvable problems often led to divergence. The divergence critic studies the proof attempt in order to detect patterns of divergence, such as accumulating term structure, by difference matching. The critic attempts to suggest lemmas that are speculated by using heuristics enabling cancellation of term structure that would otherwise accumulate and cause divergence. Potential lemmas are filtered through a type-checker and a conjecture disprover, then generalised and proved.

Louise Dennis et al. describes a divergence critic in CLAM designed to find generalisations when searching for bi-simulations in co-inductive proofs [22], used to reason about observational equivalence of functional programs. A bi-simulation is a relation containing observationally equivalent pairs. If a co-inductive proof-planning attempt fails, a patch is attempted that introduces another bi-simulation. Subsequent introductions of new bi-simulations may however lead to divergence. The divergence critic can identify accumulating term structure caused by divergence and instead suggest a generalisation.

2.6 Critics for Program Reasoning

Proof-planning and critics have also been applied to the domain of program verification of SPARK programs within the NuSPADE system. Andrew Ireland, Bill Ellis and

their colleagues have developed proof plans and associated critics for *exception freedom proofs* [42], and for *partial correctness proofs* [43]. Some of the critics within NuSPADE have been given an extended role, not just patching proof attempts but also acting as an interface between proof-planning and the generation of program properties, such as the discovery of a loop invariant.

Exception freedom proofs show that a program is free of run-time exceptions. The proof-plan is used for showing that a variable does not exceed its legal bounds, causing a buffer under- or overflow. Four critics were developed:

- The *elementary critic* searches for counter-examples for variables in exception freedom goals that cannot be immediately discharged by the simplifier. If a counter-example is found, it is used to guide the search for tighter bounds on variables.

- The *transitivity critic* describes missing proof context and helps guiding the generation of additional properties for the program specification.

- The *decomposition critic* is given a slightly different role compared to traditional critics, and simply flags problems that may relate to coding defects to the user.

- The *fertilisation critic* applies to failure of the fertilisation method, and fires when a hypothesis is weaker than the goal but still similar. The critic attempts to infer a stronger hypothesis from the weaker one.

Partial correctness proofs are concerned with the functional correctness of programs. The work in [43] focuses on array-based programs and makes use of rippling for reasoning about loop invariants. Two critics were developed. The *range generalisation critic* fires when a proof attempt of a transitive relation between adjacent array elements fails. The critic attempts to patch the proof by generalising to consider a range of elements, representing finding an auxiliary loop invariant. The *difference generalisation critic* is associated with the ripple method when a weak-fertilised goal requires further rippling towards another hypothesis (here, an invariant).

2.7 Theorem Discovery

To our knowledge, there are very few systems that are able to automatically discover inductive theorems. Other than the proof-planning critics described above [41, 27],

which use information from failed proof attempts, the MATHsAiD system [58, 59], is the only dedicated theory formation system which has been applied to the task. We will however also survey a few other systems for theory formation.

2.7.1 The AM system

Doug Lenat's AM system was one of the earliest theory formation systems [53]. Although no longer in use, we shall summarise its main features.

The AM system was equipped with an initial set of 115 basic concepts in set-theory, including set equality and common operations such as deletion. It also had a fairly large number of heuristic rules (about 250) for deducing new facts and concepts from known ones, as well as restricting search. AM did not, however, have the capability to prove any conjectures.

From its initial concepts and heuristics, AM managed to build theories about, for example, natural numbers, with addition, multiplication and exponentiation. It also went on to invent more advanced concepts such as prime numbers, and managed to conjecture Goldbach's conjecture (every even number greater than 2 is the sum of two primes).

2.7.2 HR

HR is a theory formation system in pure mathematics, originally developed by Simon Colton for his PhD [18]. HR used a set of seven production rules to derive new concepts from a small set of initial concepts. For natural numbers, such concepts include multiplication, addition and divisors. HR uses the resolution prover Otter to prove conjectures it has created, and the MACE model generator for counter-examples. HR has been applied to domains including number theory and graph theory. Although the number of interesting conjectures made was rather low, HR managed to, for example, invent some novel integer sequences. HR has also been used to generate new problems in group theory for the TPTP-library of challenge problems for (mainly classical first-order) automated theorem provers [19, 71].

2.7.3 MATHsAiD

MATHsAiD is a recent system for theory formation [58], implemented by Roy McCasland. Its aim is to generate theorems that a human mathematician would consider

interesting, for example, theorems that occur in mathematics textbooks. MATHsAiD has been successfully applied to discover inductive theorems about natural numbers [59], as well as theorems in other domains such as basic group theory. As we are mainly interested in inductive theorems, we will here summarise how MATsAiD discovers inductive theorems. It also has a similar procedure for non-inductive theorems.

When asked to discover inductive theorems, MATHsAiD first automatically generates a set of left-hand sides of potential equations. It then replaces a variable in the term, with 'TWO', where the identity of 'TWO' depends on the type of interest, e.g. $Suc(Suc\ 0)$ for natural numbers, or a list of length 2. MATHsAiD then applies forward chaining on these terms, using available theorems, to generate corresponding potential right hand sides. A candidate right hand side is some new term, also containing 'TWO'. If the new equality holds for 'TWO', a full inductive proof is attempted. MATHsAiD applies structural induction, followed by what is referred to as *piecewise search*, where the left- and right hand sides of the step-case goal are rewritten to match the corresponding side of the inductive hypothesis, using available definitions and theorems. MATHsAiD's inductive prover does not attempt to discover missing lemmas, as the proof-planners discussed above, but may return to failed proofs after additional theorems have been generated and proved.

Given some basic axioms defining the natural numbers with addition and multiplication, as well as the concepts of commutativity, associativity and distributivity, MATHsAiD was able to discover the standard theorems about commutativity, associativity and distributivity of addition and multiplication in less than two minutes [59].

2.7.4 The AGInT System

AGInT is a theorem discovery system for first-order classical logic, implemented by Yuri Puzis et al. [68]. It uses an automated first-order prover to generate logical consequences of a set of axioms. This produces a large set of statements, many of which are trivial or otherwise not considered interesting. The system then applies a set of filters and ranking scores in order to identify the interesting theorems. AGInT has been tested on axioms about set theory and about logical puzzles from the TPTP library [71], where it finds some theorems.

2.7.5 Scheme-Based Theory Exploration

Bruno Buchberger's research group has proposed a model for theory exploration based on *knowledge schemes* for the Theorema system [6]. The knowledge schemes are supposed to capture prior mathematical knowledge, for example describing the basics of structures such as groups, or ordering relations. The schemes can then be instantiated with symbols in the current theory to produce new concepts or function definitions. There are also some theory specific schemes, capturing, for example, structural induction over the natural numbers. A preliminary case-study of the natural numbers has been undertaken, but the process is not yet automated [32].

2.8 Summary

Our work on automated lemma discovery largely takes place within the context of proof-planning, which is an approach for automated theorem proving exploiting the fact that certain classes of proofs, such as proofs by induction, have a common structure. Rippling is a heuristic commonly used for inductive proofs in proof-planning. Proof critics were developed to deal with common failures during proof-planning, including the discovery of missing lemmas or the discovery of more general theorems, by which a more specific and difficult theorem can be proved. We also discussed a range of other failure reasoning techniques for inductive proofs in different systems. In addition to lemma discovery by proof critics, our work is also concerned with inductive theorem discovery by synthesising conjectures from available function definitions. We surveyed systems for theorem discovery, but only the MATHsAiD system has, to our knowledge, been applied to inductive theories.

Chapter 3

Background

3.1 Notation

Below, we introduce some notational conventions that are used throughout the thesis:

- We will follow Isabelle's convention and write theorems with assumptions separated from the conclusion using $\Longrightarrow$[1]. If there are several assumptions these are enclosed in square brackets, e.g. $[\![P;\ Q]\!] \Longrightarrow R$ states that P and Q are assumptions for the conclusion R.

- We differentiate between two types of variables, bound variables and free variables. Bound variables are variables bound by some lambda-abstraction, e.g. x in $\lambda\, x.\ n\ + x$. Free variables represent arbitrary values. Free variables that are allowed to be instantiated by unification are referred to as *meta-variables* (also sometimes called *schematic variables*). Meta-variables are prefixed by '?', e.g. $?F$, following Isabelle's conventions.

- Goals that contain meta-variables are sometimes refereed to as *schematic goals*.

- An infix function (e.g. +) applied to only one of its two arguments is written as a lambda abstraction, e.g. $\lambda\, x.\ n\ +\ x$.

- The '@'-symbol denotes the list append function. '#' denotes cons. All functions used in examples are defined in Appendix A.

[1] This is Isabelle's meta-level implication

3.2 Rules of Lambda Calculus

Lambda calculus is a model of computation and a commonly used logical system for reasoning about functions, introduced by Alonzo Church [17]. Lambda calculus also forms the basis for functional programming languages. There are several variants of lambda calculus, for example typed and untyped. Isabelle's higher-order logic is a simply typed lambda calculus.

To reason about function applications, lambda calculus provides three important inference rules, α- β- and η conversions. We let the symbol $\mapsto$ denote substitution and use the notation $x \notin E$ to mean that the variable x does not occur in the expression E.

α-conversion:

$$\frac{\lambda x.\ E}{\lambda y.\ E[x \mapsto y]} \quad y \notin E$$

We can replace any variable x with a fresh variable y of the same type. This rule is used to avoid variable name capture in substitution.

β-conversion:

$$\frac{(\lambda x.\ E)\ t}{E[x \mapsto t]}$$

This rule is concerned with the application of a function to an argument; every occurrence of the bound variable x is replaced by the argument t in the body of the lambda-abstraction.

η-conversion:

$$\frac{\lambda x.\ E}{E} \quad x \notin E$$

If the bound variable x does not occur anywhere in E, the expression can be replaced by E alone.

Terms can be transformed into various normal forms using these rules, for example a β-normal term has been reduced using β-conversion so that no applications remain. IsaPlanner puts terms in βη-normal form during rippling.

3.3 Higher-Order Unification

Higher-order unification is concerned with the problem of making two terms equivalent in typed lambda-calculus. Higher-order unification is semi-decidable, it is possible to build algorithms that return a solution if one exists, but may not terminate if one does

not. Gerard Huet proved this and designed a higher-order unification algorithm [33], a nice description of which can be found in [28] and also in [7], §17.5.

Higher-order unification works on pairs of terms in normal form that are to be made equal. A term in normal form can be written as $\lambda x_1 \ldots x_n.\ f(u_1 \ldots u_p)$, where f is called the *head* of the term. If the head is a constant or one of the bound variables $x_1 \ldots x_n$, the term is called *rigid*, as f cannot be instantiated. If the head is a variable, the term is said to be *flexible*.

3.3.1 Rigid-Rigid Pairs

Rigid-rigid pairs will fail to be unified if the heads are different, or if they have a different number of parameters, assuming the terms are in η-normal form. Otherwise, when the heads are equal, unification will succeed if the parameters can be unified. For example, the terms

$$\lambda x_1 \ldots x_n.\ f(u_1 \ldots u_p) \text{ and } \lambda y_1 \ldots y_n.\ g(v_1 \ldots v_p)$$

are rigid terms with the same number of parameters. If f and g are equal, this will result in p new pairs (from the parameters) to be unified:

$$\lambda x_1 \ldots x_n.\ u_i \text{ and } \lambda y_1 \ldots y_n.\ v_i$$

for $1 \leq i \leq p$.

3.3.2 Flexible-Flexible Pairs

Unification of two flexible terms will always succeed, with a potentially infinite number of possible unifiers. To avoid this search space explosion, the higher-order unification algorithm simply reports that the terms are unifiable without searching for specific unifiers. In a theorem prover, it is practical to record flexible-flexible pairs as constraints on further unifications.

3.3.3 Rigid-Flexible Pairs

There are two ways of unifying a pair of a rigid and a flexible term: *imitation* and *projection*. We describe these by considering unification of a pair of terms, where the second term's head is a variable, $?F$, for which a substitution should be found:

$$\lambda x_1 \ldots x_n.\ f(u_1 \ldots u_p) \text{ and } \lambda x_1 \ldots x_n.\ ?F(v_1 \ldots v_q)$$

Imitation will try to unify the terms by, if f is a constant, attempting to make $?F$ the same as that constant, suggesting the instantiation:

$$?F \equiv \lambda y_1 \ldots y_q.\ f(?G_1(y_1 \ldots y_q) \ldots ?G_p(y_1 \ldots y_q))$$

where each G_i is a fresh variable.

Projection tries to make $?F$ equal to one of its arguments, in other words $?F$ behaves as an identity. Projection suggests instantiations of the form:

$$?F \equiv \lambda y_1 \ldots y_q.\ y_i(?G_1(y_1 \ldots y_q) \ldots ?G_p(y_1 \ldots y_q)$$

given that y_i is of the appropriate type.

3.3.4 Example

Consider unifying the terms *rev* $?l$ and $?F(rev\ ?Y)$, assuming types match. These form a rigid-flexible pair. There are two possibilities of imitation for $?F$. Firstly, $?F$ can be instantiated to $\lambda x.\ rev\ ?l$. By η-conversion, $\lambda x.\ rev\ ?l$ is reduced to just *rev* $?l$. The argument *rev* $?Y$ of $?F$ has been dropped. The second imitation results in $?F \equiv \lambda x.\ rev\ x$ with the additional unification $?l \equiv rev\ ?Y$.

By projection, we get the instantiation $?F \equiv \lambda x.\ x$, which is the identity function, while $?Y$ and $?l$ form a flexible-flexible pair.

3.4 Rippling

The rippling heuristic for reducing differences between terms was introduced in §2.3. We will here provide some more details about ripple measures, static and dynamic rippling, and illustrate the main features of rippling in a worked example.

3.4.1 Sum-of-Distance Ripple Measure

Recall that the ripple measure is defined as a well-founded order on annotated terms, and required to decrease with each step of ripple-rewriting, thus ensuring termination.

We will here use a ripple-measure based on the sum of distances from outward wave-fronts to the top of the term-tree and from inward wave-fronts to the nearest sink, occurring below one of its wave-holes. IsaPlanner also supports other types of ripple-measures, but the sum-of-distance measure has been shown to be the most efficient

[25]. Figure 3.1 shows the annotated term $\mathit{Suc}\ a^{\uparrow} + \lfloor b \rfloor = \mathit{Suc}(\lfloor b \rfloor + a)^{\downarrow}$, with sum-of-distance measure 4. The distance from the outward wave-front to the top of the term-tree is 2 and the distance from the inward-wave-front to the closest sink is also 2. When the ripple-measure is 0 we expect all remaining wave-fronts to be out of the way for an application of the given, that is, either at the top of the term-tree, or in a sink.

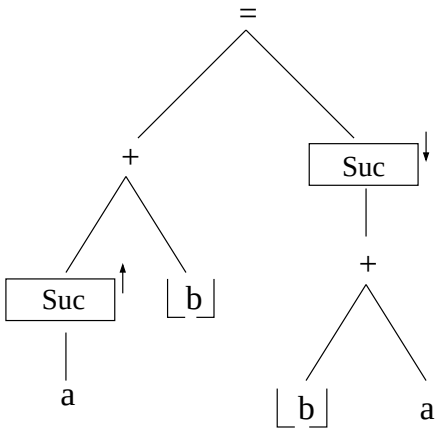

Figure 3.1: A tree-view of the annotated term, showing where the wave-fronts and sinks are located.

3.4.2 Examples

Example 1: A Rippling Proof

As an example of a rippling proof, consider the step-case of the inductive proof of the commutativity of addition:

$$\text{Given: } \forall b'.\ a + b' = b' + a$$
$$\text{Goal: } \mathit{Suc}\ a^{\uparrow} + \lfloor b \rfloor = \lfloor b \rfloor + \mathit{Suc}\ a^{\uparrow}$$

Assume the following rules are available[2]:

$$(\mathit{Suc}\ x) + y = \mathit{Suc}(x + y) \tag{3.1}$$

$$x + (\mathit{Suc}\ y) = \mathit{Suc}(x + y) \tag{3.2}$$

$$((\mathit{Suc}\ x) = (\mathit{Suc}\ y)) = (x = y) \tag{3.3}$$

[2]Following the conventions of *dynamic rippling* (see 3.4.3) the rules have not been annotated prior to rippling.

The proof of the step case then proceeds as follows:

$$
\begin{array}{rl}
\boxed{Suc\ a}^{\uparrow} + \lfloor b \rfloor = \lfloor b \rfloor + \boxed{Suc\ a}^{\uparrow} & \text{Measure: 4} \\
\Downarrow\ by\ (3.1) & \\
\boxed{Suc(a + \lfloor b \rfloor)}^{\uparrow} = \lfloor b \rfloor + \boxed{Suc\ a}^{\uparrow} & \text{Measure: 3} \\
\Downarrow\ by\ (3.2) & \\
\boxed{Suc(a + \lfloor b \rfloor)}^{\uparrow} = \boxed{Suc(\lfloor b \rfloor + a)}^{\uparrow} & \text{Measure: 2} \\
\Downarrow\ by\ (3.3) & \\
a + \lfloor b \rfloor = \lfloor b \rfloor + a & \text{Measure: 0} \\
\Downarrow\ Strong\ Fert. & \\
True &
\end{array}
$$

Each application of a rule moves the wave-fronts outwards, thus decreasing the ripple-measure. In the final step, the goal is an instance of the given and no wave-fronts remain so the measure is 0. The inductive hypothesis can be directly applied to conclude the proof. This is called *strong fertilisation*.

Example 2: Weak-fertilisation

Now, assume rule 3.3 was not available. No more rewrites would be possible at the sub-goal:

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

Rippling is said to be *blocked* at such a state. However, in many cases, including this, it is still possible to complete the proof by applying the inductive hypothesis as an extra rewrite rule. This is called *weak fertilisation*. In this example the blocked goal can be weak-fertilised to produce the new sub-goal $Suc(b+a) = Suc(b+a)$, which is true by reflexivity[3].

Example 3: A Simple Proof-Critic - Lemma Calculation

Only rule 3.1 above comes from the standard definition of addition in Peano arithmetic. Assuming rippling was only given the definitions of the function '+', and no lemmas

[3] An analogous option also exists, weak-fertilisation could be applied to the right-hand side of the goal.

proved by the user in advance, rippling would get blocked even earlier, at the sub-goal:

$$Suc(a + \lfloor b \rfloor)^{\uparrow} = \lfloor b \rfloor + Suc\ a^{\uparrow}$$

It is still possible to apply weak-fertilisation to the left-hand side of the blocked goal, resulting in the new sub-goal:

$$Suc(b + a) = b + (Suc\ a)$$

This goal is however not trivially solved, which is when the lemma calculation critic is fired. If applicable, the critic will apply common sub-term generalisation to the goal (replacing sub-terms occurring on both sides of an equation with a new variable). This is then followed by an attempt to prove the goal by a new inductive proof attempt. In this case, the lemma can indeed be proved, and is in fact the missing rule 3.2.

3.4.3 Static and Dynamic Rippling

There are two approaches to rippling, static and dynamic rippling, differing in how annotations are represented and handled. The traditional account of rippling, described in [10], is what we refer to as static rippling. In static rippling, rewrite rules are annotated in advance, in all possible measure decreasing ways with respect to some skeleton. Annotations are represented as object-level functions. Each rewrite rule typically gives rise to several annotated copies (wave-rules). These rules are then only allowed to be applied to a goal with matching annotations.

As mentioned in §2.3, dynamic rippling is required for higher-order domains, as the object-level annotations of static rippling are not stable over β-reduction [70]. In dynamic rippling, rewrite rules are not annotated at all. Instead, all ways of applying a rule to a goal are generated, and annotations then recomputed for the new goals. Any goals that turn out not to preserve the skeleton or decrease the measure are discarded. Each goal in dynamic rippling may have several possible annotations, analogous to the multiple wave-rules coming from a single rewrite rule in static rippling. Returning to the example about commutativity of addition, there are actually two ways of annotating the sub-goal $(Suc\ x + y) = y + (Suc\ x)$:

$$Suc\ x + \lfloor y \rfloor^{\uparrow} = \lfloor y \rfloor + Suc\ x^{\uparrow}$$

$$Suc\ x + \lfloor y \rfloor^{\downarrow} = \lfloor y \rfloor + Suc\ x^{\uparrow}$$

The two alternatives capture the fact that rippling may either attempt to move the wave-front on the left-hand side towards the top of the term tree (as in the previous example) or towards the sink (imagine applying rule 3.1 from right-to-left).

Grouping Ripple Measures

In many other cases it is possible to have several measure decreasing annotations. This raises the issue of how to search over alternative annotations of the same goal. If each one is treated separately, every rewrite will be considered for each annotation, which potentially increases the search space exponentially. Lucas Dixon suggests grouping all measures of a goal as a solution [25], chapter 7. This grouped measure keeps the highest measure of the current goal as a threshold, and only allows new annotations smaller than this threshold. The highest one of these becomes the new threshold.

3.5 Isabelle

Isabelle is a generic interactive theorem prover which allows implementation of a wide range of object logics, such as higher-order logic (HOL), Zermelo-Fraenkel set theory and many others [65]. Isabelle also has a language for writing proof-scripts called Isar [75]. A large library of theorems for various object logics is available on-line at `http://isabelle.in.tum.de`.

Each object logic is formalised in Isabelle's meta-logic, which is an intuitionistic higher-order logic with implication, universal quantifiers and equality [66]. Isabelle follows the LCF-approach to theorem proving, where new theorems can only be obtained from previously proved statements through applications of a small set of trusted sound inference rules. More complex tactics are built by combining these rules in different ways, ensuring that resulting proofs will also be sound. To facilitate interactive proof, Isabelle has a number of powerful automatic tactics, such as the Simplifier. The Simplifier is typically used to perform rewriting using a set of supplied equational rules. It may also introduce a split if it comes across an if-statement.

3.6 IsaPlanner

IsaPlanner is a proof-planner built on top of Isabelle [25]. Like Isabelle, IsaPlanner is generic and can support different object logics, and follows the LCF approach. There

are some important differences between Isabelle and IsaPlanner: IsaPlanner allows meta-variables to occur in assumptions, which is not allowed when writing proofs in Isabelle using Isar. IsaPlanner also gives sub-goals (and assumptions) explicit names, which facilitates writing automatic techniques. In Isabelle, sub-goals are just kept in a list, and may be reordered arbitrarily by tactics.

IsaPlanner supports automated inductive proofs using the rippling heuristic to guide search. Previous proof-planners in the CLAM-family produced an explicit plan, using methods with pre- and post-conditions, which was afterwards passed on to a theorem prover for checking. IsaPlanner does not have explicit pre- and post-conditions on methods, conditions are implicit in the control flow between *reasoning techniques*. The reasoning techniques execute tactics, so the proof-planning and execution can be interleaved, ensuring each step is sound. Reasoning techniques can both be Isabelle tactics, such as simplification, as well as more complex techniques such as rippling. IsaPlanner has a language for combining simple reasoning techniques into new, more complex ones.

A reasoning technique is applied to a *reasoning state* and produces a set of new reasoning states, one for each way the technique can be applied. A reasoning state is a data-structure capturing a snapshot of the proof-plan so far. This includes the next reasoning technique to be applied (if any) as well as contextual information about, for example, rippling annotations. It is worth mentioning that the proof-plan in the reasoning state is itself represented as a data-structure, responsible for, amongst other things, keeping track of lemmas found and of instantiations for meta-variables during the proof (see figure 3.2).

IsaPlanner also differs from the CLAM-family in that it supports having multiple versions of rippling at the same time. Rippling is implemented in a modular fashion, as a set of ML-functors with associated signatures, built on top of one another. Figure 3.3 shows how the modules depend on each other and gives a few examples of specific implementations. For example, rippling with case-analysis or lemma calculation are both instances of the 'Basic Rippling Technique' module. If we want rippling to use a different measure, no code change is needed, we simply import a different version of the modules rippling depends on. This also makes experimentation easy, we can simply compare different instances of the rippling technique module.

Isabelle and IsaPlanner are both implemented in a Standard ML, and typically used with the PolyML implementation[4]. The source code for IsaPlanner is avail-

[4] `http://www.polyml.org`

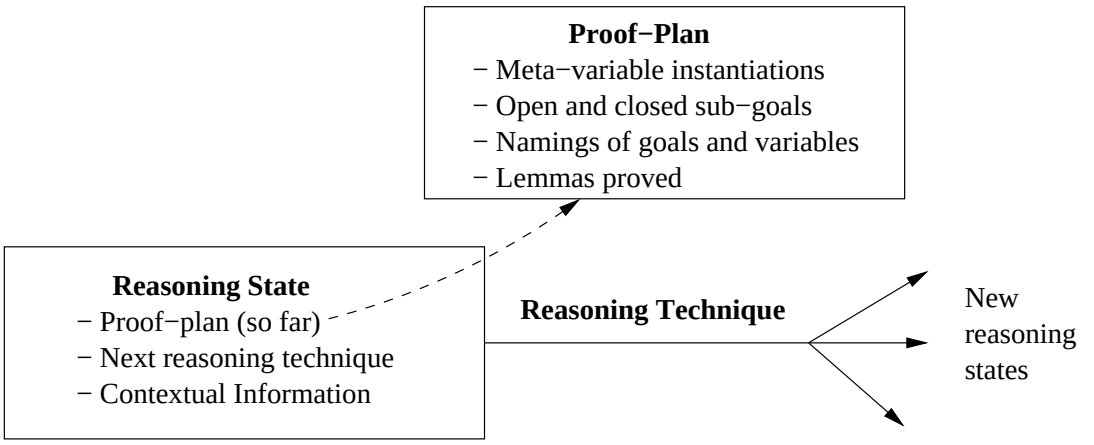

Figure 3.2: Reasoning states hold a partial proof-plan, which in turn keeps track of information such as currently open goals. Applying a reasoning technique produces new reasoning states, updating the proof-plan accordingly.

able on-line, see `http://dream.inf.ed.ac.uk/projects/isaplanner` for downloading instructions.

3.7 The Zipper Data-Structure

The zipper data-structure, first described by Huet [34], is used to represent a tree-structure, with a particular sub-tree that is the focus of attention. The focus of attention can then be moved up, down, left or right in the tree. A zipper for term-trees contains information about where a particular sub-term is located in a larger term. Figure 3.4 shows such a zipper of the term $(Suc\ a) + b$, where the focus is on the sub-term *Suc* and marked by a dashed box. The symbol '$' at the internal nodes of the term tree represents function application[5]. The term outside the focus of the zipper is sometimes refereed to as the *context* of the zipper. It is sometimes convenient to represent the context as a lambda-abstraction, where the focus is replaced by a bound variable. For the zipper in figure 3.4, the context is $\lambda\, f.\ (f\ a) + b$.

Using zippers to move around a term-tree takes time proportional to the distance moved. Access to the focused sub-term and its surrounding context is constant time. Zippers are widely used for IsaPlanner's equational reasoning (previously under the name *focus terms*) [25], and are also an important tool for the implementation of other techniques for reasoning about meta-variables.

[5]Following the convention of Isabelle's higher-order abstract syntax.

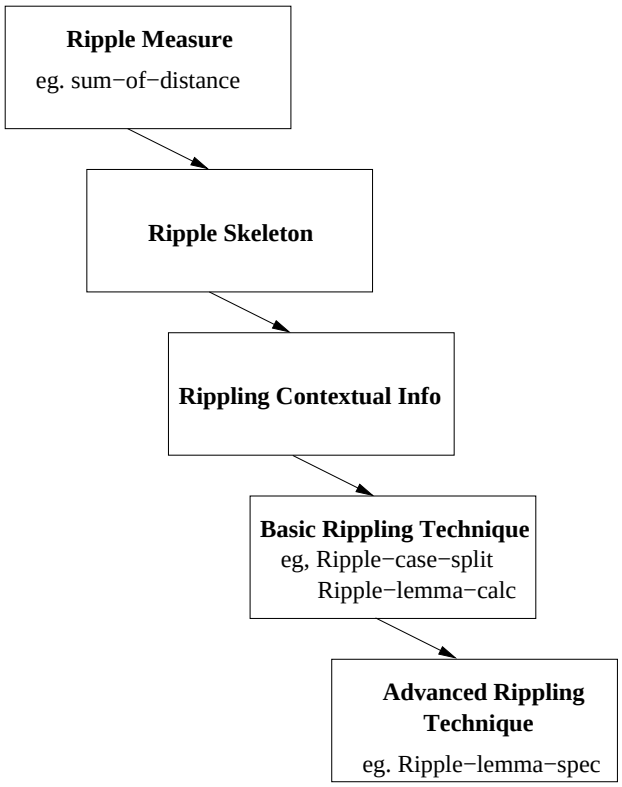

Figure 3.3: Hierarchy of IsaPlanner's implementation of rippling. Each module is defined in terms of the one above. Several versions of each module exists at the same time, allowing for multiple versions of rippling.

3.8 Summary

This chapter provides some useful background knowledge for understanding the work in this thesis. We first introduced some notational conventions. We then presented the laws of lambda-calculus, followed by a brief introduction to higher-order unification. Next, we presented the rippling heuristic for inductive proofs and explained the difference between static and dynamic rippling. The Isabelle theorem prover was described, along with conventions of how its terms are presented, followed by an overview of the IsaPlanner system in which our work has been implemented. Finally, we presented the zipper data-structure, which is widely used in our implementation for navigating and manipulating term-trees.

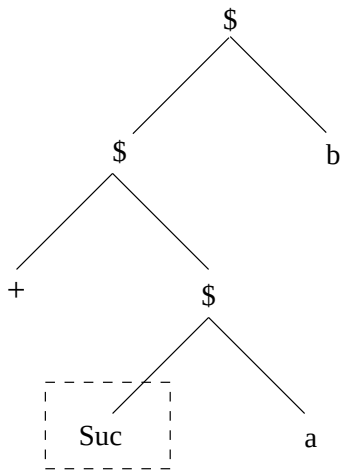

Figure 3.4: A zipper seen as a tree. The zipper is at the location *Suc*, marked by a dashed box

Chapter 4

Reasoning with Meta-Variables

4.1 Introduction

This chapter describes techniques developed for efficient reasoning with meta-variables. The *middle-out rewriting* technique (§4.2) provides heuristics for reducing the search space when rewriting goals containing meta-variables. Our *restricted resolution* technique manages resolution with rules that contain top-level meta-variables.

Meta-variables are used to stand for yet unknown term-structure by several proof critics such as lemma speculation and accumulator generalisation. The lemma-speculation critic (chapter 6) makes use of meta-variables to represent the unknown parts of lemmas that are in the process of being speculated. The schematic lemma is typically instantiated by applying it to a blocked goal, and then apply further rewriting to instantiate the meta-variables shared between the goal and the schematic lemma. However, the presence of meta-variables causes problems during rewriting as a meta-variable can be made to unify with any rule, thus producing a large search space. Restricting the number of potential unifiers is therefore important. The middle-out rewriting technique has been designed for this task.

Meta-variables may also occur in the head position of rules, standing for the context surrounding some particular sub-term of interest. If we wish to perform resolution with such a rule, restricting higher-order unification is important to avoid producing a large number of unwanted unifiers. Furthermore, without restrictions, the rule may apply to any goal, not only goals containing the particular sub-term of interest. Resolution with this type of rules is managed by our restricted resolution technique and used by the case-analysis technique described in chapter 5.

4.2 Middle-Out Rewriting

The meta-variables introduced in the goal by the application of a schematic lemma are instantiated by further rewriting guided by rippling. This instance of middle-out reasoning (see §2.4), applied to rippling rewrites, will be refereed to as *middle-out rewriting*.

Because we are using rippling and therefore want to reduce the differences between the goal and the inductive hypothesis, a candidate middle-out rewrite rule should, at least in part, match some non-variable sub-term of the goal. In order to efficiently find such rules, we introduce what we call *partial wave-rules*. A partial wave-rule contains a zipper of the wave-rule at the location of some function symbol, and is described in detail in §4.2.2.

4.2.1 Overview of the Algorithm

We here give a brief overview of the steps of the middle-out rewriting algorithm. The algorithm attempts to perform one step of rewriting with some rule at a redex containing a particular function symbol shared between the rule and the goal.

1. Find a partial wave-rule that shares a function symbol with the schematic goal[1]. The wave-rule and the goal are both represented as zippers (see §3.7) focused at the matching function symbol.

2. Move up both zippers one step at a time. At each step, check if a meta-variable occurs at the head-position of the focused sub-term of the goal-zipper.

3. When there is a meta-variable in the head-position of the focused sub-term in the goal-zipper, use the 'left-over' term context from the rule-zipper to instantiate the meta-variable in such a way that it becomes possible to rewrite the goal using the rule.

4. If there is no head meta-variable in the goal sub-term, repeat step 2, as long as the goal and rule match so far, otherwise fail.

5. If we reach the top of the rule-zipper and no meta-variable occurs in the head of the focused sub-term in the goal-zipper, it is safe to use regular unification and rewriting to rewrite the sub-term in this position of the goal.

[1] Recall that a schematic goal is a goal containing at least one meta-variable (see 3.1).

4.2.2 Partial Wave-Rules

Middle-out rewriting works by using *partial wave-rules*. A partial wave-rule contains a zipper of the wave-rule's left-hand side at the location of a function-symbol, along with the whole rule. Each wave-rule gives rise to several partial wave-rules, one for each function symbol present in its left-hand side. If a schematic goal contains any of these functions, this wave-rule is a candidate for rewriting, and thus also instantiating some of the meta-variables in the goal. We use this heuristic to reduce the number of candidate wave-rules needed to be considered. §4.2.3 describes how the rule-zipper's context is used to attempt to instantiate a meta-variable after a partial match has been found.

As an example, the wave-rule

$$(rev\ ?t)\ @\ (?h\ \#\ [\,]) = rev(?h\ \#\ ?t) \tag{4.1}$$

would give rise to three partial wave-rules with the zippers shown in figures 4.1, 4.2 and 4.3. These partial wave-rules reflect the fact that there are three function symbols, *rev*, @ and # in the left-hand side of this rule, and their corresponding zippers tells us where in the rule they occur.

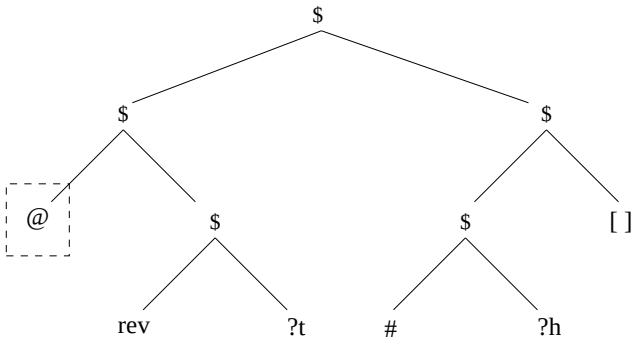

Figure 4.1: The zipper for the partial wave-rule focused at @.

4.2.3 Finding a Candidate Rewrite

The first step of the middle-out rewriting algorithm is to create a zipper of the schematic goal (which contains meta-variables), in order to be able to search through the term while maintaining its context. The leaves of the zipper are then searched for function symbols, such as *rev*. The occurrence of a particular function symbol indicates that any partial wave-rule containing the same symbol is a candidate to use for meta-variable instantiation and rewriting.

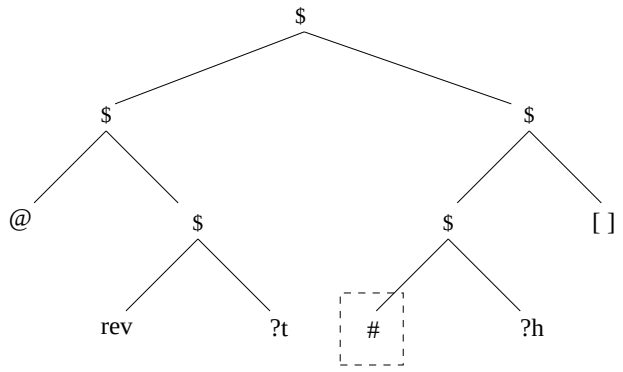

Figure 4.2: The zipper for the partial wave-rule focused at #.

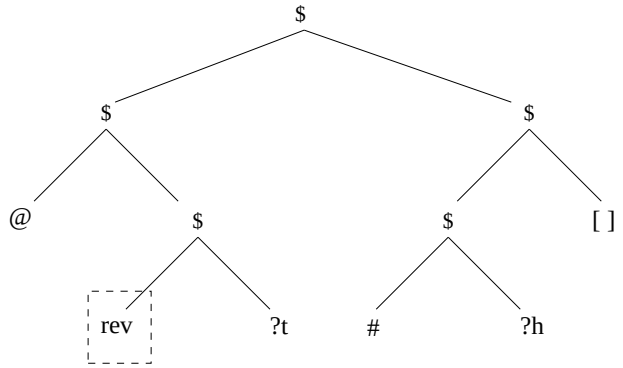

Figure 4.3: The zipper for the partial wave-rule focused at *rev*.

Assume that the term $?F(rev\ x)\ @\ y$ is part of some schematic goal. A search across a zipper of this term will find the locations of the *rev* and @ functions, and suggest using partial wave-rules about these functions. One candidate is the partial wave-rule about *rev* (from wave-rule 4.1) in figure 4.3.

When a matching partial wave-rule has been found the algorithm will incrementally move one step up the term tree in both the zippers for the rule and the goal as long as the current focused terms of the two zippers match (otherwise it fails). The procedure stops when either a meta-variable has been found in the head-position of the goal-term, or the top of the rule-zipper has been reached. In the latter case regular unification and rewriting can take place, as the sub-term being rewritten in the goal does not have a top-level meta-variable. Figure 4.4 illustrates the situation in which a meta-variable has been found.

The sub-term in the focus of the rule in the figure, $rev\ ?t$, unifies with the sub-term $rev\ x$ in the goal (with $?t \equiv x$). To make the rule applicable to the goal at this point, the meta-variable $?F$ must be instantiated to the remaining term-context of the rule-zipper.

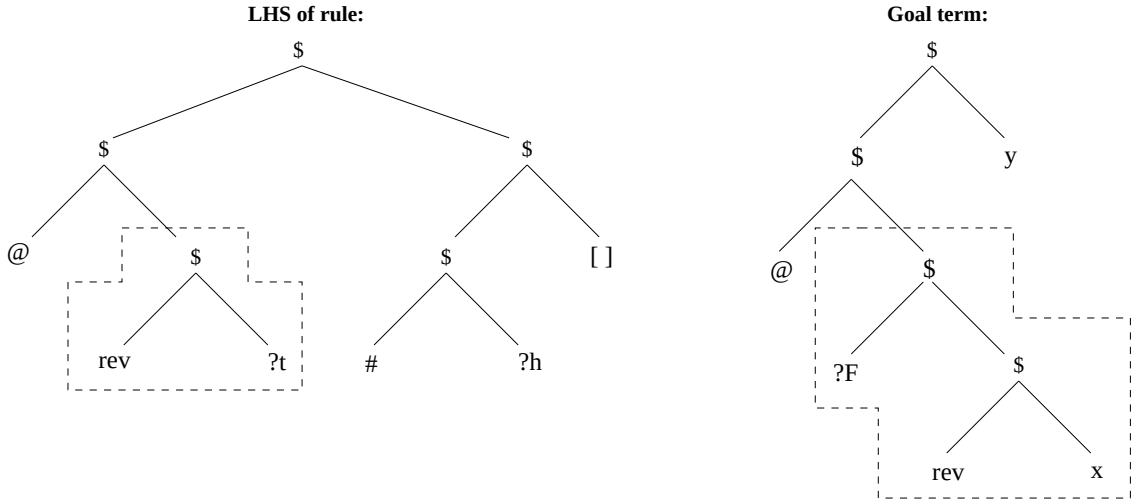

Figure 4.4: We have moved up both the rule-zipper (left) and the goal-zipper (right). The term in the focus of the goal-zipper now has a meta-variable in the head-position.

An appropriate instantiation for $?F$ can be constructed by creating a lambda-term from the rule-zipper in figure 4.4. We replace the sub-term in the focus (*rev* $?t$ in the dashed box in the figure), by a bound variable z [2], which results in the term $\lambda z.\ z\ @\ (?h_1\ \#\ [\,])$. Using this term to instantiate $?F$, and then beta-reducing, gives an instantiated version of the goal: $rev(x\ @\ (?h_1\ \#\ [\,]))\ @\ y$. Now, wave-rule 4.1, can be applied to the instantiated version of the goal without having to worry about producing too many unifiers. The goal rewrites to $rev(?h_1\ \#\ x)\ @\ y$.

4.3 Resolution with Restricted Unification

Restricting higher-order unification is sometimes also useful in the context of resolution. If resolution needs to be performed with a theorem containing a top-level meta-variable, reducing the number of unifiers is crucial. A top-level meta-variable typically stands for some arbitrary surrounding context of a sub-term we wish to resolve. Without any restrictions on unification, resolution would not only produce a large number of alternative unifiers, but also apply to any other goal, even those not containing the desired sub-term. Restricting higher-order unification in the context of resolution is essential for our case-analysis technique (Chapter 5).

Our algorithm for resolution with restricted unification first instantiates the top-

[2] At this point one must also check that this sub-term does not contain any dangling bound variables, in which case the attempted instantiation would be invalid. This is a standard variable capture avoidance condition.

level meta-variable of the theorem and then performs regular resolution.

4.3.1 Overview of the Algorithm

Below we give an overview of our algorithm for restricted resolution. As a small example, assume we have a rule of the form $?P(?a) \Longrightarrow ?P(f\ ?a\ ?b)$ which we wish to resolve with the goal $g(f\ x\ y)$.

1. **Find the argument of the top-level meta-variable:** The first step of our restricted variant of resolution is to check if there indeed is a top-level meta-variable in the conclusion of the rule, otherwise it is safe to proceed with normal resolution. If there is a top-level meta-variable, its argument should match some sub-term of the goal that is to be resolved. In our example above, there is indeed a top-level meta-variable, $?P$, in the rule, which has the argument $(f\ ?a\ ?b)$. A zipper is used to find this sub-term.

2. **Find a matching sub-term in the goal:** The next step is to find a sub-term in the goal which unifies with the argument of the rule's top-level meta-variable. Using a zipper we traverse the parse-tree of the goal until a sub-term matching the argument of the meta-variable is found. In the example, $?P$ has an argument, $(f\ ?a\ ?b)$, which matches the sub-term $(f\ x\ y)$ in the goal.

3. **Instantiate the top-level meta-variable:** The term context surrounding the matching sub-term in the goal is used to construct an instantiation for the rule's top-level meta-variable. The instantiation is created by replacing the sub-term in the goal zipper's focus with a bound variable and abstracting over it. In our example, this gives the instantiation $?P \equiv \lambda\, z.\ g(z)$.

4. **Resolve with the instantiated theorem:** Finally, resolution is performed using the instantiated version of the theorem. In our example, we resolve the goal, $g(f\ x\ y)$, with the partially instantiated rule: $g(?a) \Longrightarrow g(f\ ?a\ ?b)$. This instantiates the remaining variables to $?a \equiv x$ and $?b \equiv y$, thus producing the new goal $g(x)$.

4.3.2 Example: Splitting an If-Statement

As an example consider the following goal on which we wish to apply a case-split on the condition of the if-statement:

$$if\ (x = h)\ then\ True\ else\ (member(x,\ t\ @\ l) = member(x,\ h\#t)\ \vee\ member(x,\ l))$$

In IsaPlanner, a split on an if-statement is performed by resolving the goal with the following library theorem:

$$[\![?Q \Longrightarrow ?P(?y);\ \neg ?Q \Longrightarrow ?P(?z)\ ]\!] \Longrightarrow ?P\ (if\ ?Q\ then\ ?y\ else\ ?z) \qquad (4.2)$$

We expect this to result in two new sub-goals, as the original goal contains an if-statement as a sub-term in some context $?P$, where $?P$ is expected to unify with the remainder of the goal-term. However, before applying resolution blindly we should instantiate 4.2 to avoid undesired results.

The first step of the algorithm establishes that there is indeed a top-level meta-variable, $?P$ of theorem 4.2, and that it has a non-variable argument, $if\ ?Q\ then\ ?y\ else\ ?z$.

Next, the goal is traversed to find a matching sub-term, using a zipper. Figure 4.5 shows the zipper having located the matching if-statement in the goal. If no such sub-term can be found, restricted resolution is not applicable and the algorithm fails.

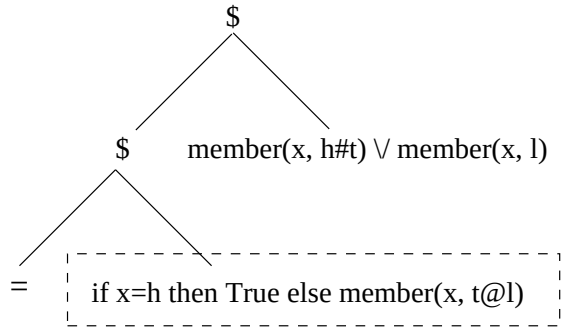

Figure 4.5: The zipper is at the location of an if-statement. Parts of the term-tree have been collapsed for clarity.

Using this zipper, an instantiation is created for $?P$. As $?P$ is to match the surrounding context of the if-statement, an instantiation-term can be created by replacing the term in focus of the goal-zipper (inside the dashed box in figure 4.5) with a new bound variable u, and abstracting over it. We get the instantiation:

$$?P\ \equiv\ \lambda u.\ u = member(x,\ h\#t)\ \vee\ member(x,\ l)$$

Theorem 4.2 now takes the form:

$$\begin{aligned}[\![(?Q \Longrightarrow ?y) = member(x,\ h\#t)\ \vee\ member(x,\ l);\\ (\neg ?Q \Longrightarrow ?z) = member(x,\ h\#t)\ \vee\ member(x,\ l)\ ]\!]\\ \Longrightarrow (if\ ?Q\ then\ ?y\ else\ ?z) = member(x,\ h\#t)\ \vee\ member(x,\ l)\end{aligned}$$

It is now safe to apply it to the goal using normal resolution, which instantiates the remaining meta-variables as expected. The result is two new goals:

$$x = h \Longrightarrow (True = member(x,\ h\#t)\ \vee\ member(x,\ l))$$

$$x \neq h \Longrightarrow (member(x,\ t\ @\ l) = member(x,\ h\#t)\ \vee\ member(x,\ l))$$

Naively applying resolution would result in a larger search space with alternative unwanted sub-goals, resulting from unifying $?P$ with the entire goal-term while allowing it to throw away its argument. In our example, such an instantiation is:

$$?P \equiv \lambda u.\ if\ (x = h)\ then\ True\ else\ (member(x,\ t\ @\ l) = member(x,\ h\#t) \vee member(x,\ l))$$

Note that the bound variable u does not appear in the body of the abstraction, the argument of $?P$ has simply been dropped. This would allow resolution to go through, but the resulting sub-goals would not be useful in a proof ($?Q$, $?y$ and $?z$ would remain uninstantiated).

4.4 Related Work

4.4.1 Middle-Out Reasoning

Previous work in middle-out reasoning has also come across the problem of getting too many higher-order unifiers, many of them undesirable.

Andrew Ireland's lemma-speculation critic [41], made use of rippling annotations to help restrict which wave-rules were allowed to be applied to schematic terms. Meta-variables were also restricted to being second order. The contents of wave-fronts (or sinks) in the goal were first matched with wave-fronts in the rules. This was possible as CLAM employs static rippling and thus annotates all wave-rules in advance. IsaPlanner, however, uses dynamic rippling, which means that there are no annotations present at the object-level. Annotations are instead re-computed after each step of rewriting. Our middle-out rewriting technique only attempts a rewrite if the same function symbol can be found both in the rule and the schematic goal. Starting from the matching function symbol, we try to unify a non-variable sub-term with some part of the rule, and deal with the rule's 'left-overs' by instantiating a meta-variable. As opposed to Ireland's technique, our version of middle-out rewriting could be used independently of rippling.

Hesketh et al. [31], used middle-out reasoning to synthesise tail-recursive functions. Here, meta-variables stand for the functions that are being synthesised. After higher-order unification, the results were filtered to eliminate 'not sensible' unifiers. Any unifiers introducing universally quantified or free variables into the body of the function were discarded. Unlike our algorithm, this would not avoid the possibility of applying every single rewrite rule to a term with a top-level meta-variable, afterwards having to discard a large number of unwanted results.

4.4.2 Higher-Order Narrowing

Narrowing is a technique for solving equational problems by rewriting, where both goal and rule may contain meta-variables, as is the case in our middle-out rewriting program. Christian Prehofer has extended narrowing to higher-order logic [67]. He found that the first order notion of narrowing could successfully be extended to a subset of higher-order terms, called *higher-order patterns*. These are restricted not to allow meta-variables to occur as arguments to other meta-variables. However, many of the schematic goals we need to deal with for lemma speculation are not higher-order patterns. For full higher-order narrowing, Prehofer suggests using *lazy narrowing*, where equational reasoning is integrated into unification. Lazy narrowing is complete, as opposed to our middle-out rewriting, which deliberately sacrifices completeness for reductions in search space size. Furthermore, we do not usually even want all of the possible unifiers and rewrites.

4.5 Summary

Many proof-critics make use of meta-variables to stand for unknown term-structures. Instantiations of these variables are typically found by applying further rewrite rules. If an unrestricted version of higher-order unification is used, any rewrite rule can be made to unify with a meta-variable, leading to a very large search space. The middle-out rewriting technique restricts this search space by first matching non-variable sub-terms, and then attempting to instantiate a meta-variable with the rest of the rule, making it applicable. This allows us to filter out undesirable cases where any rule is made to unify with a meta-variable.

Sometimes meta-variables occur on the top-level in theorems we wish to apply to a goal by resolution. Our algorithm for resolution with restricted unification first

searches for a match of non-variable sub-terms in the goal and the theorem, and uses the remaining term-structure in the goal to instantiate the top-level meta-variable. This avoids many undesirable instantiations and reduces the search space. It also ensures such a theorem cannot be applied to every goal.

Chapter 5

Case-Analysis

5.1 Introduction

Case-splits are essential if we are to automate proofs about theorems involving functions with conditionals in their definitions. Examples include common list operations, such as *member* and *delete*, as well as operations on natural numbers, such as subtraction and $\leq$. Automatic case-splitting was previously not available in IsaPlanner. We have implemented a case-analysis technique which, if necessary, can introduce case-splits both on the boolean condition of an if-statement and on arbitrary datatypes in the presence of a case-statement. Case-statements are higher-order constructs, which allow pattern matching on datatypes, commonly used in many function definitions. Identifying when introducing a split on a case-statement is appropriate can be viewed as a eureka step, as naively allowing case-splits may lead to non-termination. This is why Isabelle's simplifier does not attempt splits on case-statements. Automating case-splitting increases the number of proofs about such functions that can proved automatically, which is confirmed by our results (§5.5).

Our approach to case-analysis is to unfold if- and case-statements within rippling. The restrictions on rippling to decrease its measure ensures that termination is preserved, even when introducing splits over arbitrary datatypes. Whenever a conditional statement is discovered in the goal, the case-split technique is triggered before any more rippling to other parts of the goal is performed. The technique either picks the relevant branch, or introduces a split if necessary. After a split it is often the case that some branch no longer preserves the skeleton of rippling. Such goals are solved by simplification before rippling is resumed. Occasionally, all sub-goals after a split may be non-rippling goals and solved by simplification, in which case the proof is

concluded without further rippling[1].

Sections 5.2 and 5.3 describe how our technique handles case- and if-statements respectively.

5.2 Case-Statements

For each user-defined datatype, Isabelle automatically derives its defining equations and creates a case-construct for the datatype, to use for case-based pattern matching [65]. Isabelle also derives rules for how to split case-statements. As an example, the case-construct for natural numbers is called $nat\_case\ :\ \alpha \Rightarrow (nat \Rightarrow \alpha) \Rightarrow nat \Rightarrow \alpha$. When case-constructs are applied to their arguments, Isabelle writes them in the more conventional style:

$$case\ n\ of\ 0 \Rightarrow f_1 \mid (Suc\ x) \Rightarrow f_2\ x$$

Here, n is the third argument of the *nat_case* function above, while f_1 is the first and the function f_2 the second. We refer to these kinds of expressions as *case-statements*.

On encountering a case-statement, our technique will first inspect the term in order to pick the relevant branch if either pattern can be matched. Returning to our example of natural numbers, this involves attempting substitution with either one of the following theorems, which are automatically derived by Isabelle:

$$case\ 0\ of\ 0 \Rightarrow ?f_1\ \mid (Suc\ x) \Rightarrow ?f_2\ x\ = ?f_1$$

$$case\ (Suc\ ?n)\ of\ 0 \Rightarrow ?f_1 \mid (Suc\ x) \Rightarrow ?f_2\ x\ = ?f_2\ ?n$$

If neither of the above rules match, a case-split must be introduced, using another automatically derived theorem[2]:

$$[\![?n = 0 \Longrightarrow ?P(?f_1);\ \forall x.\ (?n = Suc\ x)\ \Longrightarrow ?P(?f_2\ x)]\!] \Longrightarrow \\ ?P(case\ ?n\ of\ 0\ \Rightarrow ?f_1 \mid (Suc\ x)\ \Rightarrow\ (?f_2\ x)) \quad (5.1)$$

The meta-variable $?P$ above, stands for the context in which the case-statement occurs. The actual case-split is implemented as a single resolution step with theorem 5.1. However, note that as the meta-variable $?P$ occurs on the top-level of the theorem, it could be applied to any goal, not just the intended goals containing case-statements. It

[1] This typically indicates that the proof did not actually require induction, but rather a proof by case-analysis.

[2] This theorem is derived by IsaPlanner, from the slightly different version derived by Isabelle.

may also produce additional undesirable instantiations for goals that do contain case-statements. In an interactive setting, the user can specify the instantiation for $?P$, but as our system is automatic, it has to employ the resolution technique with restricted unification, as described in §4.3. This ensures we only get the results we want.

Example

Consider the proof of the commutativity of the *max* function, where *max* is defined as follows for the *Suc*-case:

$$max\ (Suc\ x)\ y = case\ y\ of\ 0 \Rightarrow (Suc\ x)\ |\ (Suc\ z) \Rightarrow Suc\ (max\ x\ z) \tag{5.2}$$

The step-case of the proof is:

$$\begin{aligned} &\text{Inductive hypothesis:} && \forall b.\ max\ a\ b = max\ b\ a \\ &\text{Step-case goal:} && max\ \boxed{Suc\ \boxed{a}}^{\uparrow}\ \lfloor b' \rfloor = max\ \lfloor b' \rfloor\ \boxed{Suc\ \boxed{a}}^{\uparrow} \end{aligned} \tag{5.3}$$

By applying rule 5.2, the left hand side of the step-case is rippled to:

$$\boxed{case\ b'\ of\ 0 \Rightarrow (Suc\ a)\ |\ (Suc\ z) \Rightarrow Suc(\boxed{max\ a\ \lfloor z \rfloor})^{\uparrow}}^{\uparrow} = max\ \lfloor b' \rfloor\ \boxed{Suc\ \boxed{a}}^{\uparrow}$$

At this point, the case-split technique is triggered, due to the introduction of a case-statement. We cannot proceed down either branch of the case-statement as we lack information about the structure of b'. A case-split on b' is thus introduced by restricted resolution with theorem 5.1. This first instantiates $P \equiv \lambda\ x.\ x = max\ b'\ (Suc\ a)$, and then produces the two new sub-goals:

$$b' = 0 \implies Suc\ a = max\ b'\ (Suc\ a) \tag{5.4}$$

$$b' = Suc\ z \implies \boxed{Suc(\boxed{max\ a\ \lfloor z \rfloor})^{\uparrow}}^{\uparrow} = max\ \lfloor b' \rfloor\ \boxed{Suc\ \boxed{a}}^{\uparrow} \tag{5.5}$$

Goal 5.4 does not embed the skeleton and is thus solved by simplification[3]. Goal 5.5 on the other hand, still has an embedding and will be proved by further rippling.

Observe that following a case-split, an equational assumption, stating the particular value that the case-split term takes, is introduced for each branch. The equation is then substituted in each goal's conclusion. Not doing so would complicate further rewriting and lemma calculation, as the case-split sub-term would have two representations. In our example, this means replacing the occurrences of b' on the right-hand side with 0 and *Suc z* respectively. This is the final step involved in splitting a case-statement into its possible constructor cases.

[3]We describe experiments with other ways of solving non-rippling goals in §5.5

5.3 If-Statements

Our case-analysis technique can also handle boolean conditions in if-statements. In a similar fashion as for case-statements, the technique will first attempt to verify the condition or its negation and proceed down the corresponding branch. This is implemented as attempted substitution with the library theorems below:

$$?P \Longrightarrow (\mathit{if}\ ?P\ \mathit{then}\ ?x\ \mathit{else}\ ?y) = ?x \tag{5.6}$$

$$\neg ?P \Longrightarrow (\mathit{if}\ ?P\ \mathit{then}\ ?x\ \mathit{else}\ ?y) = ?y \tag{5.7}$$

Applying theorem 5.6 results in two sub-goals, one proving the condition $?P$ to be true and one proving the then-branch, $?x$. Similarly, applying theorem 5.7 requires us to prove $?P$ is false, and then prove the else-branch, $?y$. The goal arising from the condition is solved either by resolution with an existing assumption, or by simplification. The goal from the branch is passed back to rippling.

If failing to show that either the condition $?P$ or its negation holds, a split on the condition should be introduced. This is performed using resolution with restricted unification, to avoid undesirable unifiers (as described in §4.3), using the library theorem:

$$[\![?Q \Longrightarrow ?P(?y);\ \neg ?Q \Longrightarrow ?P(?z)\ ]\!] \Longrightarrow ?P\ (\mathit{if}\ ?Q\ \mathit{then}\ ?y\ \mathit{else}\ ?z) \tag{5.8}$$

As before, this gives us two new sub-goals. It is common that only one of these goals still embeds the skeleton from rippling. As for case-statements, non-rippling goals are required to be solved by simplification.

Example

As an example, consider the following theorem:

$$x\ \mathit{member}\ (l\ @\ m) = x\ \mathit{member}\ l\ \vee\ x\ \mathit{member}\ m$$

The proof proceeds by induction on l and then uses the definition of member:

$$x\ \mathit{member}\ (h\ \#\ t) = \mathit{if}\ (x = h)\ \mathit{then}\ \mathit{True}\ \mathit{else}\ (x\ \mathit{member}\ t) \tag{5.9}$$

By rippling using rule 5.9, the step-case goal becomes:

$$\mathit{if}\ (x = h)\ \mathit{then}\ \mathit{True}\ \mathit{else}\ x\ \mathit{member}\ (l\ @\ m)\ ^{\uparrow} = x\ \mathit{member}\ (h\ \#\ l)^{\uparrow} \vee x\ \mathit{member}\ m$$

The case-analysis technique is triggered on the discovery of an if-statement in the goal. It is not possible to prove the condition, $x = h$, or its negation at this stage, so a split is introduced. Restricted resolution with theorem 5.8, gives two new sub-goals:

$$x = h \Longrightarrow True = x\ member(h\ \#\ l)\ \vee\ x\ member\ m \tag{5.10}$$

$$x \neq h \Longrightarrow x\ member\ (l\ @\ m) = x\ member\ \boxed{(h\#l)}^{\uparrow}\ \vee\ x\ member\ m \tag{5.11}$$

The skeleton does not embed into goal 5.10 so it is passed to the simplifier, which successfully solves it. Goal 5.11 can be rippled further by rewriting the right hand side to:

$$x \neq h \Longrightarrow x\ member\ (l\ @\ m) = \boxed{if\ (x = h)\ then\ True\ else\ \boxed{x\ member\ l}}^{\uparrow}\ \vee\ x\ member\ m$$

This time, taking the *else*-branch succeeds, as the assumption introduced by the previous case-split can be used to show the negation of the condition. The proof can now be finished by strong fertilisation.

5.4 Eager or Lazy Case-Splits

Our case-analysis technique is interleaved with rippling and applied eagerly whenever a rule introduces a case- (or if-) statement. Such a rule application, followed by the case-split itself, is regarded as one ripple-step. This has two main advantages over delaying the case-splitting until rippling is blocked, or treating the case-split as a separate ripple-step.

Firstly, some ripple-measures are not reduced between the goal containing a case-statement and the resulting goal after the split. In the example about the commutativity of *max* in §5.2, the goal before the split:

$$\boxed{case\ b'\ of\ 0 \Rightarrow (Suc\ a)\ |\ (Suc\ z) \Rightarrow Suc(\boxed{max\ a\ \lfloor z \rfloor})}^{\uparrow} = max\ \lfloor b' \rfloor\ \boxed{Suc\ a}^{\uparrow}$$

has a wave-front in the same position as the ripple-goal after the split:

$$b' = Suc\ z \Longrightarrow \boxed{Suc(\boxed{max\ a\ \lfloor z \rfloor})}^{\uparrow} = max\ \lfloor b' \rfloor\ \boxed{Suc\ a}^{\uparrow}$$

Ripple measures, such as the ones currently used by IsaPlanner, do not take the size of the wave-front into account, and will thus disallow the above step as non-measure decreasing. By treating the application of the rule introducing the case-statement (here rule 5.2 from the definition of *max*) and the subsequent case-split as one ripple-step, all

known ripple measures will decrease with respect to the previous step-case goal (goal 5.3 on page 49). Similarly, for splitting data-types, substitution with the introduced case-assumption, as described in §5.2, is typically not measure decreasing and hence needs to be included as part of the compound ripple-step.

Secondly, when no actual case-split is needed, and the if- or case-statement can be reduced to a known branch, the eager approach proceeds directly to the relevant branch. If the if- or case-statement is allowed to remain in the goal, redundant rippling steps might be applied to the branch that is later to be discarded. Eager application of case-splitting is thus more efficient and provides shorter proofs on such theorems.

5.5 Evaluation

Functions with conditional definitions are very common, but many proofs requiring case-splits could not previously be found by rippling-based methods. Rippling with the case-analysis technique has been evaluated in IsaPlanner, on a set of 87 example theorems about functions defined using if- or case-statements on lists, natural numbers and binary trees. None of the theorems could previously be proved automatically in IsaPlanner, but using the case-analysis technique, 47 new proofs can be found.

Of the theorems in the evaluation corpus, 41 involve if-statements, another 41 involve case-statements and 5 involve both. Most of the theorems are a subset of inductive theorems from Isabelle's libraries for lists and natural numbers[4]. Some are more programmatic in character and taken from a corpus for the CLAM system [41], and from problems arising in dependently typed programming. We have also added some further theorems to evaluate additional properties. The evaluation corpus and results for rippling are included in Appendix B. They are also available on-line:
`http://dream.inf.ed.ac.uk/projects/lemmadiscovery/case_results.php`.

We did not expect IsaPlanner to prove all theorems in the corpus, even with the new case-analysis technique. Although some progress is made in the proofs, many of the remaining 40 theorems require more sophisticated reasoning about the side-conditions than IsaPlanner is currently capable of. Some theorems also require conditional lemmas that are beyond the scope of IsaPlanner's lemma discovery machinery. These theorems are included in the corpus to identify where IsaPlanner's case-splitting technique fails and how it can be improved. We discuss these limitation in §5.6.

In addition to the case-analysis technique, lemma calculation was also available

[4] `isabelle.in.tum.de/dist/library/HOL/index.html`

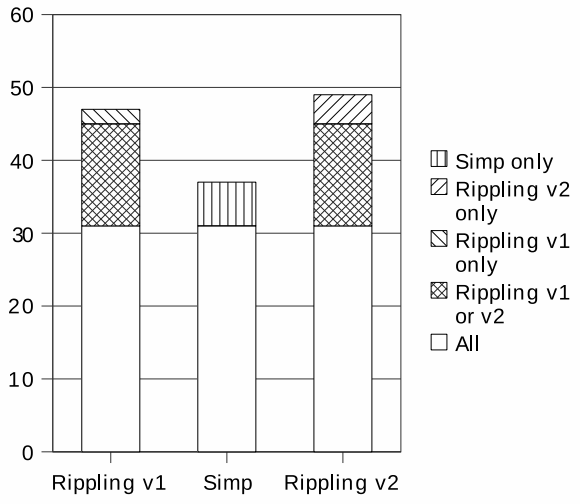

Figure 5.1: Number of theorems proved by each of the three versions of IsaPlanner's inductive prover. The number of theorems proved only by some technique(s) has also been marked. Note that each technique proves some theorems the others do not.

to rippling during the experiments. The theorems were proved only from function definitions (see Appendix A), no additional lemmas were provided in advance. The experiments were conducted on an Intel Xenon 2 GHz processor. Each proof had a timeout limit of 30 seconds, but all rippling proofs were found in less than one second, while some failed proof attempts took slightly longer, at most taking 9 seconds.

Other Versions of IsaPlanner's Inductive Prover

We have compared IsaPlanner's rippling-based inductive prover equipped with the case-analysis technique described above (we refer to this as 'version 1' of the rippling-based prover), with a simpler one, which applies induction followed by Isabelle's simplifier and lemma calculation [26]. We also experimented with an extension to our case-analysis technique ('version 2' of the rippling-based prover). This version allowed non-rippling goals arising after splits, but not solvable by simplification, to be tackled by lemma calculation. Below, a comparison of the results for each of the three versions is made. The results are visualised in figure 5.1.

Induction and Simplification Version

The simplification based prover managed to prove 37 theorems from the corpus. The full results for simplification are available on the website given above. There are 16

theorems that rippling (version 1) proves, but simplification does not. Most of these require a split on a datatype from a case-statement, which simplification is unable to preform. There are 6 theorems that the simplification based prover can solve, but not rippling (see figure 5.1). The majority of these involve if-statements with a non-trivial condition. Recall that our case-analysis technique attempts to show conditions by simplification or resolution with an assumption. The simplification based prover on the other hand, first attempts to show which branch to go down, but may then try to prove the condition using lemma calculation. The rippling-based prover does not allow lemma speculation in these situations, and hence fails. An example is the proof of the theorem *last (xs @ [x]) = x*. By induction on x, the step-case, will ripple to the goal:

$$if\ (xs\ @\ [x]) = [\,]\ then\ h\ else\ last\ (xs\ @\ [x])$$

The condition, or its negation, of the if-statement cannot be proved by simplification. However, the negated condition, can be proved by induction as a lemma: $(xs\ @\ [x] \neq [])$. Having successfully proved this, the simplification-based prover proceeds down the *else*-branch and concludes the proof by strong fertilisation.

However, applying lemma calculation more widely increases the risk of non-termination. The simplification-based prover often fails to terminate on theorems it cannot prove, as it often attempts to prove an infinite chain of increasingly more complicated conjectures. The 30 second time-out limit is reached during 39 proof attempts by the simplification based technique. In contrast, the rippling-based prover does not have these problems, and terminates on all theorems in the corpus. We conclude that the heuristic guidance of rippling leads to better lemmas being calculated.

Rippling Version 2: Lemma Calculation for Non-Rippling Goals

After a case-split we observed that the skeleton often does not embed into one of the new sub-goals. Such non-rippling goals were required to be solved by simplification in version 1 of the rippling prover. However, we observed that some more complicated theorems produced non-rippling goals that simplification could not solve. We therefore modified the case-analysis technique for rippling, to attempt to prove such goals by lemma calculation if simplification failed. This extended technique managed to prove 49 theorems, four of which the original technique failed to prove. Two theorems could however no longer be proved, as the time-out limit of 30 seconds was reached. The results are shown in figure 5.1, with full details available on-line.

As for the simplification-based prover, allowing lemma calculation for non-rippling goals after splits sometimes causes non-termination due to infinite sequences of conjectures being produced. The time-out limit was reached in 13 proof attempts. When non-rippling goals are required to be solved only by simplification (as in version 1 of our rippling-based prover), slightly fewer theorems are proved, but the prover is more reliable with respect to termination.

5.6 Limitations and Further Work

We have identified some of the main reasons why IsaPlanner fails on the remaining theorems in the evaluation corpus, and will here discuss these in more detail and suggest further extensions to rippling-based inductive theorem proving.

5.6.1 Conditional Lemmas

IsaPlanner is currently not able to produce conditional lemmas, in the form of implications. An example where this is needed is the proof that insertion sort produces a sorted list: $sorted(insertion\_sort\ l)$. $insertion\_sort$ is defined as follows for a non-empty list:

$$insertion\_sort(h \# t) = insert\ h\ (insertion\_sort\ t)$$

Using the above rule, the step-case of the proof ripples to:

$$sorted(\ \boxed{insert\ h\ (\boxed{insertion\_sort\ l})}^{\uparrow}\ )$$

After this single ripple-step the goal is blocked. To complete the proof, we require a conditional lemma, which IsaPlanner cannot currently find:

$$sorted\ m \Longrightarrow sorted(insert\ x\ m) \tag{5.12}$$

The required lemma is just a generalisation of the step-case goal with the inductive hypothesis as an assumption (the common sub-term $sort\ l$ has been generalised to m). If lemma calculation was to include assumptions (such as the inductive hypothesis) of the blocked goal, the above lemma could easily be found. However, we do not always want the inductive hypothesis as an extra assumption to our lemmas, it should only be included when it is needed in the proof of the lemma. Otherwise, the lemma may be unnecessarily hard to prove and less applicable to future proofs.

5.6.2 Reasoning with Assumptions

The proof of lemma 5.12 itself highlights further limitations in how IsaPlanner reasons with assumptions. In particular, there are problems when the induction variable occurs in an assumption. Applying induction on m in our candidate lemma gives the following step-case:

$$\text{IH:} \quad sorted\ m \Longrightarrow sorted(insert\ x\ m)$$

$$\text{Goal:} \quad sorted\ (h\ \#\ m) \Longrightarrow sorted(insert\ x(\ h\ \#\ m\ ^{\uparrow}\ ))$$

The goal can be rippled using the definition of *insert*:

$$insert\ x\ (h\ \#\ t) = if\ (x\ <\ h)\ then\ (x\ \#\ h\ \#\ t)\ else\ (h\ \#\ (insert\ x\ t))$$

Using this rule results in a new goal with an if-statement, so the case-analysis technique is triggered and introduces a split on the condition $x < h$. We get two new goals:

$$[\![sorted\ (h\ \#\ m);\ x\ <\ h]\!] \Longrightarrow sorted(x\ \#\ h\ \#\ m) \tag{5.13}$$

$$[\![sorted\ (h\ \#\ m);\ \neg(x\ <\ h)]\!] \Longrightarrow sorted(h\ \#\ (insert\ x\ m)) \tag{5.14}$$

The skeleton no longer embeds into goal 5.13 so it should be solved by simplification, using the two assumptions. The skeleton does however still embed into goal 5.14, and strong fertilisation is even possible. Recall that strong fertilisation is implemented as a resolution step in IsaPlanner. Because the inductive hypothesis itself has an assumption, the sub-goal $sorted\ (h\ \#\ m) \Longrightarrow sorted\ m$ will remain after fertilisation. This kind of fertilisation, where the hypothesis has an assumption, is called *piecewise fertilisation*. To prove the sub-goal remaining after piecewise fertilisation a forward proof could be carried out, rewriting the assumption to match the conclusion. Rippling annotations on the assumption could be used to guide rewriting, annotating the assumption with respect to the conclusion, e.g. $sorted\ (\ h\ \#\ m\ )\Longrightarrow sorted\ m$.

5.6.3 Other Induction Schemes

As mentioned above, IsaPlanner attempts to solve any non-rippling goals arising after a case-split by simplification. However, sometimes these goals will themselves require a further case-split, which the simplifier cannot perform (see 5.7.4). In these cases, IsaPlanner's default induction scheme, which employs structural induction on the relevant datatype, is not sufficient.

An example of this is the proof of commutativity of subtraction, expressed in Isabelle as *(i - j) - k = (i - k) - j*. By induction on i, and a case-split on j on the left-hand side, we get the two goals:

$$j = 0 \Longrightarrow (Suc\ i)\ -\ k = ((Suc\ i)\ -\ k)\ -\ 0 \tag{5.15}$$

$$j = Suc\ x \Longrightarrow \boxed{Suc(\boxed{i}\ -\ \lfloor x \rfloor)}^{\uparrow}\ -\ \lfloor k \rfloor = (\ \boxed{(Suc\ \boxed{i})}^{\uparrow}\ -\ \lfloor k \rfloor)\ -\ \lfloor (Suc\ x) \rfloor \tag{5.16}$$

The attempt to solve the non-rippling goal 5.15 by simplification fails, as the right-hand side would require a further case-split on k, which the simplifier cannot perform. Had another induction scheme been applied, for example simultaneous induction on i and j, the proof would only require one case-split which could be performed while rippling. One approach to solve this problem was discussed in §5.5, where non-rippling goals, such as 5.15, were generalised and proved as lemmas. This technique did however not always terminate.

Another option is to modify the ripple measure to take wave-front size into account, allowing us to delay case-splitting until after regular rippling has finished. This is not possible with the current ripple-measures, as discussed in §5.2. In the example above this means rippling both the left- and right-hand sides of the goal, so it contains two case-statements. The two case-splits are then applied only when no more rewrite rules are applicable.

5.7 Related Work

5.7.1 Recursion Analysis

Case-splits on datatypes can be avoided by selecting or deriving a custom induction scheme, using recursion analysis [5]. The first-order systems ACL2 [49] and VeriFun [74] tackle problems otherwise needing case-splits in this way. Functions we define using case-statements are instead defined by recursion on several arguments, and recursion analysis can hence construct an appropriate induction scheme. However, for functions defined using case-statements, recursion analysis fails to derive the needed induction schemes.

5.7.2 The Case-Analysis Critic for CLAM

In the CLAM system [13], conditional functions would typically be defined using several conditional rewrite rules. For example, *member* would, in the non-empty case,

generate two rules:

$$x = h \Longrightarrow x\ member\ (h\ \#\ t) = True$$

$$x \neq h \Longrightarrow x\ member\ (h\ \#\ t) = x\ member\ t$$

If such a rule is applicable but the condition cannot be proved by simplification, and there exists another rule with a complementary condition, CLAM's case-split critic is triggered and introduces a split on the condition [41]. In Isabelle/IsaPlanner, functions with conditions are typically defined using an *if*- or *case*-statement, which is why our case-analysis technique works over these, rather than complementary conditional rules as in CLAM. As CLAM is first-order, it does not include case-statements and its case-analysis critic can therefore not perform the corresponding splits on datatypes. Of the 87 theorems in the evaluation corpus, CLAM could thus not have proved any of the 46 theorems involving functions defined using case-statements. CLAM was however capable of employing its lemma discovery critics to produce simple conditional lemmas, which IsaPlanner currently cannot do.

5.7.3 Case-splitting for Coq

An automated rippling-based inductive prover is under development in Coq, for dealing with proof obligations arising from programming with dependent types [76]. This prover follows a similar approach to ours, eagerly splitting on datatypes when possible.

5.7.4 Isabelle's Simplifier

Isabelle's simplifier applies rewriting with a set of given rules and can automatically split if-statements but not case-statements [65], §3.1.9. In general, splitting case-statements might cause non-termination for rewriting and is therefore not allowed. The user must therefore identify and insert case-splits in proofs, where required, or apply a different induction scheme, such as simultaneous induction on several variables.

Our case-analysis technique is incorporated as a step in rippling and can thus ensure termination even when splitting case-statements over datatypes is allowed. As long as the ripple measure decreases, splitting case-statements is safe. IsaPlanner employs the simple default structural induction schemes for datatypes. Using the case-analysis technique, IsaPlanner still manages to automatically prove theorems such as *(i - j) - k = i - (j + k)*, which in the interactive proof from Isabelle's library uses a custom induction scheme chosen by the user.

5.8 Summary

Performing case-splits is an important feature for an automated inductive theorem prover, as many common functions are naturally defined using if- and case-statements. Our case-analysis technique can perform the needed case-analysis in many of these proofs. Case-analysis has been incorporated as a step within rippling, and can thus retain termination even though splits on arbitrary datatypes are allowed. The technique is triggered during rippling whenever an if- or case construct is encountered in a goal. If it is possible to prove the associated condition, the technique proceeds down the corresponding branch, otherwise it introduces a split. This is implemented using resolution with a relevant library theorem, aided by our heuristic for restricting the number of unifiers.

The case-analysis technique has been fully implemented and tested in IsaPlanner. Our evaluation showed 47 new theorems that IsaPlanner is now able to prove automatically, including 14 which require splitting on a datatype, which is non-terminating for other types of rewriting techniques such as Isabelle's simplifier. Many of the more difficult theorems from the evaluation corpus require the ability to conjecture conditional lemmas and improved reasoning with assumptions, which we suggest as further work.

Chapter 6

Lemma Speculation

6.1 Introduction

Finding missing lemmas is a very challenging problem for automated theorem provers. The lemma discovery technique in this chapter has been designed to solve the problem of automatically proving theorems without having to supply a large corpus of library lemmas in advance, as is the case in many interactive systems, such as Isabelle.

Proof-planning critics were first introduced by Ireland et al. [41], as an attempt at finding missing lemmas in the context of inductive proofs using rippling. The critics were implemented in the CLAM 3 system. Rippling can provide crucial guidance in the search for missing lemmas. In particular, any lemma used in the rippling proof is required to decrease the differences between the goal and the inductive hypothesis, while at the same time preserving similar parts. These restrictions provide valuable hints as to what the missing lemma might look like, and make it feasible to attempt to automate lemma discovery.

Ireland presents two critics for lemma discovery in CLAM 3, *lemma calculation* and *lemma speculation*. The lemma calculation critic simply attempts to prove a generalised version of any goal remaining after the inductive hypothesis has been applied (see example in §3.4). Lemma calculation may be a simple technique, but has proved useful for many automated inductive proofs.

Lemma speculation is a more advanced technique, intended to discover lemmas in cases not covered by lemma calculation. This technique is thus typically applied when there are no more applicable rewrites but the inductive hypothesis cannot yet be applied. The central idea of the lemma speculation critic is to create a schematic lemma, initially containing meta-variables, that can unblock some part of the goal.

Subsequent ripple-rewrites applied to the goal are intended to help instantiate the meta-variables, until fertilisation is applicable and the meta-variables shared with the lemma have been fully instantiated.

The lemma speculation critic will construct a schematic lemma using a sub-term of the blocked goal as the left-hand side. There might be several such sub-terms, resulting in alternative lemmas. The right hand side of a schematic lemma is constructed by inserting meta-variables, standing for yet unknown structures into the skeleton of a blocked sub-term. This guarantees that the new lemma will preserve the skeleton, even though we do not yet know what it looks like. As a small example, consider a blocked goal of the form:

$$f(x,\ \boxed{g(\boxed{x})}^{\uparrow})$$

The lemma speculation critic creates a schematic lemma using this blocked term as the left-hand side. The right-hand side is created by inserting a meta-variable in the skeleton: $f(x,\ g(x)) = ?F(f(x,\ x))$, which rewrites the goal to

$$\boxed{?F(\boxed{f(x,\ x)})}^{\uparrow}$$

This lemma preserves the skeleton of the goal, and is thus allowed to be used by rippling. Further ripple steps are expected to help instantiate $?F$. Note however, that any instantiation where $?F$ is instantiated to something of the form $\lambda\ y.\ z$, is not allowed, as 'ignoring' the argument would break the skeleton.

Our work differs from that of Ireland in that we are focusing on lemma speculation in higher-order logic, and for dynamic rippling. CLAM 3 only supports static rippling and did not deal with higher-order theorems.

6.2 A Higher-Order Example

As a running example throughout the chapter, we use the inductive proof of the theorem:

$$\begin{aligned} \forall b\ n.\ foldl\ (\lambda\ x\ y.\ x + y)\ n\ ((rev\ a)\ @\ b) = \\ foldl\ (\lambda\ x\ y.\ x + y)\ n\ (b\ @\ a) \end{aligned} \tag{6.1}$$

The theorem states that if we are to compute the sum of two lists appended onto each other, it does not matter if one of them is reversed[1]. The function *foldl* is defined

[1] Here we start at some number n, this is to avoid the need for generalisation

in Appendix A. Its first argument is a function of two arguments (here +), which is applied to each argument of the list together with the value of the accumulator (here initially n), thus computing the sum of the list added to n.

We will use theorem 6.1 to illustrate how a schematic lemma is constructed from a blocked goal and how instantiations of its meta-variables are found. The step-case conclusion and inductive hypothesis are shown below:

$$\begin{aligned}\text{IH:}\quad & \forall b\ n.\ foldl\ (\lambda\ x\ y.\ x + y)\ n\ ((rev\ a)\ @\ b) = \\ & foldl\ (\lambda\ x\ y.\ x + y)\ n\ (b\ @\ a)\end{aligned}$$

$$\begin{aligned}\text{Step Case:}\quad & foldl\ (\lambda\ x\ y.\ x + y)\ \lfloor n' \rfloor\ ((rev\ \boxed{(h\#a)}^{\uparrow})\ @\ \lfloor b' \rfloor) = \\ & foldl\ (\lambda\ x\ y.\ x + y)\ \lfloor n' \rfloor\ (\lfloor b' \rfloor\ @\ \boxed{(h\#a)}^{\uparrow})\end{aligned}$$

To distinguish the universally quantified non-induction variables b and n in the hypothesis these have been renamed b' and n' in the step-case. n and n' are of type *nat* and a, b and b' have type *nat list*.

The following wave-rules from the definitions of *rev* and *foldl* are initially available to rippling:

$$rev(?h\ \#\ ?t) = (rev\ ?t)\ @\ [?h] \tag{6.2}$$

$$foldl\ ?f\ ?a\ (?h\ \#\ ?t) = foldl\ ?f\ (?f\ ?a\ ?h)\ ?t$$

After a ripple-step with rule 6.2 the step-case of our example proof becomes blocked:

$$\begin{aligned}foldl\ (\lambda\ x\ y.\ x + y)\ \lfloor n' \rfloor\ \boxed{((rev\ a)\ @[h])}^{\uparrow}\ @\ \lfloor b' \rfloor) = \\ foldl\ (\lambda\ x\ y.\ x + y)\ \lfloor n' \rfloor\ (\lfloor b' \rfloor\ @\ \boxed{(h\#a)}^{\uparrow})\end{aligned} \tag{6.3}$$

At this point, rippling is blocked but fertilisation is not yet applicable. Hence, the lemma speculation critic is triggered, as we shall see below.

6.3 Constructing a Schematic Equational Lemma

When rippling has become blocked before fertilisation, the lemma speculation critic is fired with the aim of constructing a lemma that will allow rippling to resume. The first step is to decide which sub-term a new lemma could attempt to unblock. There are typically several alternatives for choosing this sub-term, which will become the left-hand side of the lemma. Our heuristics for choosing a candidate sub-term for unblocking are outlined below:

1. The sub-term considered must contain at least one wave-front to unblock, as the speculated lemma must be used by rippling and thus decrease the measure.

2. The skeleton of the sub-term must contain at least one function symbol (applied to something) to be considered interesting, as we use the skeleton to construct the right-hand side of the lemma.

3. The sub-term, and its skeleton, is not allowed to contain any dangling bound variables (bound by a lambda higher up in the term tree), as this would produce a badly formed term.

Returning to the blocked goal 6.3, a number of alternative sub-terms are available. From the left-hand side of 6.3 we get the following candidates:

$$\begin{array}{l} \boxed{(\lfloor (rev\ a) \rfloor @ [h])}^{\uparrow} \\ \boxed{(\lfloor (rev\ a) \rfloor @ [h])}^{\uparrow} \ @ \ \lfloor b' \rfloor \\ foldl\ (\lambda\ x\, y.\ x + y)\ \lfloor n' \rfloor\ (\ \boxed{(\lfloor (rev\ a) \rfloor @ [h])}^{\uparrow} \ @ \ \lfloor b' \rfloor) \end{array} \tag{6.4}$$

The right-hand side of the blocked goal suggests another two possibilities:

$$\begin{array}{l} \lfloor b' \rfloor \ @ \ \boxed{(h \# \boxed{a})}^{\uparrow} \\ foldl\ (\lambda\ x\, y.\ x + y)\ \lfloor n' \rfloor\ (\lfloor b' \rfloor \ @ \ \boxed{(h \# \boxed{a})}^{\uparrow}) \end{array} \tag{6.5}$$

Some candidate sub-terms occur on both sides of the equation, but are only included once. Note that the sub-term $\boxed{(h\#\boxed{a})}^{\uparrow}$ is not suggested, as its skeleton only consists of the variable a. This would produce a lemma with right-hand side $?F(a)$, which is not particularly useful for middle-out rewriting.

Alternative annotations

As IsaPlanner uses dynamic rippling, there sometimes exist several annotations for the same term. The blocked goal 6.3 in the example has an alternative annotation, where the singleton list $[h]$ is considered to be in the sink position on the left-hand side, rather than b' as above:

$$\begin{array}{l} foldl\ (\lambda\ x\, y.\ x + y)\ \lfloor n' \rfloor\ \boxed{(\lfloor (rev\ a) \rfloor @ \lfloor\ [h]\ \rfloor) \ @ \ b'}^{\uparrow} = \\ \qquad\qquad foldl\ (\lambda\ x\, y.\ x + y)\ \lfloor n' \rfloor\ (\lfloor b' \rfloor \ @ \ \boxed{(h\#\boxed{a})}^{\uparrow}) \end{array}$$

This alternative annotation gives rise to two additional blocked sub-terms:

$$\boxed{(\boxed{(rev\ a)\ @\ \lfloor[h]\rfloor})\ @\ b'}^{\uparrow} \tag{6.6}$$

$$foldl\ (\lambda\ x\,y.\ x\ +\ y)\ \lfloor n'\rfloor\ (\ \boxed{(\boxed{(rev\ a)\ @\ \lfloor[h]\rfloor})\ @\ b'}^{\uparrow}\ )$$

Inserting meta-variables

After choosing a candidate sub-term for unblocking, which will make up the left-hand side of a schematic lemma, the right-hand side is constructed by inserting meta-variables into the *erasure* of the blocked term. The erasure is simply an instance of the skeleton with the sink instantiated to its current contents. As mentioned above, alternative annotations may produce alternative contents of sinks. Meta-variables are inserted above each function symbol, as well as in positions of sinks, where the contents of the sink become an argument to the meta-variable. Each meta-variable needs to be given some additional context parameters, here a, b', h and n', as these are not otherwise allowed to be used in meta-variable instantiations. For readability, parameters of meta-variables may be excluded in examples when it will not cause confusion.

To illustrate how to construct the right-hand side of a schematic lemma and why erasures are used, rather than the skeleton, consider the two alternative annotations 6.4 and 6.6. These will have different erasures and thus produce different schematic lemmas. Inserting meta-variables in the erasure of 6.4 gives the following schematic lemma[2] (the lemma is not annotated, following conventions for dynamic rippling):

$$((rev\ a)\ @\ [h])\ @\ b' = ?F(?F_1(rev\ a)\ @\ (?F_2\ b'\ a\ h\ n'))$$

Here the meta-variable F has been inserted above the @-symbol in the erasure, while $?F_1$ has been inserted above the *rev*, and F_2 in the sink position of b'. Note that the contents of the wave-front in the blocked term 'disappear' on the right-hand side of the schematic lemma, as it is replaced by meta-variables. The alternative annotation of the blocked term, 6.6, gives rise to a similar schematic lemma:

$$((rev\ a)\ @\ [h])\ @\ b' = ?F(?F_1(rev\ a)\ @\ (?F_2\ [h]\ b'\ a\ h\ n'))$$

Note that this version differs in that F_2 now also has the argument $[h]$, from the sink.

Neither of the above schematic lemmas will however lead to the discovery of an actual lemma, so for the purpose of continuing our example, we consider the schematic

[2] For clarity, we often omit the parameters a, b', h and n' to all meta-variables except those arising from sinks in the goal.

lemma arising from the blocked term 6.5:

$$foldl\ (\lambda\ x\, y.\, x\ +\ y)\ n'\ (b'\ @\ (h\#a))\ =$$
$$?F(foldl\ (\lambda\ x\, y.\, x\ +\ y)\ (?F_2\ n'\ a\ b'\ h)\ ?F_3((?F_4\ b'\ a\ h\ n')\ @\ a)) \quad (6.7)$$

We insert meta-variables in all possible places where a wave-front could be present, unlike Ireland's critic where only one speculative wave-front was considered at a time [41]. The search space should however be the same, with CLAM possibly having to explore many more schematic lemmas, but with fewer meta-variables in each.

6.4 Rippling and Instantiation of Meta-Variables

After applying a schematic lemma, the critic attempts to instantiate meta-variables using middle-out rewriting as described in §4.2. The search space for middle-out rewriting may still be very large. We use rippling to guide the search towards a goal where the meta-variables have been instantiated in such a way that fertilisation is possible. The applications of alternative lemmas are attempted starting with the lemma having the fewest meta-variables, and hence expected to have the smallest search space for middle-out rewriting.

New issues arise when considering rippling in the presence of meta-variables. It is no longer clear how ripple-measures should be computed, which has implications for the termination of rippling. Below we discuss our solution for retaining termination by repeatedly recomputing ripple-measures for the trace of middle-out rewriting steps. We also discuss the risk of an explosion in search space size when a large number of different possible annotations exist.

6.4.1 Potential Wave-Fronts and Measures

It is not obvious how the ripple-measure should be computed for a schematic goal. Because we insert meta-variables in all positions where a wave-front might occur, each meta-variable introduces a *potential wave-front*. This is a wave-front whose existence depends on the instantiation of the meta-variable. Potential wave-fronts are annotated by dashed boxes, and their wave-holes underlined. Potential wave-fronts arising from the introduced meta-variables might disappear if the meta-variable turns out to be a projection onto an argument in its wave-hole. As an example, consider the schematic

term $f(\boxed{?F\ \underline{g(x)}\ y}^{\uparrow})$ which contains one potential wave-front, introduced by the meta-variable $?F$. If $?F$ is instantiated to a projection on its first argument, the term becomes $f(g(x))$, and no longer contains any wave-fronts. If $?F$ is instantiated to something of the form $F \equiv \lambda u\ v.\ h(u,\ v)$, the potential wave-front becomes a regular wave-front: $f(\boxed{h(g(x),\ y)}^{\uparrow})$. The former instantiation results in a term with measure 0, while the latter has measure 1.

Now, recall the blocked goal 6.3:

$$foldl\ (\lambda\ x\ y.\ x\ +\ y)\ \lfloor n' \rfloor\ \boxed{((rev\ a)\ @ [h])}^{\uparrow}\ @\ \lfloor b' \rfloor) =$$
$$foldl\ (\lambda\ x\ y.\ x\ +\ y)\ \lfloor n' \rfloor\ (\lfloor b' \rfloor\ @\ \boxed{(h\ \#\ a)}^{\uparrow})$$

If we apply the schematic lemma 6.7 to the right hand side of the goal above we get a new goal containing several potential wave-fronts:

$$foldl\ (\lambda\ x\ y.\ x\ +\ y)\ \lfloor n' \rfloor\ \boxed{((rev\ a)\ @ [h])}^{\uparrow}\ @\ \lfloor b' \rfloor) =$$
$$\boxed{?F_1(\underline{foldl\ (\lambda\ x\ y.\ x\ +\ y)\ \lfloor (?F_2\ n'\ a\ b'\ h) \rfloor}\ \boxed{?F_3(\underline{\lfloor (?F_4\ b'\ a\ h\ n') \rfloor\ @\ a})}^{\uparrow})}^{\uparrow} \quad (6.8)$$

The blocked goal has ripple measure 4. If we were to compute the ripple measure as we normally do, goal 6.8 would not decrease the measure, as we have introduced two potential wave-fronts on the right-hand side, giving a naive ripple measure of 6. However, the possible meta-variable instantiations include projecting $?F_1$ and $?F_3$ onto their first arguments, and $?F_2$ and $?F_4$ onto n' and b'respectively, which would remove any wave-fronts from the right-hand side. This gives a best-case measure of only 2 for goal 6.8 (the measure of 2 comes from the wave-front on the left-hand side of 6.8). Hence, we consider the rewrite with the schematic lemma to be *potentially* measure decreasing and allow it, at least until the actual instantiations of the meta-variables are known. A schematic lemma is always constructed in such a way that each meta-variable will have some argument that is a sub-term of the skeleton. If instantiated to a projection on this argument, the meta-variable will contribute 0 towards the ripple-measure. Thus, each schematic lemma is initially potentially measure decreasing by construction.

As partial wave-rules are subsequently applied to the goal, some meta-variables will be instantiated. Recall that instantiations that would break the skeleton are not allowed. The instantiation of a meta-variable also changes its associated potential wave-front to an actual wave-front. After each instantiation, the ripple-measure must

therefore be recomputed for the whole sequence of middle-out rippling steps, as prior measures may be too optimistic. This ensures that a valid ripple trace still exists as a possibility. The new measures are again computed by considering projections of the current set of uninstantiated meta-variables. If the new ripple-trace still is potentially measure decreasing, the latest step is allowed, and rippling may continue. Otherwise, it is blocked.

Returning to our example, there are two possible middle-out rewrites that instantiate meta-variables and improve the ripple measure at this point, using wave-rules from the definitions of $foldl$ or *append* (see Appendix A). Here, we consider a wave-rule for $foldl$:

$$foldl\ ?f\ ?n\ (?h\ \#\ ?t) = foldl\ ?f\ (?f\ ?n\ ?h)\ ?t \tag{6.9}$$

Recall the middle-out rewriting algorithm described in §4.2. This will search for function symbols occurring both in the rule and in the schematic goal. The $foldl$-function occurs both in rule 6.9 and in goal 6.8. The redex in the goal does not have a top-level meta-variable, and it is hence safe to use regular unification and rewriting, rather than the version with heuristic restrictions described in §4.2. The rule application instantiates the meta-variable $?F_3$ to $\lambda\, x.\ (?h_1\ a\ b'\ h\ n')\ \#\ x$, which then beta-reduces to $(?h_1\ a\ b'\ h\ n')\ \#\ ((?F_4\ b'\ a\ h\ n')\ @\ a)$. Thus 6.8 is rewritten to[3]:

$$foldl\ (\lambda\ x\ y.\ x\ +\ y)\ \lfloor n' \rfloor\ \overbrace{(\boxed{(rev\ a)}\ @ [h])}^{\uparrow}\ @\ \lfloor b' \rfloor) =$$

$$\overbrace{?F_1(foldl\ (\lambda\ x\ y.\ x\ +\ y)\ \lfloor((?F_2\ n'\ a\ b'\ h)\ +\ (?h_1\ a\ b'\ h\ n'))\rfloor\ (\lfloor(?F_4\ b'\ a\ h\ n')\rfloor\ @\ a))}^{\uparrow}$$

To check if this is a valid ripple step, we first need to re-compute the ripple-measure for 6.8, as $?F_3$ has been instantiated. We then need to check that our new goal can potentially improve on the new measure of goal 6.8. The remaining meta-variables are instantiated by exploring their possible projection. Instantiating $?F_1$ to a projection onto its first argument, $?F_2$ to a projection onto n', F_4 to a projection onto b' and $?h_1$ to a projection onto h, gives the following valid ripple trace:

[3]The actual beta-expanded instantiation for $?F_3$ is $\lambda\, x\ y\ z\ u\ v.\ (?h_1\ \ a\ b'\ h\ n')\ \#\ x$. This is because $?F_3$ also takes the (omitted) parameters $a,\ b', h,\ n'$ as arguments.

$$foldl\ (\lambda\ x\,y.\ x\ +\ y)\ \lfloor n'\rfloor\ \big((rev\ a)\ @[h]\big)^{\uparrow}\ @\ \lfloor b'\rfloor) =$$

$$foldl\ (\lambda\ x\,y.\ x\ +\ y)\ \lfloor n'\rfloor\ (\lfloor b'\rfloor\ @\ (h\#a)^{\uparrow}) \quad \text{Measure: 4}$$

$$\Big\Downarrow\ by\ (6.7)$$

$$foldl\ (\lambda\ x\,y.\ x\ +\ y)\ \lfloor n'\rfloor\ \big((rev\ a)\ @[h]\big)^{\uparrow}\ @\ \lfloor b'\rfloor) =$$

$$foldl\ (\lambda\ x\,y.\ x\ +\ y)\ \lfloor n'\rfloor\ h\#(\lfloor b'\rfloor\ @\ a)^{\uparrow} \quad \text{Measure: 3}$$

$$\Big\Downarrow\ by\ (6.9)$$

$$foldl\ (\lambda\ x\,y.\ x\ +\ y)\ \lfloor n'\rfloor\ \big((rev\ a)\ @[h]\big)^{\uparrow}\ @\ \lfloor b'\rfloor) =$$

$$foldl\ (\lambda\ x\,y.\ x\ +\ y)\ \lfloor n'+h\rfloor(\lfloor b'\rfloor\ @\ a) \quad \text{Measure: 2}$$

$$\Big\Downarrow\ Weak\ Fert.$$

$$foldl\ (\lambda\ x\,y.\ x\ +\ y)\ n'\ ((rev\ a)\ @[h])\ @\ b') =$$

$$foldl\ (\lambda\ x\,y.\ x\ +\ y)\ (n'+h)((rev\ a)\ @\ b) \quad (6.10)$$

In addition to giving a valid ripple trace, these instantiations will allow for weak fertilisation to take place. Lemma 6.7 has been instantiated to

$$foldl\ (\lambda\ x\,y.\ x\ +\ y)\ n'\ (b'\ @\ (h\#a))\ = foldl\ (\lambda\ x\,y.\ x\ +\ y)\ n'\ h\#(b'\ @\ a) \quad (6.11)$$

By recomputing the ripple measures for the whole trace, we retain termination of rippling, even in the presence of meta-variables and potential wave-fronts. This follows from the standard termination property of rippling, and from the fact that the length of the potential middle-out rippling trace is limited by the measure of the last regular rippling goal. The last regular rippling goal does not contain any meta-variables, and has thus a fixed ripple measure, m. Every subsequent middle-out step has to have a measure less than m. The ripple measure is defined to be minimal, 0, when fertilisation is possible. Thus, the maximum number of possible middle-out rewriting steps is $m-1$, as each step has to reduce the measure from the previous step by at least 1.

When measures are recomputed for the whole trace, we ensure that a decreasing trace is still a possibility after each instantiation. Retaining termination of rippling with lemma speculation is a major improvement on the lemma speculation algorithm, compared to the CLAM 3 version. In CLAM 3, there were no checks for potential measure decrease of schematic goals, and termination was thus lost.

6.4.2 Multiple Annotations

As was pointed out in §3.4.3, using dynamic rippling means each goal may have several different annotations. The current version of IsaPlanner will, after each rewrite, compute all possible annotations for the whole of the new goal. Schematic goals may have an even larger number, if several of the meta-variable projections produce terms in which the skeleton can be embedded.

If each annotated copy of a schematic goal is treated separately, the rippling search space will increase exponentially, and quickly become unmanageable. The solution is to use a grouped ripple measure, as described in §3.4.3, where each goal has a list of possible embeddings and measures. Preliminary experiments showed that the search space of lemma speculation quickly became unfeasibly large without grouping the ripple measures. For example, without grouped measures, the proof in the previous section had a search space so large that the ML-process ran out of memory.

6.5 Eager Fertilisation

After each step of middle-out rewriting, our lemma speculation critic attempts to explore the projections of the remaining meta-variables to compute ripple measures and to check if fertilisation is applicable, in which case no more middle-out rewriting is needed.

If weak fertilisation is the last step of the critic, the result is two lemmas: the now instantiated lemma that was speculated by the critic and a generalisation of the goal after weak-fertilisation (a lemma calculation). In addition to the speculated lemma 6.11, the example above would also suggest the need for an additional lemma arising from the generalised post-fertilisation goal 6.10. In cases where strong fertilisation applies, only the speculated lemma needs to be proved.

Eager fertilisation is often applicable, but in many cases the meta-variables are instantiated in such a way that the schematic lemma becomes false. Therefore, the critic uses Isabelle's counter-example checker and only allows fertilisation if it cannot find a counter-example for the instantiated lemma.

After passing counter-example checking, a proof of the lemma is initiated. In some cases, the proof of the lemma itself will require further lemma speculations. This raises some concerns about the termination of repeated applications of lemma speculations, the prover might potentially apply a never ending chain of lemma speculations. An

alternative is to let the proof-planner inform the user that a proof of the original conjecture can be found, if supplied with a proof of the speculated lemma. The default for the current implementation is to attempt to prove any lemmas using the standard version of rippling with lemma calculation, but not speculation.

6.6 Evaluation

Lemma speculation is applicable when rippling is blocked, but fertilisation (and possibly lemma calculation) is not yet possible. Furthermore, for lemma speculation to work, there must be at least one additional ripple step after the schematic lemma has been applied to help instantiate the meta-variables by middle-out reasoning. Unfortunately, in the mathematical domains we have explored, there are few proofs failing in this way. In equational theories, weak fertilisation and lemma calculation are often applicable and successful. When surveying Isabelle's list library[4], which contains hundreds of theorems, we found only one inductive theorem to which lemma speculation was even applicable (theorem 8 in table 6.1 on page 73).

To find higher-order examples, we looked at the domain of higher-order function synthesis [21]. Here, higher-order equivalents of first-order ML functions were constructed using functions like *fold* and *map* in order to provide potential parallelisation. An example is $rev\ l = foldl\ \#\ [\ ]\ l$. Many of these proofs are hard and could not be proved automatically by IsaPlanner, even with the addition of lemma speculation. Lemma speculation is not applicable (or appropriate) to these proofs, as one side of the equation is typically very simple and will allow weak-fertilisation (here the left-hand side, $rev\ l$). To automate these proofs in IsaPlanner, lemma calculation with an improved generalisation technique seems more promising.

Exploring non-equational theorems from Isabelle's natural number library[5] about, for example, operators like $<$ and $\leq$, also gave few interesting examples. Proofs in this domain seem to often require conditional lemma discovery, as discussed in §5.6. For other theorems, such as $m \leq n \Longrightarrow m \leq (Suc\ n)$, which IsaPlanner currently fails to prove with its default induction scheme, induction on two variables simultaneously is required.

In the evaluation of critics in CLAM 3, only seven theorems required lemma spec-

[4] http://www.cl.cam.ac.uk/research/hvg/Isabelle/dist/library/HOL/List.html
[5] http://www.cl.cam.ac.uk/research/hvg/Isabelle/dist/library/HOL/Nat.html

ulation[6] [41]. Of these, theorem 1 in table 6.1, could equally well have been solved by generalising the variables apart. Most of the examples involve reversal of lists, where rippling can become blocked as a cons-operator (#) changes into an append of a singleton list.

In addition to the examples from CLAM 3, we have compiled an additional test set of (mostly) higher-order theorems. Due to the small number of theorems requiring lemma speculation, the test set is rather small and many of the examples are somewhat contrived as they had to be constructed in such a way that both sides of the equation got blocked. The results from our experiments are shown in the table 6.1 on page 73. The experiments were run on an Intel Xenon 2 GHz processor. We are mainly concerned with finding the correct lemmas in these experiments. Proof attempts of lemmas are only allowed to use lemma calculation, not additional speculations, for efficiency reasons. The lemmas given in the tables have been generalised by IsaPlanner, using common sub-term generalisation. We used a ripple measure which is a variant of the sum-of-distance measure, but which also keeps track of the path of the wave-fronts through the term tree, thus reducing the number of symmetric annotations. We have also experimented with using the plain sum-of-distance measure, but found that this largely gave the same results.

IsaPlanner manages to find *and* prove lemmas for 4 of Ireland's examples (1 and 4-6 in table 6.1) and one of the additional theorems (theorem 12). Lemmas are found for theorems 2 and 3, but were not automatically proved. However, the lemma for theorem 2 (which is identical to theorem 1) could have been proved if additional speculations were allowed in proofs of lemmas. The lemmas for theorem 3 require generalisation apart of the variables, which is not available in IsaPlanner. IsaPlanner fails to even find a lemma for theorem 7, because the missing lemma, $(a \mathbin{@} b) \mathbin{@} c = a \mathbin{@} (b \mathbin{@} c)$, is the last step before fertilisation. Thus, there is no chance of finding the instantiations for this lemma by middle-out ripple steps. For the higher-order examples, theorems 8-9 and 13 have the same problem. For theorems 10-11, lemmas are found, but not proved due to failure of lemma calculation.

[6] In Ireland's paper, an eighth theorem $sorted(sort\ l)$ is claimed to need lemma speculation, although it in fact requires a form of conditional lemma discovery described in §5.6, which is quite different from the critic described here.

	Theorem	Speculated lemma(s)	Proved lemma	Time
1	$x + (Suc\ x) = Suc(x + x)$	$Suc(x + y) = (Suc\ y) + x$	yes	6.247
2	$even(x + x)$	$x + (Suc\ x) = Suc(x + x)$	no	3.871
3	$even(len(l\ @\ l))$	$len(l\ @\ (h\#l)) = Suc(len(l\ @\ l)$ $len(l\ @\ (h\#l)) = len(h\#(l\ @\ l))$	no	13.476
4	$rev((rev\ l)\ @\ m) = (rev\ m)\ @\ l$	$rev\ (h\ \#\ l)\ @\ m = rev\ l\ @\ (h\ \#\ m)$	yes	8.041
5	$rev(rev(l\ @\ m)) = rev(rev\ l)\ @\ rev(rev\ m)$	$rev\ (l\ @\ [h]) = h\ \#\ rev\ l$	yes	0.232
6	$rev(rev\ l)\ @\ m = rev(rev(l\ @\ m))$	$rev\ (l\ @\ [h]) = h\ \#\ rev\ l$	yes	0.247
7	$rotate\ (len\ l)\ (l\ @\ m) = m\ @\ l$	fail	-	6.906
8	$rev(concat\ l) = concat(map\ rev\ (rev\ l))$	fail	-	8.450
9	$len(concat\ (map\ f\ l)) = len(maps\ f\ l)$	fail	-	1.529
10	$foldl\ (\lambda\ x\ y.\ y + x)\ n\ ((rev\ l)\ @\ m) =$ $foldl\ (\lambda\ x\ y.\ y + x)\ n\ (m\ @\ l)$	$foldl\ (\lambda\ x\ y.\ y + x)\ n\ (l\ @\ (h\ \#\ m)) =$ $foldl\ (\lambda\ x\ y.\ y + x)\ h\ (n\ \#\ (l\ @\ m))$	no	137.539
11	$foldl\ (\lambda\ x.\ y.\ x + (len\ y))\ n\ ((rev\ l)\ @\ m) =$ $foldl\ (\lambda\ x.\ y.\ x + (len\ y))\ n\ (m\ @\ l))$	$foldl\ (\lambda\ x\ y.\ x + (len\ y))\ n\ (l\ @\ (h\ \#\ m)) =$ $foldl\ (\lambda\ x\ y.\ x + (len\ y))\ n\ (h\ \#\ (l\ @\ m))$	no	33.909
12	$x \leq (y + x)$	$Suc\ z \leq a + (Suc\ z) = z \leq Suc(a + z)$	yes	43.743
13	$count\ n\ l \leq count\ n\ (l\ @\ m)$	fail	-	22.800

Table 6.1: Examples 1-7 come from Ireland. The remaining higher-order and inequality examples has been added by the author. Run-times are in seconds. The lemmas in the column 'Speculated lemma(s)' have been automatically speculated by our critic. The column 'Proved lemma' indicate whether IsaPlanner could prove the lemma automatically. To ensure termination, further lemma speculations were here not allowed in the proofs of lemmas.

6.7 Limitations

Several limitations of lemma speculation have become evident during our implementation and evaluation in IsaPlanner. We will summarise them below.

6.7.1 Applicability

As discussed in §6.6 we found surprisingly few theorems for which lemma speculation was applicable. Although we cannot with certainty claim that lemma speculation is as rarely applicable in domains other than those we have explored, it does seem like other techniques are more promising to explore. Lemma calculation is certainly far more widely applicable, which suggests that further work should be directed towards improving lemma calculation, extending it to cope with conditions and perhaps improve its generalisations.

6.7.2 Underspecified Lemmas

The lemma speculation critic relies on the assumption that it will be possible to instantiate the meta-variables by application of further wave-rules followed by the exploration of projections for remaining meta-variables in order to make fertilisation applicable. Problems arise when the lemma sought would take us directly from a blocked goal to a goal where fertilisation is applicable, without any intermediate ripple steps to help instantiate meta-variables. This happens for five of the theorems in the test set, namely theorems 7-10 and 14 in table 6.1.

Fertilisation will often be trivially applicable to the goal resulting after applying a newly speculated lemma, because of the way the lemma is constructed (see §6.3). Just projecting away the meta-variables can give an instance of the skeleton. However, this will almost always suggest a lemma that is not valid. Applying fertilisation directly after applying a schematic lemma, without any intermediate middle-out steps, is therefore not allowed in the current implementation. Ireland's version in CLAM 3 allows for direct fertilisation if the meta-variables are in sink positions [41]. The resulting lemma will however not be fully specified, as meta-variables remain. CLAM 3 attempts an inductive proof of the schematic lemma, expecting the base-case to allow the critic to find an instantiation for the remaining meta-variables. Theorem 7 in table 6.1 can be proved in this way, however, the technique will not work in general, and fails on the other theorems. In these cases one might consider using middle-out reasoning to try to

instantiate the meta-variables in the step case instead. This does however raise several issues. Firstly, preliminary work brought up a problem with Isabelle's induction tactic (used by IsaPlanner), which does not allow induction to be applied to goals containing meta-variables. Secondly, unlike middle-out rippling for lemma speculation, rippling could in this situation no longer guarantee termination as there are meta-variables both in the skeleton and in the goal. Middle-out rippling for lemma speculation terminates because the last blocked ripple-goal provides an upper bound on the ripple-measure of the subsequent steps. If the skeleton, and thus also the first step of rippling, contained meta-variables there would be no such upper bound. Instantiations of meta-variables could potentially introduce new meta-variables, thus increasing the initial ripple measure. Even if we recomputed the measures for the whole trace, ensuring a decrease, there is nothing stopping the initial starting measure to increase with each instantiation.

Example

As an example of a speculation that fails due to the required lemma being the last step before fertilisation, consider theorem 9 in table 6.1. It becomes blocked as:

$$len(\ \boxed{(f\ h)\ @\ concat(map\ f\ l)}^{\uparrow}\ ) = len(\ \boxed{(f\ h)\ @\ (maps\ f\ l)}^{\uparrow}\ )$$

The required lemma is the distributivity of *len* over append:

$$len(a\ @\ b) = (len\ a)\ +\ (len\ b)$$

If this lemma was available, the blocked goal above can be rippled to:

$$\boxed{len(f\ h)\ +\ len(concat(map\ f\ l))}^{\uparrow} = len(\ \boxed{(f\ h)\ @\ (maps\ f\ l)}^{\uparrow}\ )$$

Weak fertilisation is now applicable on the left-hand side. Hence, there are no ripple-steps that could have instantiated the lemma and the lemma speculation critic fails with an underspecified lemma.

6.7.3 Search Strategy

The search space for lemma speculation can still be very large, as some of our experiments show. A possible improvement would be to use a different search strategy, when trying the different alternative schematic lemmas. Currently, depth-first search is used, starting with the lemma that initially has the fewest meta-variables. However, during rewriting more meta-variables can be introduced, so even a small lemma might give

rise to a large search space. A better strategy might be some form of best-first search based on the number of meta-variables in the goal.

6.8 Related Work

Our work is an extension of Andrew Ireland's lemma speculation critic in CLAM 3 [41]. Unlike the critic in CLAM 3, IsaPlanner's critic is implemented for dynamic rippling and higher-order logic. This allows IsaPlanner to apply lemma speculation to higher-order theorems, unlike CLAM. Our critic will also terminate, thanks to the re-computation of ripple measures for the whole middle-out rewriting trace, which ensures each step is indeed measure decreasing. This also helps reduce the search space for middle-out rewriting. CLAM 3 did not have such restrictions and employed an iterative deepening search instead, which could not ensure termination. A further difference between our critic and CLAM's, is the heuristic for restricting which wave-rules are applicable during middle-out rewriting, described in chapter 4. As discussed in §4.4, CLAM uses static rippling and can thus make use of matching object-level annotations to restrict the number of wave-rules applicable to a schematic goal. IsaPlanner on the other hand, uses dynamic rippling, where rules are not annotated in advance. To avoid any rule matching the schematic goal, we only consider rules which share some function symbol with the goal.

IsaPlanner failed to find a lemma for one of the seven examples of lemma speculation given in [41] (theorem 7 in table 6.1), due to an underspecified lemma (see §6.7.2). CLAM managed to find a lemma by initiating an inductive proof of the underspecified lemma, where a meta-variable instantiation was found in the base-case. However, this technique does not work for many other proofs where lemma speculation fails to instantiate the lemma.

6.9 Summary

Lemma discovery is a very hard but vitally important problem in automated theorem proving. Lemma speculation is a technique used in conjunction with the rippling heuristic to suggest missing lemmas in inductive proofs. As opposed to lemma calculation, which is applied after fertilisation, lemma speculation is applicable when rippling is blocked but fertilisation is not yet possible. We have implemented a lemma speculation critic in the IsaPlanner proof planner.

Rippling requires any rewrite to preserve the embedding of the inductive hypothesis into the new goal. The lemma speculation critic takes advantage of this and constructs schematic lemmas by inserting meta-variables standing for yet unknown term-structures into the skeleton of a blocked term. The critic then attempts to instantiate the meta-variables by further rippling and middle-out rewriting. A restricted version of higher-order unification is used to help reduce the search space of applicable rewrite rules. Rippling helps to further reduce the number of candidate rewrites by ensuring that the ripple measure is still (potentially) decreasing over the whole trace after each instantiation of a meta-variable. After each rewrite the critic will also check if fertilisation has become possible for any term resulting from exploring the projections of any remaining meta-variables. If fertilisation is successful, the now instantiated lemma is subjected to counter-example checking, followed by an inductive proof-attempt.

Lemma speculation has some serious limitations. It will for example fail if fertilisation must be applied straight after application of the lemma, without any intermediate middle-out rewriting steps to help instantiate the meta-variables. Furthermore, it appears that lemma speculation is rarely applicable. We have surveyed a number of different theories, for example from Isabelle's libraries, and found very few proofs where it applies, unlike the simpler lemma calculation critic which is often successful.

Chapter 7

Conjecture Synthesis

7.1 Introduction

We have developed a program for synthesising inductive theorems, which we call IsaCoSy (**Isa**belle **Co**njecture **Sy**nthesis), as an alternative to lemma speculation. Experiments showed that lemma speculation is not applicable as often as expected, and in many of the cases where it is applicable, it fails to fully instantiate the lemma (see §6.7). This leads to an intractable problem of trying to synthesise terms for the uninstantiated meta-variables without any heuristic guidance from rippling and middle-out reasoning.

The problems with lemma speculation motivate our attempt to synthesise lemmas from available constants and variables in a 'bottom-up' fashion. We incrementally build larger terms using the set of available constants and function symbols in a given theory. The key idea for making this tractable is to turn rewriting upside down: only irreducible terms (not matching any rewrite rule) are synthesised. In terms of the implementation, these restrictions turn into constraints on the term-synthesis process, thus avoiding a naive and inefficient generate-and-test style procedure. Counter-example checking is still used to prune out obviously false conjectures, but as this can be rather slow, we want to use it as little as possible. Remaining conjectures are given to IsaPlanner to prove by induction and rippling. Any theorems found can then be used to generate further constraints as synthesis is attempted on larger terms, as well as being added as wave-rules, thus improving the power of IsaPlanner.

The aim of the IsaCoSy program is to automatically generate inductive theorems and lemmas that are interesting or will be useful as lemmas in further proofs in a given Isabelle theory.

The current implementation consists of three main parts:

- A language for expressing constraints on synthesis (§7.3).
- A constraint generator, which produces constraints from available theorems (§7.4).
- The synthesis engine itself, including procedures for updating and propagating constraints (§7.7)

Notation

Term Size

We define the size of a term to be the number of symbols (constants or variables) it contains. For example, the term $?h_1 = ?h_2$ is of size 3 (two variables and one constant, the '='-symbol).

Holes

During synthesis, *holes* are positions in the term-tree that have not yet been synthesised. Holes are implemented as meta-variables. Holes will have various constraints associated with them, such as a specified size and restrictions on which constants and variables are allowed to occur inside them.

Naming

Both holes and constraints are identified by unique names. We will use names of the form $?h_i$ for holes and C_i for constraints.

7.2 Motivating Examples

To illustrate the types of constraints that restrict term synthesis, we shall in this section consider a few examples about natural numbers. These examples are instances of the types of constraints that can be expressed in IsaCoSy's constraint language, described in §7.3, and used to control synthesis.

The constraints express restrictions on instantiations of holes, including which constants a hole is allowed to be instantiated to, and restrictions about which holes are not allowed to be instantiated to equal terms. Certain combinations of hole instantiations may also be forbidden. The constraint language must also be able to express

restrictions on holes that do not exist yet, but may in the future, given some particular instantiations of the current holes.

Example 1: Definition of Addition

Addition is defined as follows:

$$0 + y = y$$

$$(Suc\ x) + y = Suc(x + y)$$

The above definitions can be used as rewrite rules. The first applies to any term that has 0 in the first argument position of $+$, while the second applies to any term that has a *Suc* in the first position (regardless of what the *Suc* is applied to). We would like any such terms to be excluded by synthesis. Our constraint generation algorithm, described in detail in §7.4, will process the definitional theorems above. For the first theorem it produces a constraint stating that synthesis is never allowed to put a 0 in the first argument position of $+$. Similarly, for the second theorem, it generates a constraint disallowing *Suc* to appear in the first argument of $+$. This ensures that no term, which can be rewritten by the definitions of addition, is ever synthesised.

Example 2: Injectivity of *Suc*

Assume we know *Suc* to be injective, expressed in Isabelle as the rewrite rule $(Suc\ n = Suc\ m) = n = m$. To avoid synthesising terms to which this rewrite is applicable, we need a constraint that forbids the two arguments of $=$ to both be instantiated to *Suc* at the same time. Either one of them may be instantiated to *Suc*, but not both simultaneously. §7.5.2 describes these constraints more formally.

Example 3: Reflexivity

Reflexivity can be expressed as the rewrite rule $(x = x) = True$. The constraint we derive from this theorem is that the two arguments of $=$ never should be equal in a term we have synthesised. This is described further in §7.5.3.

Example 4: Conditional Constraints

Imagine we have a partially synthesised term, $Suc\ ?h_3 = ?h_2$, to which the reflexivity constraint (above) applies. The reflexivity constraint disallows the left- and right-hand side of the equation to be equal. The left-hand side has been partially instantiated to

Suc $?h_3$. Synthesis now only needs to consider the inequality constraint *if* the right-hand side hole $?h_2$ also becomes instantiated to *Suc*. The constraint propagation algorithm (§7.7.3) takes care of producing conditional constraints when needed.

Conditional constraints may also arise from rewrite rules. For example, if a rule has a left-hand side of the form $f(g\ 0)$, then *if* a synthesised term, containing an f, instantiates its argument to g, *then* 0 is not allowed to occur as an argument of g.

7.3 Constraint Language

Motivated by rewrite rules such as those in §7.2, we have developed a small language for expressing constraints on term synthesis. The constraint language allows us to capture the requirement that no synthesised term should be reducible by an existing rule.

7.3.1 Representation of Constraints

Each constraint comes from a rewrite rule about some top-level function[1]. When the constraint is derived it is given a unique name, and stored in a table associated with its top-level function. When a particular function-symbol is used during synthesis, its associated constraints are attached to the new holes.

When we talk about *arguments* in constraints, we refer to a particular argument position for a function, which can be a hole during synthesis. Other representations of arguments are described in §7.3.2. Some constraint-types refer to several arguments, and may thus be attached to more than one hole during synthesis.

The constraint language consists of five different types of constraints, captured by the datatype:

```
datatype Constr =
  NotAllowed of Arg * ConstantName
| VarNotAllowed of Arg * VarName
| NotSimult of (Arg * ConstraintName)  list
| UnEqual of Arg list
| IfThen of Arg * (ConstantName * ConstraintName)
```

The first two constructors of the constraint-type, *NotAllowed* and *VarNotAllowed*, simply state that some argument is not allowed to be instantiated to some (uniquely

[1] This is the top-level function in the left-hand side of the rule

named) constant or variable (see Example 1 of §7.2). The *NotSimult*-constraint captures dependent constraints. Several arguments may not be allowed to have a particular combination of instantiations simultaneously (see Example 2 of §7.2). The *UnEqual*-constraint specifies a list of arguments that are not all allowed to be instantiated to the same term (some of the arguments may have the same instantiation, but not all). An example of this is the reflexivity theorem in Example 3 of §7.2. Finally, the *IfThen*-constraint describes a condition under which a constraint on a future hole should be considered. If the argument of the *IfThen*-constraint is instantiated to the specified constant, the resulting new holes will have to adhere to the named constraint (Example 4 of §7.2).

We believe this language to be sufficient to capture constraints from standard equational rewrite rules. However, the constraint language is not designed to capture constraints from rules with side-conditions and rules containing lambda expressions, these kinds of rules would require additions to the constraint language above. Conversely, our synthesis algorithm does not attempt to synthesise conjectures with side conditions, or conjectures containing lambda-expressions. Techniques for finding the correct side-condition to make an inductive conjecture true have been studied in [63], but extending this technique to synthesis is left as further work. Allowing lambda-expressions in synthesised conjectures is equivalent to allowing synthesis of new functions, which would increase the search space size. We leave function synthesis as further work and here restrict ourselves to only synthesising terms about existing functions.

7.3.2 Representation of Arguments

Each constraint talks about one or several arguments of some function. Arguments are represented differently at different stages of the synthesis process. The following data-type is used for representing arguments:

```
datatype Arg =
  Hole of HoleName
| Path of int list
| LocalIndex of ConstraintName * int
```

Arguments of *Hole*-type, have already been described in §7.1, so we will here introduce the remaining two constructors.

The *Path* constructor is only used temporarily when analysing a new theorem for constraints. It specifies a position in a term as a path from the top of the term-tree. For example, in the term $(a * b) + c$, the variable b has the path $[1,\ 2]$, as it is the first (leftmost) argument of plus, and the second argument of multiplication. Variable a has path $[1,\ 1]$, while the path of c is just $[2]$.

The *LocalIndex* constructor is used to represent future constraints on some hole that does not yet exist, but may in the future. This is also how arguments are initially represented in constraints generated from rewrite rules, which are computed and stored prior to synthesis, when no holes exist. As synthesis proceeds, arguments represented using *LocalIndex* are gradually replaced by holes. The constraint name in a *LocalIndex* indicates the name of the constraint that has to be triggered in order for the *LocalIndex* to be updated to a *Hole*-type. This is either the parent-constraint of the constraint in which the *LocalIndex* occurs[2] or, if the constraint has no parent, itself. Several new holes may be produced at the same time, so the integer-index part of a *LocalIndex* indicates which new hole is intended. We abbreviate an argument *LocalIndex(C_i, j)* to $C_i.j$ in order to improve readability.

As an example illustrating the use of *LocalIndex*-constraints, assume we are synthesising an equality, and initially have a term with two holes:

$$?h_1 = ?h_2$$

Also suppose there are two constraints, C_1 and C_2, (from the zero-case of the definition of addition) with C_1 attached to $?h_1$:

$$\begin{aligned} C_1: &\quad IfThen(?h_1,\ \text{`plus'},\ C_2) \\ C_2: &\quad NotAllowed(LocalIndex(C_1, 1)\ \text{`zero'}) \end{aligned}$$

The constraints above state that if $?h_1$ is instantiated to plus, the first of the resulting new holes is not allowed to be instantiated to zero. Note that C_2 must use the *LocalIndex*-constructor for its argument, as the first argument position of plus does not yet exist as a named hole. Constraint C_1 must be triggered for such a hole to be created. This happens if h_1 is indeed instantiated to plus, resulting in the new term:

$$?h_3 + ?h_4 = ?h_2$$

[2] If this constraint is a sub-constraint of a *NotSimult*, its arguments are named after the 'grandparent'-constraints, otherwise a *LocalIndex*-name might not be unique.

The new holes are named $?h_3$ and $?h_4$, with $?h_3$ being in the first (leftmost) argument position of plus, thus instantiating constraint C_2:

$$C_2: \quad NotAllowed(?h_3, \text{`zero'})$$

7.4 Generating Constraints

The constraints used during synthesis are automatically inferred from equational theorems. Initial constraints are derived from the definitions of recursively defined functions, as well as from theorems about reflexivity and commutativity of equality and theorems about datatypes that Isabelle's datatype-package proves automatically. This section describes the constraint generation algorithm, and shows how it derives constraints from rewrite rules.

7.4.1 Constraints and Information about Functions

To initialise synthesis, we compute some relevant information about each function. This includes:

- The type of the function and each of its arguments.
- A domain for each argument, specifying which constants are allowed to occur in that position. The domain is initially all the symbols with a matching type, and is later restricted by constraints from rewrite rules.
- A set of constraints for each of the function's arguments, arising from the initial rewrite rules.
- Information about whether the function is known to be commutative and/or associative. This is updated as synthesis progresses, as the relevant theorems are discovered. If a function is known to be commutative, we can further restrict synthesis by imposing an order on its arguments. For example, always requiring the first argument to be of larger or equal size than the second, according to some measure on terms.

The above information is stored in a table indexed by the function-symbol's unique name. As synthesis proceeds, and more theorems are proved, these can be fed back into the constraint generation mechanism to produce more constraints on future synthesis attempts.

7.4.2 Constraint Generation Algorithm

The constraint generation algorithm infers constraints from the left-hand sides of rewrite rules. The algorithm traverses the left-hand side term top down, producing a set of constraints that will be attached to the top-level symbol of the left-hand side of the rewrite rule. As a running example, consider a rewrite rule with left-hand side: $f\ ?a\ ?a\ (g\ 0) = \ldots$.

Overview of the Algorithm

1. Traverse the term and find its left-hand side (LHS). In the example, the left-hand side of the rule is $f\ ?a\ ?a\ (g\ 0)$.

2. Create equality constraints. Positions of variables that occur several times may not be allowed to be instantiated to the same term. In the example, $f\ ?a\ ?a\ (g\ 0)$, we need to consider disallowing the first and second argument of f to be the same, as the variable $?a$ occurs in both these positions. To find variables, traverse the LHS top down, keeping track of the path taken. On encountering a variable, store its name and path in a table. For those variables that have more than one path, create an *UnEqual*-constraint, e.g. $\mathit{UnEqual}(\mathit{Path}(p_1)\ \ldots\ \mathit{Path}(p_n))$.

   For the rule in the running example, there are two occurrences of the variable $?a$, in the first and second argument position of f. Using the *Path*-constructor from §7.3.2, the two occurrences of $?a$ are represented as $\mathit{Path}[1]$ and $\mathit{Path}[2]$. These are not allowed to be equal, so we store a constraint: $\mathit{UnEqual}(\mathit{Path}[1], \mathit{Path}[2])$.

3. If the LHS term is a function application, $f(x_1\ \ldots\ x_n)$, compute constraints of its argument terms $(x_1\ \ldots\ x_n)$. If not, there are no constraints.

   In the running example, the LHS of rule is a function application, so we proceed to compute constraints for the arguments of f.

   Constraints for an argument-term x_i are computed depending on whether the argument-term is a variable, constant or function application. In the constraint, the argument is referred to by its position, and parent-constraint name, using the *LocalIndex*-constructor.

   Returning to the example, assume we give the name C_0 to the top-level constraint we are constructing for f. The arguments of f will thus be in positions named

$C_0.1$ (for $?a$), $C_0.2$ (for the second occurrence of $?a$) and $C_0.3$ (for g 0). Computing the constraints for the arguments now proceeds as follows depending on the type of the argument:

- **Variable** v**:** Look up the variable name v in the table created in the step 2, to check if it is involved in any *UnEqual*-constraint. If so, the argument-type is updated to use a *LocalIndex* instead of a *Path* as we now know the name of its parent constraint.

  In the running example, both the first and second arguments are indeed variables, and are involved in the *UnEqual*-constraint created in step 2. We give this constraint the name C_1 and update it to:

  $$C_1 : \quad UnEqual(C_0.1,\ C_0.2)$$

  When the algorithm terminates, all arguments will be in *LocalIndex*-format.

- **Constant** c**:** Create a new *NotAllowed*-constraint for this argument and constant: $NotAllowed(C_p.i,\ c)$

- **Function application** $g(y_1\ \ldots\ y_m)$**:** Recursively compute the constraints of the arguments. If the number of constraints on the arguments is:

  - Greater than 1, i.e. a list $(arg_j,\ C_j)\ \ldots\ (arg_n,\ C_n)$. Create a *NotSimult*-constraint:

    $$NotSimult((arg_j,\ C_j)\ \ldots\ (arg_n,\ C_n))$$

  - Exactly 1, i.e. (arg_j, C_j). Create an *IfThen*-constraint:

    $$IfThen(C_p.i,\ g,\ C_j)$$

  - 0, i.e. all arguments are variables occurring once in the term. Create a *NotAllowed*-constraint for this argument and function-symbol:

    $$NotAllowed(C_p.i,\ g)$$

  In the example, the third argument of f is a function application g 0. To compute a constraint on this, which we shall name C_2, we first compute a constraint for the single argument of g, the constant 0. This gives the constraint $C_3 : NotAllowed(C_2.1,$ 'zero'$)$. Hence we also get $C_2 : IfThen(C_0.3,$ 'g', $C_3)$. Informally, this means that if the position $C_0.3$ is instantiated to g, the first argument of g is not allowed to be 0.

4. When the argument-constraints are computed, we can determine the top-level constraint. As before it becomes a *NotSimult*-constraint if there are several argument-constraints. If there is only one, and that is a *NotAllowed*-constraint, this constant can simply be removed from the domain of the relevant argument position.

   In the example, the complete set of constraints for the three arguments of f are:

   $$C_1 : \ UnEqual(C_0.1,\ C_0.2)$$

   $$C_2 : \ IfThen(C_0.3,\ \text{'g'},\ C_3)$$

   $$C_3 : \ NotAllowed(C_2.1,\ \text{'zero'})$$

   This results in the top-level constraint, C_0, becoming:

   $$C_0 : \ NotSimult((C_0.1,\ C_1),\ (C_0.2,\ C_1),\ (C_0.3, C_2))$$

   This constraint captures that synthesis is not allowed to simultaneously violate both constraints C_1 and C_2, while synthesising a term containing the function f.

5. The final step of the constraint generation algorithm is to store all the constraints in the constraint-table for the top-level function symbol on the LHS (in the example, f). We also store the name of the top-level constraint (here C_0). These will later be used if synthesising a term containing the function f.

7.5 Sources of Initial Constraints

The initial constraints given to the synthesis machinery are derived from function definitions, datatype theorems and other library theorems. Below, we give examples of each type, and show the constraints that are derived from them.

7.5.1 Constraints From Function Definitions

Function definitions provide an important source for initial constraints for synthesis. Recall the definition of natural numbers:

$$0 + y = y$$

$$(Suc\ x) + y = Suc(x + y)$$

Following the algorithm in §7.4.2, the first definitional theorem has the left-hand side $0 + y$. This is indeed a function application (of plus to 0 and y), so we proceed to compute constraints on the arguments. 0 is a constant, so generates a *NotAllowed*-constraint on the first argument of $+$. y only occurs once, so it does not contribute to any constraints. As the *NotAllowed*-constraint for 0 is the only constraint, 0 can be removed from the domain of the first argument of plus.

For the second theorem, with left-hand side $Suc\ x + y$, the first argument is a function application ($Suc\ x$), but its argument does not produce any constraints. Hence we again get only a single *NotAllowed*-constraint, this time forbidding *Suc* to occur as the first argument.

In general, theorems from function definitions will restrict the domain of the argument(s) on which the function is recursive.

7.5.2 Constraints From Datatype Theorems

Isabelle's datatype package will automatically derive a number of useful theorems when a new datatype is defined. These will typically be used to provide constraints on equalities. Our program automatically uses the injectivity and so called distinctness theorems (if available) for each datatype to provide synthesis with useful constraints.

Returning to our running example of natural numbers, Isabelle derives an injectivity theorem (§7.2, Example 2):

$$((Suc\ n) = (Suc\ m)) = (n = m)$$

It also derives a so called distinctness theorem[3]:

$$(Suc\ n = 0) = False$$

From injectivity, we can derive constraints stating that the first and second arguments of an equality are not simultaneously allowed to be instantiated to *Suc*:

$$\begin{aligned} C_1: &\quad NotSimult((C_1.1, C_2),\ (C_1.2, C_3)) \\ C_2: &\quad NotAllowed(C_1.1,\ \text{`Suc'}) \\ C_3: &\quad NotAllowed(C_1.2,\ \text{`Suc'}) \end{aligned}$$

[3]Isabelle actually derives a slightly different variant, of the form $Suc\ n \neq 0$, which IsaCoSy uses to derive an equivalent theorem suitable for our constraint derivation algorithm. There is also a commuted version of this theorem, $(0 = Suc\ n) = False$, but this version is not necessary for our constraint generation as we know $=$ is commutative (see §7.5.3)

The distinctness theorem forbids the two arguments of an equality being instantiated to opposite constructors:

$$\begin{aligned} C_1: &\quad \mathit{NotSimult}((C_1.1, C_2),\ (C_1.2, C_3)) \\ C_2: &\quad \mathit{NotAllowed}(C_1.1,\ \text{`Suc'}) \\ C_3: &\quad \mathit{NotAllowed}(C_1.2,\ \text{`zero'}) \end{aligned}$$

7.5.3 Reflexivity: Equality Constraints

Recall the reflexivity theorem from Example 3 of §7.2: $(x = x) = \mathit{True}$. To avoid this being applicable as a rewrite rule, we do not want to synthesise any terms with identical left- and right-hand sides. This results in an equality constraint on the two arguments of the equality:

$$C_1: \quad \mathit{UnEqual}(C_1.1,\ C_1.2)$$

Of course, it might not be possible to establish that the two sub-terms are indeed different until the whole term is fully synthesised. If the two arguments become instantiated to different top-level symbols, the constraint can be dropped. Otherwise, the equality is broken down into sub-constraints on new holes appearing after instantiation. Unlike the constraints from injectivity and distinctness in the previous section, we do not know in advance how many levels down the term tree we might have to look before an equality constraint, such as reflexivity, can be dismissed.

7.5.4 Commutativity: Argument Order Constraints

Commutativity theorems are used to avoid symmetries in synthesis. If we know that a function is commutative, we can impose an order on its argument, for example always require the leftmost argument to be of greater or equal size. IsaCoSy has initially access to the commutativity theorem for equality:

$$(x = y) = (y = x)$$

For an equality, $?h_1 = ?h_2$, we impose the constraint that the size of $?h_2$ is always smaller or equal to the size of h_1, allowing us to cut the search space in half[4].

Size constraints are currently not expressed in the constraint language described above, but attached to holes during synthesis for ease of implementation.

[4] Some symmetries will however remain, when both sides of the equation have the same size.

Although only the commutativity theorem for equality is currently given at the start, other commutativity properties will be found during synthesis and can be identified and then used in a similar fashion.

7.6 Additional Heuristics

We also make use of some additional heuristics to constrain synthesis which are not directly derived from rewrite rules.

7.6.1 Variable occurrence

A common heuristic for equational rewriting is to only allow rules where the variables in the right-hand side are a subset of the variables on the left. For example, $f(x,\ y) = x$ is a valid rewrite rule, but $x = f(x,\ y)$ is not. As we are interested in synthesising valid rewrite rules, the default settings for IsaCoSy is to only allow holes in the left-hand side to be instantiated to fresh variables, while variables on the right-hand side are only allowed to be picked from those already occurring on the left. For example, if we have the following partially synthesised term, $f(x,\ y) = ?h$, the only variable-instantiation allowed for $?h$ is x or y.

7.6.2 Number of Variables Allowed

IsaCoSy allows the user to specify how many different variables should be allowed to occur in the synthesised terms. In many common theories, such as lists or natural numbers, the interesting theorems often have no more than two or three variables. Studying the theorems in Isabelle's libraries[5] for natural numbers and lists, suggests that a good default heuristic for the number of different variables is 1 + the maximum arity of any function involved. While restricting the number of variables obviously may cause theorems to be missed, it is very useful in reducing the search space.

7.6.3 Eager Check for Associativity and Commutativity

Another option for synthesis is to eagerly check functions for associativity and commutativity properties, prior to synthesis. If the AC-option is switched on, any binary

[5] `www.cl.cam.ac.uk/research/hvg/Isabelle/dist/library`

function, with arguments of the same type, is plugged into associativity and commutativity templates, of the forms $f\ (f\ x\ y)\ z = f\ x\ (f\ y\ z)$ and $f\ x\ y = f\ y\ x$ respectively. The resulting conjectures are passed through counter-example checking followed by a proof attempt in IsaPlanner. Should the proof of commutativity succeed, we can impose ordering restrictions on the function's argument during synthesis (recall §7.5.4). Furthermore, if the function is commutative, it is likely that the commuted variants of its definitional theorems will be useful to our prover, so these are also added to the set of synthesised terms. For example, the commuted definitions of plus (defined as in §7.2) give us the two theorems[6]:

$$y\ +\ 0 = y$$

$$y\ +\ (Suc\ x) = Suc(y\ +\ x)$$

7.7 Synthesising Conjectures

After the initial constraints of the current theory have been computed, synthesis can start. The synthesis algorithm is given a table of current constraints for the available function symbols, along with a specified top-level symbol, in our case equality. It is also given a maximum size limit, and will synthesise terms from the minimum size possible given the top-level constant, up to the limit. We may optionally specify a customised minimum size, should we wish to do so. At each iteration, non-theorems are filtered out by counter-example checking. Conjectures that can be proved are fed into the constraint generation mechanism to provide further constraints, thus restricting the search space when synthesising larger terms. We have also experimented with allowing constraints to be generated from terms that pass counter-example checking, but IsaPlanner fails to prove (see §8.6).

7.7.1 A Data-Structure for Synthesis

The algorithm uses a data-structure we call *STerm* to keep track of information related to the current synthesis attempt. This data-structure contains:

- The term synthesised so far.
- The name, type and size of each uninstantiated hole.

[6]These are obtained by commuting both the left- and right-hand sides. Commuting only one side is also an option, which would give another two versions, but these are less useful to the prover.

- A table of current constraints, indexed by their unique names.
- The constraints associated with each hole.
- The domain of allowed constants that each hole potentially can be instantiated to.
- Constraint dependencies, keeping track of parent-constraints where relevant.

As synthesis progresses, constraints will be evaluated and either dropped (if they no longer apply) or refined to provide restrictions on new holes.

7.7.2 Overview of the Algorithm

IsaCoSy starts from some specified minimum size and performs one iteration of the algorithm below for each size, up to the given maximum size.

1. Initialise synthesis by importing the constraints associated with the given top-level function (for example, equality). Also compute the allowed size combinations for the holes.
2. Pick the next hole to be instantiated from the search agenda. The current version of IsaCoSy uses depth-first search, but the synthesis machinery is compatible with other search strategies.

   Depending on the size associated with the next hole, instantiate:

   - If *hole-size = 1*: Instantiate the hole to a constant of size 1 (e.g. the constant 0 for natural numbers or the empty list), or to a variable. Variables can either be fresh or already exist in the term. Existing variables are filtered against *VarNotAllowed*-constraints on the hole, in case they are forbidden in this position.

     If synthesising an equality, fresh variables are typically only allowed in the left-hand side. Similarly, if we have chosen a maximum number of different variables, fresh variables are only allowed as long as this limit has not been exceeded.
   - Else, *hole-size* $>$ 1: Instantiate the hole to a function with arguments providing new holes. Consider all function-symbols in the domain of the hole that have a minimum term size satisfying $1 <$ *min-size* $\leq$ *hole-size*.

3. Update and propagate constraints, given the instantiation and possible new holes (see §7.7.3).

4. Terminate when there are no more open holes. Filter the resulting terms through the counter-example finder.

5. Attempt to prove remaining conjectures.

6. Feed new theorems to the constraint generation algorithm to produce additional constraints before the next iteration for synthesising larger terms.

Note that IsaCoSy's synthesis algorithm is independent of the order in which holes are instantiated. The term-synthesis machinery and constraint language is designed to allow experimentation with different search strategies. This also allows implementation of additional heuristics to exploit particular search strategies. An example is the variable occurrence heuristic from §7.6, which works with depth-first search. As a possible future extension, the synthesis machinery could also be used for synthesising specific terms of interest, starting from a partially instantiated term. For example, the user may specify a template term containing some holes, e.g. $rev(?h_1) = ?h_2$, should he/she wish to only generate equational theorems with *rev* on the left-hand side.

7.7.3 Constraint Propagation

The constraint propagation mechanism is crucial for the synthesis algorithm's efficiency. Our constraint language supports expressing future constraints, depending on instantiations of current holes. These constraints need to be updated and propagated to any new holes. In particular, we need to manage the propagation of equalities, which will break up into several new constraints as holes are instantiated. Constraint propagation will also have to take dependencies into account, by checking if constraints are part of a *NotSimult*.

Constraint Propagation Algorithm

Assume the hole $?h$ has been instantiated to some symbol s, and has an attached constraint, named C. The constraint C might be a sub-constraint of some other constraint, C_p, which must be of *NotSimult*-type, as this is the only type of constraint that talks about the same holes as its sub-constraints[7]. Below, we use *const* to stand for some

[7] *IfThen*-constraints on the other hand, only have sub-constraints that talk about possible future holes.

arbitrary constant, and v to stand for a named variable. We say that a constraint is *satisfied* when some hole it refers to is instantiated in such a way that the constraint no longer applies, e.g. the constraint $VarNotAllowed(?h,\ v)$, is satisfied when $?h$ is instantiated to anything other than v. A constraint is *violated* when the instantiation is contrary to what is specified, e.g. if $?h$ *is* instantiated to v. This is allowed if the constraint is part of a larger *NotSimult*-constraint, which specifies a set of constraints that may not *all* be violated simultaneously.

Depending on the type of the constraint C, the following updates are made:

$NotSimult[(arg_1,\ C_1),\ \ldots,\ (arg_i,\ C_i),\ \ldots,\ (arg_n,\ C_n)]$**:** Assuming $arg_i = ?h$, the constraint propagation function is called on the sub-constraint C_i, associated with the hole $?h$ that is being instantiated. In processing C_i, the fact that it is part of a *NotSimult* must be taken into account. If the constraint expressed by C_i has been satisfied, and it no longer applies, its parent *NotSimult*-constraint is also satisfied, and can be dropped. On the other hand, if C_i is violated or replaced by a sub-constraint, its parent and sibling constraints must remain. The *NotSimult* parent-constraint is thus updated as follows:

- C_i is satisfied: Delete C_i, along with its parent *NotSimult*-constraint (and sibling-constraints).
- C_i is violated: Delete C_i from its parent. The initial constraint thus becomes:

  $NotSimult[(arg_1,\ C_1),\ \ldots,\ (arg_{i-1},\ C_{i-1}),\ (arg_{i+1},\ C_{i+1}),\ldots,\ (arg_n,\ C_n)]$.

  Check that there is still more than one constraint in the parent, otherwise there is no need for a *NotSimult* constraint, and it can be replaced by its last child-constraint. For example, if we get $NotSimult[(arg_j,\ C_j)]$, it is sufficient to keep the constraint C_j on its own.
- C_i is replaced by its sub-constraint(s), C_i': The sub-constraint will be attached to some new hole, $?h'$. We replace C_i by C_i' and let $arg_i' = ?h'$ in the parent constraint, e.g. $NotSimult[(arg_1,\ C_1),\ \ldots,\ (arg_i',\ C_i'),\ \ldots,(arg_n,\ C_n)]$. If C_i' also happens to be a *NotSimult*-constraint, it is merged with the parent constraint.

$NotAllowed(?h,\ const)$**:** This constraint should only occur if it is a sub-constraint of a *NotSimult*. Otherwise *const* would have been removed directly from the domain of $?h$ and the constraint dropped.

Hence, C must be a sub-constraint of a *NotSimult*-constraint, C_p. Assume the hole $?h$ is instantiated to s:

- If $s \neq const$: The constraint is satisfied and C_p and all its sub-constraints can be dropped (as described above).
- Else, $s = const$: The constraint is violated, we may drop C, but C_p must remain and is updated as described above for *NotSimult*-constraints.

$VarNotAllowed(?h,\ v)$**:** If considering instantiating a hole with a variable, this constraint is checked at instantiation, ensuring the hole is not instantiated to v. If it is a sub-constraint of a *NotSimult*, the process is analogous as above for *NotAllowed*.

$IfThen(?h,\ s',\ C_j)$**:** Assume the hole $?h$ is instantiated to s:

- If $s \neq s'$: The constraint is satisfied, and can be dropped along with its sub-constraint, C_j. If any parent constraint exists, this is updated accordingly.
- Else, $s = s'$: The sub-constraint C_j must be considered. C_j should be attached to some new hole(s). To determined which one(s), the argument-types in the sub-constraint C_j are updated from *LocalIndex*-type to *Hole*-type, as described in §7.3.1. The $IfThen$-constraint C is then deleted.

  If C has a parent *NotSimult*-constraint C_p, C is replaced by C_j in the parent, as described above. Otherwise, if the sub-constraint C_j turns out to be of type *NotAllowed*, the domain of the relevant hole is updated accordingly, before C_j is deleted.

$UnEqual[?h,\ arg_1\ \ldots arg_n]$**:** Recall that *UnEqual*-constraints express that a set of holes cannot all have the same instantiation (although some may). *UnEqual*-constraints always break down into further constraints, until eventually disallowing particular variables or constants for the last hole left to be instantiated.

Equality constraints are updated differently depending on the instantiation of $?h$:

- Variable v: The other arguments of the equality are not simultaneously all allowed to be instantiated to the same variable v as $?h$ was instantiated to.

The following new constraints are added to express this[8]:

$$\begin{aligned} C_j: &\quad NotSimult((arg_1,\ C_{arg_1})\ \ldots\ (arg_n,\ C_{arg_n})) \\ C_{arg_1}: &\quad VarNotAllowed(arg_1,\ v) \\ &\quad\ \vdots \\ C_{arg_n}: &\quad VarNotAllowed(arg_n,\ v) \end{aligned}$$

Finally, the original equality, C, is dropped.

- Constant c: As for variables but with a $NotAllowed(arg_i,\ c)$ for each of the remaining arguments.
- Function $f(?x_1 \ldots ?x_m)$: If $?h$ is instantiated to a function with new holes, $?x_1 \ldots ?x_m$, being created, we must create future constraints on the potential instantiations for the other arguments, $arg_1\ \ldots arg_n$, of the *UnEqual*-constraint.

  The *UnEqual*-constraint can only be violated if the other arguments also are instantiated to f, and the argument-positions of f are instantiated to the same symbol everywhere. We thus first create the following new equality constraints on the new holes $?x_1 \ldots ?x_m$:

$$\begin{aligned} C_{x_1}: &\quad UnEqual[?x_1,\ C_{arg_1}.1\ \ldots\ C_{arg_n}.1] \\ &\quad\ \vdots \\ C_{x_m}: &\quad UnEqual[?x_m,\ C_{arg_1}.m\ \ldots\ C_{arg_n}.m] \end{aligned}$$

  Furthermore, we need to create an *IfThen*-constraint for each argument $arg_1\ \ldots arg_n$ of the original constraint (the constraints named $C_{arg_1} \ldots\ C_{arg_n}$ above). These need to specify the further constraints applying to potential new holes in argument-positions of f:

$$\begin{aligned} C_{arg_i}: &\quad IfThen(arg_i,\ f,\ C'_{arg_i}) \\ C'_{arg_i}: &\quad NotSimult((C_{arg_i}.1,\ C_{x_1})\ \ldots\ (C_{arg_i}.m,\ C_{x_m})) \end{aligned}$$

[8]The *NotSimult*-constraint is obviously only necessary if there is more than one other argument involved in the equality-constraint.

Finally, all of the *IfThen*-constraints created above, are dependent on each other. We thus create a top-level *NotSimult*-constraint to express the dependency amongst the arguments of the original constraint:

$$C' : \quad NotSimult((arg_1,\ C_{arg_1})\ \ldots\ (arg_n,\ C_{arg_n}))$$

An *UnEqual*-constraint C, may have several parent constraints for its different arguments. If there is a parent *NotSimult*-constraint, C_p, for the hole we instantiated, $?h$, the constraints $C_{x_1} \ldots C_{x_m}$ on the new holes $?x_1\ \ldots\ ?x_m$ replace C in C_p. If a parent constraint exists for any of the other arguments of C, $arg_1\ \ldots arg_n$, C is replaced by the new constraint C' (defined above) in the parent.

7.7.4 After Synthesis

After the synthesised terms have been filtered through the counter-example checker, the remaining conjectures are passed on to IsaPlanner for a proof attempt. IsaPlanner applies induction with rippling, lemma calculation and case-analysis. Lemma calculation is helpful in some proofs, as a conjecture may sometimes need a lemma that has not been synthesised yet. An example is the proof of $rev(rev\ l) = l$, for which IsaPlanner calculates and proves a needed lemma: $rev(l\ @\ [h]) = h\#(rev\ l)$. Without lemma calculation, the proof above would have to wait until synthesis had found the theorem $(rev\ a)\ @\ (rev\ b) = rev(b\ @\ a)$, which is a more general variant of the lemma above. This also means that the constraints gained would not be available until later, thus delaying a search space reduction for synthesis.

IsaCoSy will occasionally produce theorems that are special cases of other theorems, e.g. $(a + b) + a = (b + a) + a$ as well as the general version $(a + b) + c = (b + a) + c$. As these specialised variants rarely are of interest, IsaCoSy has a subsumption check to filter theorems for which a more general variant exists. Note that the subsumption check does not reduce the search space, it merely acts as a filter on what is displayed to the user.

7.8 Case Study: A Small Theory about Natural Numbers

To illustrate how IsaCoSy works, consider a minimal theory about natural numbers, with one recursive function '+' defined in the usual way. In total we have three function

symbols: $+$, *Suc* and $=$, as well as the constant 0. To generate initial information about constraints and argument domains, we have the injectivity and distinctness rules for *Suc*, reflexivity as well as the two rules defining +. Finally, we also assume that the heuristic for only allowing fresh variables in the left-hand side of an equation is used. We do not impose any restrictions on how many different variables are allowed, nor do we attempt to eagerly discover associativity and commutativity theorems. We will compare the number of conjectures synthesised by IsaCoSy with a naive version of synthesis, as used in [57], which generates all possible terms.

We will use the notation $x \in \{\ldots\}$, to specify the set of constants an argument x is allowed to be instantiated to. Addition will initially have the following associated information about argument domains and constraints[9]:

Name:	$x + y$
Min size:	3
Argument Domains:	$x \in \{+\}$
	$y \in \{0, Suc, +\}$
Term-Size:	-
Constraints:	-

Recall that the omission of 0 and *Suc* from the domain of the first argument comes from the defining equations being treated as rewrite rules.

For $=$, the initial information is:

Name:	$l = r$	
Min size:	3	
Argument Domains:	$l \in \{0, Suc, +\}$	
	$r \in \{0, Suc, +\}$	
Term-Size:	$l \geq r$	(Commutativity)
Constraints:	$\mathbf{C_1} : NotSimult((l,\ C_{l_1}),\ (r,\ C_{r_1}))$	(Injectivity)
	$C_{l_1} :\ NotAllowed(l,\ Suc)$	
	$C_{r_1} :\ NotAllowed(r,\ Suc)$	
	$\mathbf{C_2} :\ NotSimult((l,\ C_{l_2}),\ (r,\ C_{r_2}))$	(Distinctness)
	$C_{l_2} :\ NotAllowed(l,\ Suc)$	
	$C_{r_2} :\ NotAllowed(r,\ 0)$	
	$\mathbf{C_3} :\ UnEqual(l,\ r)$	(Reflexivity)

[9] In the implementation, arguments are identified only by the index of the arguments position. Here we will however give them names for readability

Finally the initial information for *Suc* is:

Name:	*Suc n*
Min size:	2
Argument Domains:	$n \in \{0, Suc, +\}$
Term-Size:	-
Constraints:	-

We want to synthesise equations. This means we have to start synthesising terms of size 3, with $=$ as the top level symbol and two holes, each of size 1. The initial term is thus:

$$\underbrace{?h_1}_{size\ 1} = \underbrace{?h_2}_{size\ 1}$$

The holes, represented by the meta-variables $?h_1$ and $?h_2$ will, in addition to their specified size, inherit the restrictions specified for the corresponding arguments of $=$ above.

Size 3

We can generate two terms of size 1, the constant 0 or a variable a. Putting these together, IsaCoSy synthesises only one term: $a = 0$ (out of a possible five for the naive version of synthesis). The synthesised term is not a theorem, so it is discarded after counter-example checking. Note that IsaCoSy does not synthesise $a = a$ or $0 = 0$ thanks to the equality constraint from reflexivity. Neither does it synthesise $0 = a$ or $a = b$, as both of these have variables in the right-hand side that do not occur on the left.

Size 4

For terms of size 4, IsaCoSy starts from the template $\underbrace{?h_1}_{size\ 2} = \underbrace{?h_2}_{size\ 1}$. Note that we do not consider terms where the right-hand side is larger than the left, due to the constraint arising from the commutativity of equality.

IsaCoSy only synthesises one non-theorem (which is caught by counter-example checking) for this size:

$$Suc\ a = a$$

Note that conjectures of the form $Suc(\ldots) = 0$ are not generated as this can be written to *False* by the distinctness theorem.

The naive version produces ten conjectures of size 4 (we use / to separate alternative right-hand sides):

$$a = Suc\ 0\ /\ Suc\ a\ /\ Suc\ b \quad 0 = Suc\ 0\ /\ Suc\ a$$
$$Suc\ a = 0\ /\ a\ /\ b \qquad Suc\ 0 = 0\ /\ a$$

Note that IsaCoSy does not produce conjectures such as $Suc\ a = b$ above due to the heuristic which disallows fresh variables in the right-hand side of an equation.

Size 5

For terms of size 5, we get two possibilities to start from:

$$\underbrace{?x_1}_{size\ 2} = \underbrace{?x_2}_{size\ 2} \text{ and } \underbrace{?y_1}_{size\ 3} = \underbrace{?y_2}_{size\ 1}$$

From the size restriction, the former can only attempt to generate terms of the form $Suc\ ?x = Suc\ ?y$, but this is disallowed due to the injectivity of *Suc*, so no terms will be generated for this case.

Using the second template, IsaCoSy produces 8 conjectures:

$$a + b = 0\ /\ a\ /\ b \qquad a + 0 = 0\ /\ a$$
$$a + a = 0\ /\ a \qquad Suc(Suc\ a) = a$$

The list above includes one theorem: $a + 0 = a$. The remaining conjectures are filtered out by counter-example checking. The theorem found can be proved automatically and is then given to the constraint generator, which will conclude that we no longer should generate terms where 0 is the second argument to $+$.

The naive version of synthesis generates a total of 45 conjectures of size 5.

We will revisit this case-study in §8.3.1, where we evaluate IsaCoSy and compare its performance with the naive version on several larger theories as well. We will also discuss the effect of additional heuristics on the search space.

7.9 Summary

We have developed and implemented a new way of synthesising lemmas for inductive theories in the IsaCoSy program. It works by only generating new terms, that cannot be rewritten by any existing rules. This is achieved by deducing a set of constraints from available theorems, initially function definitions, automatically deduced theorems about datatypes and library theorems such as reflexivity.

Synthesised conjectures are first given to a counter-example checker to avoid trivially false statements. Inductive proofs are attempted on the remaining conjectures, and any theorems found can be used to deduce further constraints on synthesis. The synthesis procedure starts synthesising small terms, so any theorems discovered help reducing the search space when synthesising larger, more complicated terms. This way, we can greatly reduce the number of possible terms compared to a naive version.

Chapter 8

Evaluation of Conjecture Synthesis

8.1 Introduction

This chapter will present the evaluation of IsaCoSy. The implementation was described in chapter 7.

We wish to verify the following main hypothesis about our system:

- IsaCoSy has a smaller search-space and considers fewer non-theorems than a naive version of term synthesis, thus making theorem synthesis computationally feasible on a regular computer.
- IsaCoSy produces interesting theorems, e.g. the kind of theorems found in Isabelle's libraries.
- IsaCoSy produces lemmas that are useful in further proofs. Theorem synthesis is thus a viable alternative to lemma speculation. Andrew Ireland's paper on proof critics contains a set theorems that require additional lemmas, found by the lemma speculation proof critic [41]. We would like IsaCoSy to produce background theories containing lemmas which allow such theorems to be proved without critics.

IsaCoSy has been evaluated on various inductive theories about natural numbers, lists and binary trees. We describe the methodology for the experiments in §8.2. Experiments providing evidence for the first part of the hypothesis, concerning the search space size for synthesis, are discussed in §8.3. In §8.4, we discuss whether the theorems synthesised by IsaCoSy are interesting, by comparing them to Isabelle's libraries, and calculating precision and recall. In §8.5, we address the third part of the hypothesis, and show that rippling using a background theory synthesised by IsaCoSy can

prove theorems that would otherwise require lemma speculation. We then present and evaluate some improvements of the synthesis algorithm. In §8.6 we consider constraint generation from conjectures that have passed counter-example checking but that IsaPlanner failed to prove. In §8.7, we restrict instantiations of type variables to avoid synthesising uninteresting terms about nested lists. We discuss some limitations of IsaCoSy in §8.8, and discuss related work in §8.9.

8.2 Methodology

To verify the hypothesis above, we have evaluated IsaCoSy on a number of inductive theories about natural numbers, lists and binary trees.

The first experiment is a continuation of the case-study from §7.8. We compare the effects of IsaCoSy's two optional heuristics, presented in §7.6, on the size of the search space, using a small theory about addition. Recall that the optional heuristics concern whether to attempt to prove associativity and commutativity properties prior to synthesis, as well as restrictions on the number of different variables allowed in terms. In all experiments where this heuristic was used, the maximum number of variables in a synthesised term was set to 1 + maximum arity of any function. This was motivated by a survey of number of variables in theorems in Isabelle's libraries (see §7.6.2).

The majority of the analysis in this chapter is based on experiments on six slightly larger theories, with IsaCoSy's optional heuristics switched on. Here, we recorded the run-time for the different tasks that make up the synthesis algorithm, as well as search space size and which theorems and conjectures were synthesised. These theories are listed below:

- Natural Numbers:
  - addition and multiplication
- Lists:
  - append, reverse, length
  - append, reverse, map
  - append, reverse, quick-reverse (qrev)
  - append, foldl, foldr
- Binary Trees:

– mirror, height, nodes, max.

The definitions of the functions above can be found in Appendix A. We found that most theorems in Isabelle's library contain no more than around three function symbols, which is why we limit our evaluation-theories to three or four function symbols each. We suggest adding heuristics for managing larger theories as further work (see §9.3).

Some preliminary experiments were undertaken to check the maximum term size our synthesis algorithm could manage before the ML-process ran out of memory. We found this to be 14 (for the theories involving append and reverse) and thus ran all experiments up to that size. Furthermore, Isabelle's library does not contain any equational theorems about the functions in our theories that are larger than size 14 and of the kind IsaCoSy can produce. The majority are of smaller sizes.

The experiments were run on a computer with a 2 GHz Intel Xenon processor. Full results from the experiments are available on-line[1], including all theorems, conjectures, proofs and run-time statistics for each theory. The synthesised theorems from the experiments are listed in Appendix C.

8.3 Synthesis Search Space

We expect IsaCoSy to cut down the synthesis search space considerably compared to a naive version of synthesis that simply generates all possible terms, as used in [57]. We will first present a brief evaluation of IsaCoSy's optional heuristics in §8.3.1, illustrating their effect on the search space size. Having established the benefits of these heuristics, in §8.3.2, we present a comparison of the search space size for IsaCoSy and the naive version of synthesis on the larger theories listed in §8.2. As we see in §8.3.3, the overall run-time of IsaCoSy is proportional to how many terms are generated and have to be counter-example checked. Hence, we do not provide any run-times for the naive version but only compare how many terms are generated by each algorithm, bearing in mind that this is proportional to the overall run-time in most cases.

8.3.1 Effect of Heuristics

We now continue where we left the case-study about addition on natural numbers from §7.8. Figure 8.1 summarises the number of terms synthesised for increasingly large term sizes. As well as the naive variant and IsaCoSy's basic version of synthesis which

[1] `http://dream.inf.ed.ac.uk/projects/lemmadiscovery/synth_results.php`

was described in §7.8, results are also included for two versions of synthesis using the optional heuristics described in §7.6. The first version behaves as the basic version of the synthesis algorithm, but restricted to only allow three different variables in the synthesised terms (the arity of $+$ is 2, so we allow $1 + 2 = 3$ variables). The second version does, in addition, also eagerly attempt to synthesise associativity and commutativity properties for appropriate functions, prior to commencing the synthesis. As we shall see, this version of synthesis performs the best and is thus used in all further experiments in this chapter.

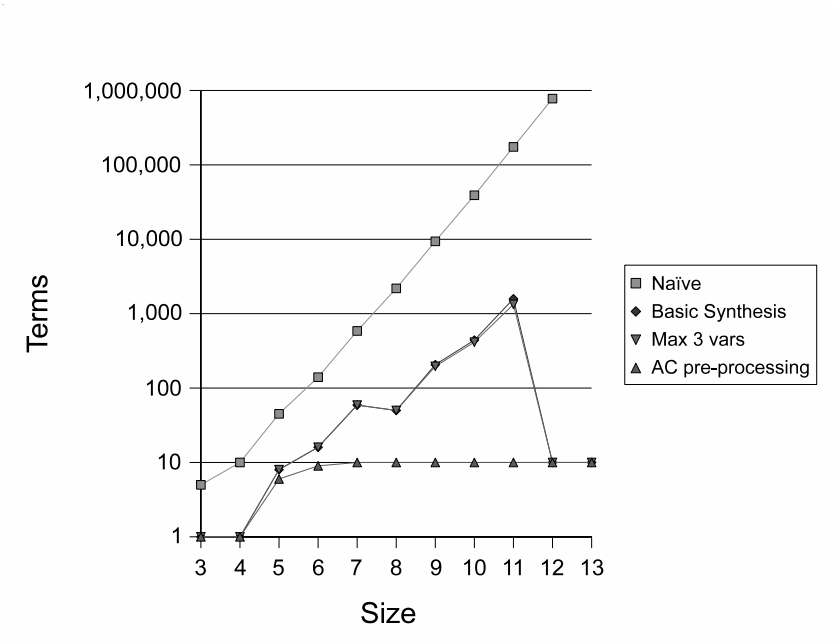

Figure 8.1: The number of equational terms generated up to size 13 about natural numbers with addition for a naive version of synthesis, IsaCoSy's basic version of synthesis and synthesis with additional heuristics. The graph shows the search space reduction achieved by adding a pre-processing step to look for associativity and commutativity properties, as well as restricting the number of different variables allowed in synthesised terms. Note that the y-axis scale is logarithmic for better visibility.

The number of terms increases rapidly for the naive version, until the ML process runs out of memory when reaching size 13. IsaCoSy's basic synthesis version performs considerably better. When reaching size 7, the commutativity of addition is discovered, which allows the arguments to be ordered, cutting out many symmetries. In fact, fewer terms of size 8 than 7 are synthesised by IsaCoSy. The largest number of conjectures is synthesised for size 11. At this point, the associativity of addition is discovered. Now, all theorems in our small theory, adhering to the constraints of the synthesis algorithm,

have been discovered, and all corresponding constraints are available. The domains for the arguments of addition are now empty, meaning no function symbols can occur in the arguments of plus. If asked to continue, only 10 terms are considered of size 12 and 13. These are 'silly' non-theorems of the form $Suc(\ldots Suc(Suc(a + b))) = a + b$, stacking up lots of successor functions.

When restricting IsaCoSy to only 3 different variables (each of which may occur several times), even fewer terms are generated for larger sizes. The same theorems are still discovered. The greatest difference comes from the AC pre-processing heuristic, which in this toy theory, manages to discover all relevant theorems as consequences of associativity or commutativity (apart from associativity and commutativity themselves, this includes the commuted variants of the definition of plus). Hence, it is only possible to synthesise a few non-theorems possible for each size.

Size	Theorem
5	$a + 0 = a$
7	$a + Suc\ b = Suc(a + b)$
7	$a + b = b + a$
11	$(a + b) + c = (b + a) + c$
11	$(a + b) + c = (a + c) + b$
11	$(a + b) + c = (c + b) + a$

Table 8.1: Theorems about addition discovered by IsaCoSy without optional heuristics.

Table 8.1 shows the theorems discovered and proved by IsaCoSy using the basic setting. These are, as expected, the commuted variants of the definition of plus, as well as commutativity and associativity. Associativity appears in a few different variants, but not in the common form: $(x + y) + z = x + (y + z)$. This is because of the size constraints imposed after the discovery of commutativity. The common form is arguably more useful to rippling, which is why the AC pre-processing technique generates this variant.

8.3.2 Search Space Reduction over Naive Synthesis

We will now move on to comparisons on slightly larger theories. IsaCoSy is here configured to use all the optional heuristics. The number of different variables allowed in terms is thus restricted to 1 + maximum arity of any function in the theory,

and IsaCoSy includes pre-processing steps looking for associativity and commutativity theorems prior to the start of synthesis.

Figure 8.2 compares the total number of terms generated by IsaCoSy and by a naive version, up to size 11 (the naive version runs out of memory for larger sizes) for the six evaluation theories on natural numbers, lists and trees.

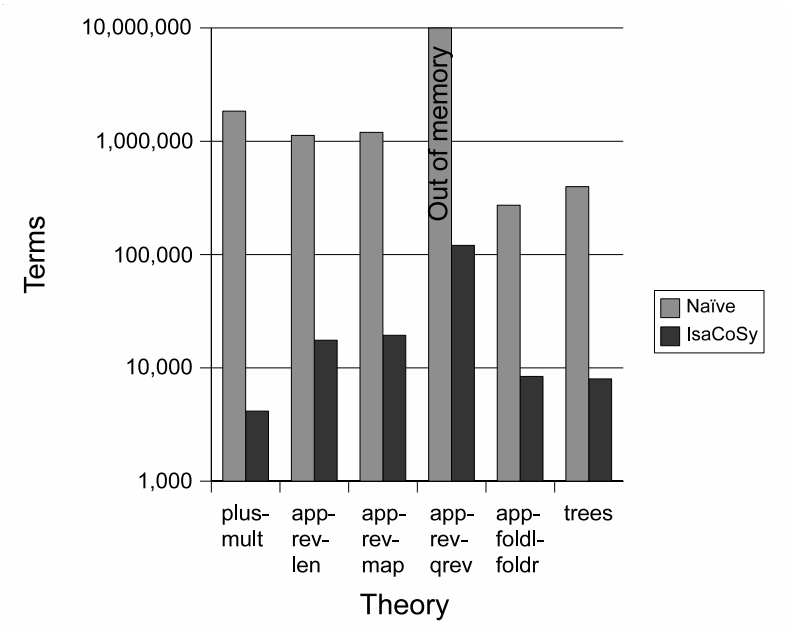

Figure 8.2: Number of terms generated up to size 11 for IsaCoSy and for a naive version on six different theories on natural numbers, lists and trees. The search-space for IsaCoSy is shown to be considerably smaller. Note that the y-axis scale is logarithmic for better visibility.

In the natural number theory, the naive variant generates over 1.8 million terms of sizes up to 11, compared to just over 4000 for IsaCoSy. For both the list theories with append, reverse and length/map the naive version produces over 1.1 million terms of size 11, compared to less than 20 000 for IsaCoSy. For the theory about quick-reverse (*qrev*) the naive version runs out of memory. IsaCoSy produces less than 10 000 terms for both the theory about foldl/foldr and about trees, while the naive version produces between 250 000 - 400 000.

The reason for the big difference between IsaCoSy and the naive approach on the natural number theory is because both addition and multiplication are associative and commutative. These properties produce useful constraints, which greatly helps cutting down the search space. The more structure a theory has, the more efficient synthesis will be. IsaCoSy produces the largest number of terms (compared to how big the

possible term-space is) for the quick-reverse theory. This is because many theorems in this theory require generalisation of an accumulator variable, which is beyond the capabilities of IsaPlanner. As the theorems cannot be proved they are not used to generate additional constraints, so the search space remains large. We discuss this in more detail in §8.6.

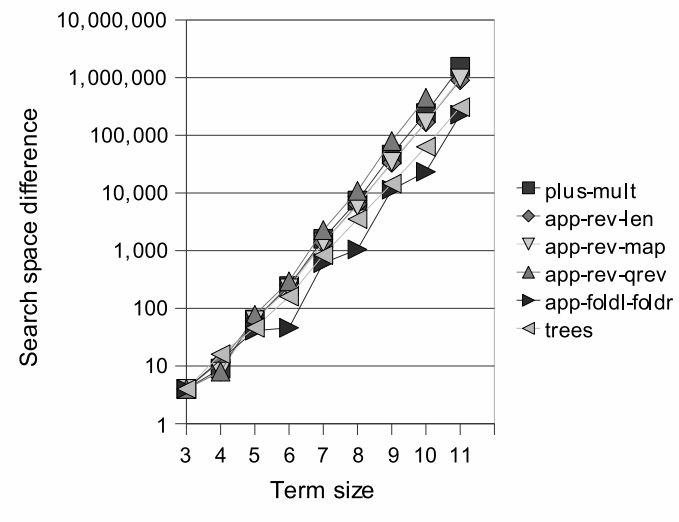

Figure 8.3: The differences between the naive version and IsaCoSy in search space size for synthesis of terms sized 3 - 11, calculated on six theories about natural numbers, lists and binary trees. The differences in search space size form exponential curves. This shows that IsaCoSy manages an exponential cut of search space size compared to naively generating all terms. Note that the y-axis scale is logarithmic.

Figure 8.3 shows the difference in search space size between the naive version and IsaCoSy, computed by subtracting the number of terms generated for each term-size (from 3 up to 11) on the same six theories as above. For each theory, IsaCoSy's heuristics manage an exponential cut of the search space of all naively generated terms.

8.3.3 Run-time and Space Usage

We found that the run-time and space usage of IsaCoSy increases exponentially with the size of the terms synthesised. Furthermore, the time taken per iteration is proportional to the number of synthesised terms of that size. Figures 8.4 and 8.5 illustrate two examples, on a theory about natural numbers and lists respectively. The number of terms generated has been plotted together with the run-time for each size. The graphs increase exponentially, and clearly mirror each other.

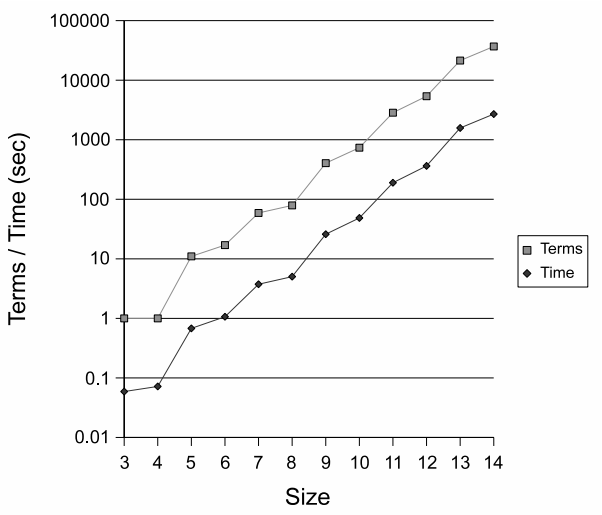

Figure 8.4: The two graphs show time and number of synthesised terms for each iteration from size 3 - 14 of synthesis, on the theory about natural numbers with addition and multiplication. As the graphs mirror each other, we can conclude that the run-time is proportional to the number of terms synthesised for each size. The y-axis uses a logarithmic scale, which means that the growth in search space size is exponential as the size of terms synthesised is increased.

The reason for the correlation between the number of terms and the run-time is that IsaCoSy spends most of its time doing counter-example checking on the terms it generates. Proof attempts make up a considerably smaller proportion of the total time. This is because the vast majority of terms generated are non-theorems and filtered out by counter-example checking. The total run-times, as well as timings for the different tasks making up the synthesis process are summarised in table 8.2 on page 112. For all theories in the experiments, IsaCoSy spends the majority of time performing counter-example checking. Comparatively little time is spent on proof attempts.

8.4 Precision/Recall Analysis

To asses the quality of the theorems produced by IsaCoSy we perform a precision/recall analysis using Isabelle's library as reference. However, Isabelle does not have a standard library for binary trees, so this theory could not be analysed here. The quick-reverse function is also not included in the library, and thus had to be excluded.

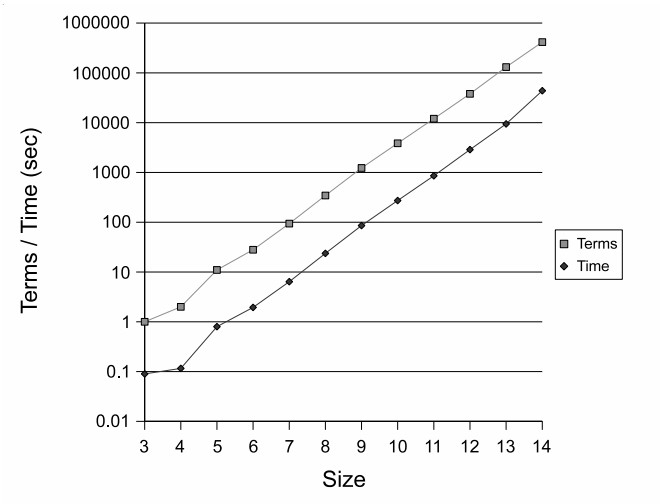

Figure 8.5: The two graphs show time and number of synthesised terms for each iteration from size 3 - 14 of synthesis, on the theory about lists with append, reverse and length. The graphs mirror each other, showing that the run-time is proportional to the number of terms synthesised for each size. The y-axis uses a logarithmic scale. The growth of search space size/run-time is thus exponential.

8.4.1 Natural Numbers

The theorems synthesised for natural numbers, about addition and multiplication are shown in table C.2 in Appendix C. The standard commutativity and associativity theorems are synthesised, along with commuted versions on the function definitions. IsaCoSy also synthesises theorems for the distributivity of multiplication over addition.

Isabelle's library contains 12 equational theorems about addition and multiplication[2], 10 of which are synthesised by IsaCoSy:

$$\begin{array}{cc} a + 0 = a & a + Suc\, b = Suc(a + b) \\ a * 0 = 0 & a * Suc\, b = a + (a * b) \\ a + b = b + a & a * b = b * a \\ (a + b) + c = a + (b + c) & (a * b) * c = a * (b * c) \\ (a * b) + (c * b) = (a + c) * b & (a * b) + (a * c) = (b + c) * a \end{array}$$

Using Isabelle's 12 theorems as a benchmark for 'interestingness' we can calculate precision and recall for IsaCoSy. With ten of our theorems included in the library, this gives recall of 83%. IsaCoSy synthesised a total of 16 theorems for this theory, which gives precision of 63%.

[2]Note that we exclude any theorems of forms that IsaCoSy cannot produce, such as theorems with assumptions. Theorems containing more than one equality were also excluded as IsaCoSy was not allowed to use equality other than at the top-level in these experiments.

Theory	Total time	Counter-examples	Proof	Synthesis
plus-mult	1h 22 min	1h 20 min	1 min	48 sec
app-rev-len	15h 56 min	15h 30 min	17 min	9 min
app-rev-map	17h 21 min	16h 51 min	17 min	11 min
app-rev-qrev	18h 43 min	17h 20 min	1h 11 min	12 min
app-foldl-foldr	6h 8 min	6h 5 min	3 sec	3 min
trees	9h 46 min	9h 41 min	3 min	2 min

Table 8.2: Total run-time for IsaCoSy on six theories up to size 14, along with a break-down on how much time was spent on each sub-task during the synthesis process. Times are rounded up to nearest hours and minutes. Note that for all theories, the largest proportion of time is spent on counter-example checking.

The two theorems from Isabelle's library that are not synthesised are below:

Label	Theorem
add_suc_shift	$(Suc\ m)\ +\ n = m\ +\ (Suc\ n)$
nat_left_commute	*x + (y + z) = y + (x + z)*

These theorems are trivially derivable by simplification from theorems we do synthesise. The theorem *add_suc_shift* is not synthesised as its left-hand side is identical to the left-hand side of the definition of addition, and thus not allowed in synthesis. Should we wish to derive theorems of this form, one solution would be to add them to the set of theorems IsaCoSy tries to prove when discovering that a function is commutative. Currently, it derives the commuted versions of the function's definition as theorems (for example the theorem $b\ +\ Suc\ a = Suc(b+a)$). This could easily be extended to also include trying to prove theorems of the same form as *add_suc_shift*, without having to relax constraints on synthesis. The theorem *nat_left_commute* is not synthesised because of the size-constraint requiring the first argument of plus to be larger than the second, introduced on discovering that plus is commutative.

8.4.2 Lists

Isabelle's list library does not contain the qrev-function, so this theory has been excluded from the analysis. Recall that the *foldl/foldr* functions compute a single value from a list by recursively applying a binary function to an accumulator and the head of the list, producing a new accumulator value. The definitions can be found in Appendix A.

The theorems synthesised for the list theories are shown in tables C.3, C.4 and C.5 in Appendix C. IsaCoSy produces a total of 24 theorems in these theories, while Isabelle's list theory contains 9 relevant theorems. All of these are synthesised, giving recall of 100%. The 'interesting' theorems are listed below:

$$a \mathbin{@} [\,] = a \qquad (a \mathbin{@} b) \mathbin{@} c = a \mathbin{@} (b \mathbin{@} c)$$

$$rev(rev\ a) = a \qquad (rev\ a) \mathbin{@} (rev\ b) = rev\ (b \mathbin{@} a)$$

$$rev(map\ a\ b) = map\ a(rev\ b) \qquad (map\ a\ b) \mathbin{@} (map\ a\ c) = map\ a\ (b \mathbin{@} c)$$

$$foldl\ a\ (foldl\ a\ b\ c)\ d = foldl\ a\ b\ (c \mathbin{@} d) \qquad foldr\ a\ b\ (foldr\ a\ c\ d) = foldr\ a\ (b \mathbin{@} c)\ d$$

$$len(rev\ a) = len\ a$$

Because of the 14 extra synthesised theorems, not in Isabelle's library, the precision is just 38%. Most of the perhaps uninteresting extra theorems IsaCoSy produces are about reverse and append. We discuss why so many such theorems are synthesised in §8.8.2.

8.5 Using Synthesised Theories Instead of Lemma Speculation

We want to further assess the quality of the theorems and lemmas produced by IsaCoSy by testing whether using a synthesised theory makes it possible to prove difficult theorems automatically without having to use the lemma speculation critic (see chapter 6). In Ireland and Bundy's paper on proof critics [41], there are seven theorems not provable by rippling from function definitions with just the simpler and more efficient lemma calculation critic. These require additional lemmas that can only be found by the more complicated lemma calculation critic. We applied rippling with the additional theorems found by IsaCoSy as extra wave-rules. Rippling could in addition use the lemma calculation critic, if needed. The results are shown in table 8.3.

Rippling now manages to solve six out of the seven theorems without having to use lemma speculation. The first two theorems are proved by using theorem N2, $a + Suc\ b = Suc(a + b)$, from table C.2 in Appendix C. The list theorems are proved using three of the synthesised theorems; the associativity of append (L1 in table C.3), the distributivity of append over reverse, $(rev\ a) \mathbin{@} (rev\ b) = rev\ (b \mathbin{@} a)$, (L7 in table C.3) and theorem L9: $rev(a \mathbin{@} [b]) = b \# (rev\ a)$. The only proof requiring even lemma calculation in the step case was that of theorem 7, where lemma calculation produced

Label	Theorem	Proved	Lemma(s)
1	$x + Suc\ x = Suc(x + x)$	Yes	N2
2	$even(x + x)$	Yes	N2
3	$rev(rev(x\ @\ y)) = rev(rev\ x)\ @\ rev(rev\ y)$	Yes	L9 or L7 + L1
4	$rev((rev\ x)\ @\ y) = (rev\ y)\ @\ x$	Yes	L7
5	$rev(rev\ x)\ @\ y = rev(rev(\ x\ @\ y)$	Yes	L9
6	$even(len(x\ @\ x))$	No	-
7	$rotate\ (len\ x)\ (x\ @\ y) = y\ @\ x$	Yes	L1

Table 8.3: Theorems requiring lemma speculation or lemmas found by synthesis prior to the proof attempt. The names used for lemmas refer to the labels used for theorems in Appendix C.

the lemma $l\ @\ [a]\ @\ m = l\ @\ (a\ \#\ m)$. Note that theorem 4 was conjectured by synthesis, but the proof at that time failed as L7 was not yet available.

The only theorem that failed to be proved was theorem 6. The step-case of theorem 6 becomes blocked after rewriting using the definitions of *len* and *even*:

$$even(\ Suc(len(x\ @\ h\#x^{\uparrow}))^{\uparrow}\ )$$

At this point rippling fails as the required lemma, $len(x\ @\ (y\ \#\ z)) = Suc(len(x\ @\ z))$, is not synthesised. We discuss the failure of synthesising a more general version of this lemma, $len(x\ @\ y) = (len\ x)\ +\ (len\ y)$, in §8.8.2. However, even if we had succeeded in synthesising the more general lemma, it would in this case not be allowed as a wave-rule in rippling, because the inductive hypothesis does not embed into the resulting goal:

$$even(Suc((len\ x)\ +\ (len(h\ \#\ x))))$$

Neither does rippling allow applying the synthesised theorem L5: $len(a\ @\ b) = len(b\ @\ a)$. The goal resulting from applying L5 as a wave-rule does have an embedding, but does not decrease the measure:

$$even(\ Suc(len(\ h\#x^{\uparrow}\ @\ x))^{\uparrow}\ )$$

This suggests that for proof-tools such as rippling, mathematically 'uninteresting' theorems may sometimes be useful in practice. However, it is worth noting that if best-first rippling was used [46], theorem 6 would be provable, without lemma speculation. In

best-first rippling, steps are allowed to be non-measure decreasing if nothing better is available, so L5 could be applied to unblock the step-case.

8.6 Allowing Conjectures to Generate Constraints

Quick-Reverse (*qrev*) is the tail recursive version of reverse. Many theorems about tail recursive functions such as *qrev* will require a generalisation of the accumulator variable in order for the inductive hypothesis to apply. IsaPlanner does not have the capabilities to discover such generalisations, which is why it fails to prove many synthesised theorems in this theory. One example is the theorem $qrev\ a\ [\,] = rev\ a$, which needs the more general theorem $(rev\ a)\ @\ b = qrev\ a\ b$ to complete the proof. In total IsaPlanner proves 19 theorems about *qrev* (see table C.6 in Appendix C), while a further 46 conjectures pass counter-example checking, but fail to be proved. As a consequence of failing to prove many theorems, few constraints were generated and the search space remained relatively large compared to the other theories, as shown in figure 8.2 on page 108. Some of the synthesised theorems IsaPlanner manages to prove are also rather contrived.

A solution to make IsaCoSy generate fewer terms in situations where IsaPlanner fails to prove many conjectures, is to allow conjectures that have passed counter-example checking, but not been proved, to also generate constraints. This should benefit synthesis in the theories such as the one about quick-reverse. We observed that many of the conjectures in this theory pass counter-example checking but are not proved by IsaPlanner appear to be theorems. A random selection of 20 out of the 46 unproved conjectures were proved by hand. Furthermore, we have not observed any non-theorems that have passed counter-example checking in any of the experiments, which supports our confidence in Isabelle's counter-example checker.

We repeated the experiment on the theory about quick-reverse, this time allowing unproved conjectures to generate constraints. The run-time for generating terms up to size 14 was reduced by 11 hours, now only taking 7 hours and 40 minutes. 11 fewer theorems were generated, but it was the larger, more contrived ones that were cut out. Theorems L23-24 and L26-31 in table C.6 were still generated. As an example, theorem L25: $qrev\ (qrev\ a\ b)[\,] = qrev\ b\ a$, is no longer generated as the simpler (but unproved) theorem $qrev\ a\ [\,] = rev\ a$ is now allowed to generate constraints. There were also fewer unproven conjectures, only 8 as opposed to 46, as larger variants of previous unproven conjectures were no longer allowed to be generated.

Allowing unproved conjectures to generate constraints makes IsaCoSy less dependent on the underlying theorem prover and lets it operate efficiently even when many conjectures remain unproved. There is of course a risk of missing desired theorems if any non-theorems slip through counter example checking, although we have not yet encountered any such cases in the experimental theories. The benefits in efficiency seem to be worth the risk.

8.7 Restricting Polymorphic Types

Large gains in run-time can also be obtained by disallowing instantiations of polymorphic type variables to another polymorphic type. This would disallow nested lists (lists of lists) and nested trees. We noticed that many non-theorems in the list domain contained highly nested lists. As an example of terms that it would be beneficial to prune from the term space, consider a term $?h_0\#l$, which is of type $?\alpha\ list$, with $?h_0 :: ?\alpha$. Suppose $?h_0$ gets instantiated to another cons-operator, which also instantiates the type variable $?\alpha$ to $?\beta\ list$. The original term is now $(h_1\ \#\ h_2)\#l :: ?\beta\ list\ list$. Subsequent instantiations may cause further nesting of lists. The restriction we suggest would disallow a type-variable, such as $?\alpha$ above, to be instantiated to a new type containing another variable.

We repeated the synthesis experiments on some of the list theories, with a specialised lists datatypes over natural numbers, rather than polymorphic lists. The same theorems as before were still found, but the number of non-theorems decreased significantly as the search space was reduced. As run-times are proportional to the search-space size, IsaCoSy ran considerably faster. The figures for precision and recall from §8.4 was however not affected, as the same theorems are still discovered. The results are summarised below:

Theory	Run-time		Non-theorems	
	polymorphic	non-polymorphic	polymorphic	non-polymorphic
app-rev-len	15h 56 min	5h 8 min	601 405	229 104
app-rev-map	17h 21 min	4h 31 min	636 361	195 800
app-foldl-foldr	6h 8 min	17 min	249 404	14 503

In the future, we could achieve these cuts by implementing restrictions on type-variable instantiations, as we often do want to use polymorphic datatypes.

8.8 Limitations

8.8.1 Dealing with Commutativity

Commutative functions may cause synthesis to produce a large number of different variants of the same theorem. For example, the natural number theory from our experiments produced eight variants of distributivity of multiplication over addition with variables commuted in different orders:

$$\begin{aligned}
\text{N9:} \quad & (a * b) + (c * b) = (a + c) * b \\
\text{N10:} \quad & (a * b) + (c * a) = (b + c) * a \\
\text{N11:} \quad & (a * b) + (c * a) = (c + b) * a \\
\text{N12:} \quad & (a * b) + (c * b) = (c + a) * b \\
\text{N13:} \quad & (a * b) + (a * c) = (b + c) * a \\
\text{N14:} \quad & (a * b) + (a * c) = (c + b) * a \\
\text{N15:} \quad & (a * b) + (b * c) = (a + c) * b \\
\text{N16:} \quad & (a * b) + (b * c) = (c + a) * b
\end{aligned}$$

Using Isabelle's library for reference, as we did for the precision and recall analysis, theorems N9 and N13 are 'interesting'. However, it is difficult to build a filter that would decide which variant(s) should be kept, and IsaCoSy does not currently attempt such filtering.

8.8.2 Term Ordering

IsaCoSy currently uses a very simple ordering on terms, based on their size. Synthesised equations are required to have a left-hand side of larger or equal size than the right-hand side. As this measure is not a total ordering, it is not sophisticated enough to avoid the problems illustrated by the examples below.

8.8.2.1 Symmetries

An inefficiency arising when synthesising theorems with equal sized left- and right-hand sides, is that IsaCoSy tends to produce two symmetric equations. This happens in the theory about append, reverse and map, where the theorem $map\ a\ (rev\ b) = rev(map\ a\ b)$ is synthesised both ways around. This is not a problem when the theorems are supplied to rippling, but if made available to a rewriting system, such as Isabelle's simplifier, this would cause non-termination. Implementing a total ordering on terms would solve the problem.

8.8.2.2 Constraints from Invalid Rewrite-Rules

We tried to extend the theory about append, reverse and length to also include addition. IsaCoSy did however not produce any new theorems other than those that had been synthesised prior to including addition. We would have expected to get theorems such as $len(a\ @\ b) = (len\ a)\ +\ (len\ b)$. This is not synthesised as we unintentionally prematurely removed @ from the domain of len. This constraint has arisen from the theorem stating that append is commutative under len: $len(a\ @\ b) = len(b\ @\ a)$, which is of smaller size and thus generated first. We note that this theorem is not actually a valid rewrite rule, it is not measure decreasing in any meaningful way and both sides are of the same size. However, it is still an interesting theorem, and we still want to synthesise it. A solution to the problem would be to only allow constraints to be generated if the theorem is a valid rewrite rule. This again requires a total ordering on terms to detect which theorems are valid rewrite rules.

In the future, a more suitable constraint would be to observe that @ is commutative under len, and impose size restrictions in these situations. IsaCoSy can currently identify commutativity theorems about a left-hand side top-level function, e.g. plus in $a\ +\ b = b\ +\ a$, from which it generates size restrictions for the functions arguments. As further work, we suggest extending this machinery to also detect commutativity under additional term-context.

8.8.2.3 Low precision for Lists

IsaCoSy only achieved 38% precision in the list domain, as many additional theorems involving append and reverse were synthesised. The reason these are synthesised is that we do not disallow sub-terms of the form $rev(?h_1\ @\ ?h_2)$. The single argument to rev, is thus always allowed to be instantiated to append. We would expect the theorem for distributing rev over append, $(rev\ a)@(rev\ b) = rev(b\ @\ a)$, to generate such a constraint, but due to the current term ordering, based on size, it is oriented in the opposite direction. Recall that constraints are only generated from the theorem's left-hand sides, as the algorithm never produces terms that can be rewritten by a previously synthesised theorem. Another term-ordering might perform better, but this is left as further work. However, note that if the distributivity of rev over append was oriented in the opposite direction, IsaCoSy would not synthesise the theorem $rev(a\ @\ [b]) = b\ \#\ (rev\ a)$, which was shown to be a useful wave-rule for rippling in §8.5. There is a perhaps a trade-off between the desire to synthesise a small 'neat' theory, and

synthesising useful wave-rules.

8.8.3 Limitations of Isabelle's Counter-Example Checker

We also experimented with an extension to the natural number theory, adding exponentiation. This caused problems with Isabelle's counter-example checker, which runs out of memory on exponential terms such as $a^{b^{(c * a)}} = a$. It was therefore only possible to run our program up to size 8. Thus, only two theorems were found[3]:

$$suc\ 0^a = suc\ 0$$

$$0\ ^a\ *\ a = 0$$

IsaCoSy also found two theorems that passed counter-example checking, but IsaPlanner was unable to prove:

$$0^{a^a} = 0$$

$$0^{(suc\ a)^b} = 0$$

8.9 Related Work in Theory Formation

There are two main approaches to theory formation, *generative* and *deductive*. Theory formation following the generative approach produces conjectures according to some set of heuristics, and then checks which of these are theorems by counter-examples and/or proof. Our work falls into this category. Other systems, such as HR [18], also follow this approach, but have not been applied to inductive theories.

Systems using a deductive approach attempt to produce new theorems as logical consequences of known facts. This approach has the advantage of not having to use counter-example checking to filter out non-theorems, but still has to apply filtering to avoid trivial or uninteresting logical consequences. The MATHsAiD [58, 59] and AGInT [68] systems used deductive methods for theorem generation. To our knowledge, MATHsAiD is the only other system that has been applied to theorem synthesis in inductive theories.

We also note that some systems, such as HR and the suggested scheme-based theory exploration model [6], include the capabilities for forming new concepts. IsaCoSy is not concerned with this side of theory formation. It is designed to produce conjectures and theorems about existing functions and datatypes, not invent new ones. A

[3]Note that in our definition of exponentiation (see Appendix A) $0^0 = 1$.

follow-on project is however currently underway, combining IsaCoSy with a program for inventing recursive functions and datatypes [16].

IsaCoSy's pre-processing step, looking for associativity and commutativity properties is similar to how the scheme-based method discovers theorems of a certain form. The scheme-based theory exploration model is being further studied and applied to theory formation in an Isabelle setting in the ongoing PhD project of Omar Montano Rivas [64]. Also related to the scheme-based approach, but not concerned with concept formation, is the lemma discovery technique for the inductive prover built for reasoning about dependent types in Coq [77]. It uses a similar technique to IsaCoSy's pre-processing step. The Coq-prover looks for a larger set of properties than IsaCoSy, also including for example distributivity.

Comparison to MATHsAiD

We will here discuss and compare the way MATHsAiD and IsaCoSy produce inductive theorems.

Unlike IsaCoSy, which generates whole terms at once and then discards most after counter-example checking, MATHsAiD first produces a set of potential left-hand sides, called *terms of interest*. Smaller terms of interest can be used to build larger ones. The generation of interesting terms is guided by heuristics, which include rules for producing terms about associativity, commutativity and distributivity for relevant functions. Our system implements a similar idea in the pre-processing step which searches for AC-properties. However, we do not currently have built in heuristics to look for distributivity, which might further decrease our search space.

After generating the terms of interest, MATHsAiD proceeds to generate theorems by replacing a variable in the term with 'TWO' (corresponding to $Suc(Suc\ 0)$ for natural numbers or $[a,\ b]$ for lists) and reasoning forward to find an appropriate right hand side of the equation (see §2.7.3). This forward reasoning may have a large search space, which is why MATHsAiD imposes a size limit on potential right-hand sides, computed as *(number of function-symbols in LHS) + number of function symbols in 'TWO' + 2*. Our system also imposes restrictions on size, requiring the left-hand side to be larger than the right-hand side. However, IsaCoSy is perhaps inefficient in that it will try to generate terms with big differences in LHS and RHS size, that are unlikely to produce any theorems.

As IsaCoSy works on Isabelle-theories, properties such as well-definedness of

functions are proved automatically by Isabelle's function package. This makes it easy to apply it to different theories. We have experimented with natural numbers, lists and binary trees. MATHsAiD can not be extended to new domains so easily, as much of the configuration has to be done by adding axioms by hand.

MATHsAiD has been applied to the domain of natural numbers with addition and multiplication [59]. It generates the common associativity, commutativity and distributivity theorems our system finds (although fewer variants of distributivity). It also produces the following three extra theorems:

$$a + (Suc\ 0) = Suc\ a, \quad a * (Suc\ 0) = a, \quad (Suc\ 0) * a = a$$

IsaCoSy does not generate these theorems as they are subsumed by more general ones. MATHsAiD was designed to aid human mathematicians, and thus has a slightly different heuristic for what an 'interesting' theorem is, which includes the above identities about 1. We note that specialised theorems are sometimes useful for an automated prover, for example, in the domain of lists, similar theorems about singleton lists are sometimes useful for rippling. IsaCoSy does not synthesise the following special case of associativity of append: $(a\ @\ [b])\ @\ c = a\ @\ (b\#c)$. This lemma was needed in the proof of the rotate-length theorem in §8.5 (it could however be found by lemma calculation), so IsaCoSy's subsumption criteria may sometimes be too strong.

MATHsAiD is considerably faster than IsaCoSy, the theorems for the natural number theory were generated in just 84 seconds. This was expected as MATHsAiD has more heuristics encoded, including a heuristic particularly looking for distributivity theorems which our system lacks. MATHsAiD has, however, not been applied to any higher-order theories (such as lists with *map* and *fold*). Our system can deal with these without modifications. IsaPlanner is capable of automatically proving harder theorems (including higher-order ones) than MATHsAiD. IsaPlanner can use its lemma calculation critic and prove theorems that MATHsAiD has to return to later, when the appropriate lemma has been generated.

8.10 Summary

IsaCoSy has been evaluated on several inductive theories about natural numbers, lists and binary trees. We aimed to verify the hypothesis that IsaCoSy is more efficient than a naive version of synthesis, which explores the whole search space, and that it

produces good quality theorems, of the kind that are found in Isabelle's libraries, and that are useful as lemmas in later proof-attempts.

The first part of the hypothesis was verified by comparing IsaCoSy to a naive version of synthesis on several different inductive theories, showing an exponential reduction in search space size. IsaCoSy is thus not only faster, but also able to explore larger term-sizes before running out of memory. IsaCoSy's run-time does however still grow exponentially with the size of the terms synthesised. We also observed that the run-time in most cases is proportional to the number of terms generated, as the program spent most of its time performing counter-example checking. IsaCoSy works most efficiently on highly structured theories, such as that of addition and multiplication, where many constraints are available from properties such as associativity and commutativity.

To evaluate the quality of theorems found by IsaCoSy, we compared them with those in the Isabelle's libraries (when available). IsaCoSy produces many good theorems, resulting in high recall of 83% for natural numbers and 100% for lists. It does however produce a number of less interesting theorems too, so precision is lower, 63% for natural numbers and 38% for lists. However, some of these extra theorems are useful to IsaPlanner's rippling machinery as wave-rules or generalisations. Using a synthesised background theory, we also showed that IsaPlanner is able to prove harder theorems, without having to rely on lemma discovery by complex techniques such as lemma speculation.

In most cases, synthesis currently takes several hours. However, we observed that most 'interesting' (according to Isabelle's libraries) theorems are often quite small, and could be found considerably quicker. Restrictions on instantiations of polymorphic types can further decrease the run-time by cutting out many non-theorems from the search-space. In the theory about quick-reverse, where IsaPlanner has difficulty proving many theorems, we showed that allowing un-proved conjectures to generate constraints improves performance and does not appear to cause IsaCoSy to miss interesting theorems.

IsaCoSy's simple term-ordering based on size, will occasionally allow for symmetric versions of equations to be synthesised when both sides are of the same size. Furthermore, there is currently no check on whether synthesised theorems are valid rewrite rules, before being used to generate constraints. If constraints are generated from an invalid rewrite rule, other expected theorems may not be synthesised.

Chapter 9

Further Work

9.1 Introduction

We will here address some limitations of our work, and propose directions for further work improving IsaPlanner and IsaCoSy. In §9.2, we summarise some suggested improvements, identified in §5.6, to IsaPlanner's capabilities of reasoning about conditional theorems and discovering conditional lemmas. We discuss a range of potential improvements to the efficiency of IsaCoSy in §9.3. In §9.4, we suggest how critics and synthesis can be combined. Different rewriting techniques require different sets of rewrite rules. In §9.5 we propose exploring configurations for IsaCoSy to synthesise sets of rewrite rules suitable for particular techniques, such as rippling or simplification. Finally, in §9.6, we discuss some potential applications for IsaCoSy.

9.2 Proofs with Conditions

Although the case-analysis technique allowed a range of proofs involving conditional statements to be proved automatically, IsaPlanner still has limitations that prevent it proving many harder theorems. This was discussed in §5.6, and will be summarised here.

IsaPlanner lacks the capability to produce conditional lemmas, in the form of implications. This is needed in, for example, proofs about sorting. We suggested a modification to lemma calculation, allowing assumptions from the blocked goal to be lazily carried through lemma calculation if they are needed in the proof of the lemma. Assumptions for a lemma should only be kept if they are needed in its proof, otherwise we risk producing lemmas that are less general and thus less likely to be applicable in

further proofs.

A further issue is fertilisation of conditional theorems, where the inductive hypothesis itself will have an assumption. In these cases, a sub-goal will remain after strong fertilisation, which IsaPlanner currently does not anticipate. The fertilisation technique should be extended to deal with possible new sub-goals arising, solving them by forward reasoning and rippling.

Some proofs fail due to IsaPlanner's default structural induction scheme not being sufficient. These proofs require simultaneous induction on several variables, or otherwise, several case-splits. As discussed in chapter 5, it is often the case that one sub-goal after a case-split is no longer rippling. If simplification is used to solve such goals, additional case-splitting is not allowed (as it may cause non-termination) which causes failure on proofs requiring several splits. A potential solution is to delay case-splitting until no more rewrites are applicable, thus avoiding complicated non-rippling goals. This does however require a modified ripple-measure which takes wave-front sizes into account.

9.3 Improvements for Conjecture Synthesis

IsaCoSy is still often quite slow, taking a couple of hours to finish, and sometimes fails to synthesise interesting theorems. A number of suggestions about how to further decrease the search space and perhaps improve the quality of theorems are outlined below.

9.3.1 Invalid Rewrite Rules and Term Orderings

The constraint mechanism in IsaCoSy currently 'assumes' that all theorems it is given are valid rewrite rules, orientated in the direction we intend to apply them (with the exception of simple commutativity theorems, which can be identified and are treated differently). It then deduces constraints based on the left-hand side of the rule. However, this is not completely correct, as we saw in §8.8.2, where the theorem

$$len(a \mathbin{@} b) = len(b \mathbin{@} a) \tag{9.1}$$

resulted in constraints excluding the theorem $len(a \mathbin{@} b) = (len\ a) + (len\ b)$. The offending theorem 9.1 is not a valid rewrite rule, as it can be applied infinitely many times to a matching term. Theorems that are not valid rewrite rules are however still

potentially interesting. We do not necessarily want to exclude them by, for example, disallowing terms with equal size left- and right-hand sides, which would ensure all synthesised terms were valid rewrite rules, under our simple size measure.

Ideally, the constraint generator should check if theorems given to it are valid rewrite rules, before generating constraints. This is however a non-trivial problem, many rewrite systems such as Isabelle's simplifier, rely on the user to provide it only with valid rules. A possible solution is to extend IsaCoSy with a total term ordering, such as a *recursive path ordering* [24]. Imposing a total order on terms would also get rid of symmetric theorems, where equations are included both ways around. Choosing a suitable order may also help orientating certain theorems in a direction that will result in constraints cutting down the large number of theorems synthesised about append and reverse in the list domain, as was discussed in §8.8.2.

As was mentioned in §8.8.2, theorems such as 9.1 above, should be identified as commutativity theorems, from which size constraints can be derived. IsaCoSy can currently only identify simple commutativity theorems, not ones where a function is commutative under some particular term-context, as in theorem 9.1. We suggest extending IsaCoSy's capabilities for identifying commutativity to cope with these kind of theorems.

9.3.2 Restrictions on Hole Sizes

The search space could potentially be cut down quite drastically if we reconsidered the assignment of allowed sizes for holes during synthesis. IsaCoSy will currently consider all possibilities where the left-hand side of an equation is of larger or equal size than the right. If we are to synthesise equational terms of size 10, this includes considering terms where the LHS is of size 7 or 8, and the RHS much smaller, only size 2 or 1. In our experiments, we found no theorems that had extremely large differences in size between left- and right-hand sides, suggesting that these extreme cases perhaps could be excluded. The largest difference between sides found was 4, for the theorem $rev((rev\ a)\ @\ [b]) = b\ \#\ a$. MATHsAiD has a heuristic formula for calculating the possible sizes for the right-hand side of an equation (see §8.9), excluding extreme cases.

9.3.3 Optimising Term Generation

IsaCoSy is currently generating whole terms, of given sizes. A possible optimisation is to cache sub-terms of each size, as they will be re-used in many term constructions. As theorems are discovered and more constraints generated, the cached terms no longer allowed must be filtered out. This way, sub-terms would only have to be synthesised once.

Currently, terms of a specified size are generated as a batch, followed by counter-example checking, proof and constraint generation for all of them. Individually checking each one after it has been synthesised, and discarding the non-theorems, would use less memory than the current approach.

9.3.4 Restricting Function Nesting

Currently, IsaCoSy produces a lot of silly terms, built from a large number of datatype constructors. A heuristic, limiting the depth of function nesting or specifying a maximum number of occurrences of a single symbol, would cut out many of these terms, for example, terms stacking up a large number of successor functions. This could be implemented as a heuristic much like the current option on how many different variables are allowed in one term (see §7.6).

9.3.5 Synthesis on Larger Theories

We have so far only applied synthesis to relatively small theories with just three or four different functions. If applied to larger theories, with many different functions, it would probably be beneficial to restrict the number of different function symbols occurring in the same term. Most theorems only contain a small number of different functions. Again, this could be implemented as a default heuristic value which the user can configure should he/she wish to do so.

9.4 Combining Critics and Synthesis

Synthesis could be used for instantiating meta-variables in proof critics, instead of middle-out reasoning. Critics for both lemma speculation and accumulator generalisation introduce meta-variables to stand for yet unknown terms structures [41].

9.4.1 Synthesis for Lemma Speculation

Instead of applying further rippling and middle-out reasoning, which may often have a large search space, an interesting alternative would be to let our synthesis algorithm suggest possible instantiations. An advantage of using synthesis rather than middle-out reasoning is that we no longer require intermediate ripple-steps between the application of the schematic lemma and fertilisation to instantiate the meta-variables. This was a major problem and caused failure in many of the examples in chapter 6.

The synthesis algorithm would need some modifications, for example, meta-variables may, in the critics context, sometimes need to 'disappear' by being instantiated to the identity function. Extra constraint machinery would also need to be built for restricting synthesis to only produce rules that decrease the ripple measure and preserve the skeleton. We may also want to restrict synthesis to only use the function symbols and datatypes occurring in the blocked goal. This avoids unrelated functions being introduced, which is unlikely to produce a useful lemma but will increase the search space. Other efficiency improvements, such as the ones described above, would perhaps also be required as a human user is unlikely to want to wait for a long time for a proof.

9.4.2 Synthesis and Accumulator Generalisation

As well as experimenting with using synthesis to instantiate meta-variables in an accumulator generalisation critic, a technique for identifying known theorems as generalisations would be useful. When a proof-attempt fails, a more general theorem might perhaps already have been found by IsaCoSy. As an example consider the proof of the theorem:

$$qrev\ a\ [\,] = rev\ a \tag{9.2}$$

IsaPlanner cannot currently prove 9.2 as it requires accumulator generalisation. When applied to a theory about *qrev*, IsaCoSy discovered and proved the correct generalisation (see table C.6):

$$qrev\ a\ b = (rev\ a)\ @\ b \tag{9.3}$$

A very simple generalisation critic could, after the failed proof-attempt of theorem 9.2, simply check if any other theorems could be used to rewrite the original goal. In this case, 9.3 can rewrite 9.2 to $(rev\ a)\ @[\,] = rev\ a$, which can easily be proved automatically.

9.5 Configuring IsaCoSy for Different Proof Techniques

Different rewriting-based proof techniques require slightly different sets of rules. For example, Isabelle's simplifier relies on its rewrite rules being orientated in such a way that rewriting terminates, while rippling terminates regardless of the directions of the rules. However, rippling requires skeleton preservation, and will sometimes need lemmas that are more specific versions of some theorems that IsaCoSy produces. An example of this was discussed in §8.5.

It would be interesting to experiment with different constraint configurations for IsaCoSy, attempting to synthesise a set of rewrite rules suitable for a particular technique. For rippling, it may, for example, be beneficial to allow theorems involving singleton lists even though more general theorems exist. For simplification, it would be interesting to attempt to configure IsaCoSy to produce a confluent set of rewrite rules, or perhaps combining it with Knuth-Bendix completion [50].

In addition to rippling and simplification, another area of further work is to consider rewriting modulo associativity and commutativity. Recall that we already impose constraints on the argument sizes of functions found to be commutative during synthesis (see §7.5.4). With an AC-rewriting technique, IsaCoSy would not have to consider synthesising commuted versions of theorems.

9.6 Future Applications

We believe the IsaCoSy program has the potential to be useful for assisting theory development as well as for generating challenge problems to test automated inductive theorem provers.

In the theory development setting, a synthesis tool could be applied to functions and datatypes defined by a user. It could then either be left to run to some finite level of completion (e.g. a specified maximum term size), or possibly left to run in the background, finding and proving routine lemmas that may be of use for later proofs. Alternatively, IsaCoSy could be called when the user is stuck in some proof. In this scenario, synthesis can be further restricted to build lemmas from constants and function symbols in the goal, as waiting for a long time in the middle of an interactive proof is not acceptable.

Secondly, a synthesis tool could be used to automatically generate test-sets for inductive theorem provers, perhaps for inclusion in a library such as TPTP [71].

9.7 Summary

The current implementation of IsaCoSy could potentially be improved and made more efficient in several ways. We propose extending IsaCoSy with a total term-ordering, to avoid accidentally generating constraints from invalid rewrite rules and to avoid synthesising symmetric theorems. Further improvements include optimisations to term generation, as well as restrictions on terms with large size differences between left- and right-hand sides, and on the occurrence of datatype constructors.

Another interesting area for further research is to combine synthesis and proof-critics. IsaCoSy could be used to find instantiations for meta-variables in schematic lemmas, instead of middle-out reasoning. We also suggested a very simple critic for accumulator generalisation, which applies a suitable generalisation from the previously synthesised background theory.

We believe that IsaCoSy can be further developed into a useful tool to assist theory development by finding routine lemmas. IsaCoSy could potentially be configured to synthesise sets of rewrite rules suited for a particular technique, such as simplification or rippling. Another possible application is automatic generation of test-sets for inductive theorem provers.

Chapter 10

Conclusions

10.1 Introduction

The aim of this project was to improve automation of inductive proofs by automating lemma discovery and case analysis. The hypothesis stated in chapter 1 is revisited below:

1. Theory formation by conjecture synthesis can be implemented in a computationally tractable fashion, and can produce useful theorems and lemmas, while lemma speculation is rarely applicable and produces few useful lemmas.

2. Automating case-analysis allows many theorems involving conditional statements to be proved automatically.

We will here summarise our work and discuss whether the hypotheses above have been verified.

10.2 Lemma Discovery

To explore techniques for lemma discovery we implemented both a lemma speculation critic and a program for conjecture synthesis within IsaPlanner.

10.2.1 Lemma Speculation

IsaPlanner's lemma speculation critic works in higher-order logic with dynamic rippling, unlike an earlier version implemented in CLAM 3 [41]. Unlike previous work,

it also guarantees termination of rippling in the presence of meta-variables by recomputing the ripple measures for the whole trace of middle-out steps as meta-variables get instantiated. Lemma speculation is applicable when rippling is blocked but the inductive hypothesis cannot yet be applied. It introduces meta-variables into the skeleton of the blocked term to create a schematic lemma. Meta-variables are then instantiated by further rippling-rewrites (and projections), until it is possible to apply the hypothesis. During evaluation, we found few proofs where lemma speculation was applicable. Furthermore, lemma speculation will fail, producing an underspecified lemma, in cases where the missing lemma is the last step before fertilisation. In these cases, there are no possible steps to help instantiate the meta-variables of the lemma.

Due to the negative results obtained, we do not suggest exploring the lemma speculation critic further, but rather focus on more commonly applicable techniques such as improvements to lemma calculation. More could probably be gained by adding support for reasoning with conditionals and combining lemma calculation with additional generalisation techniques, other than just common sub-term generalisation.

10.2.2 Conjecture Synthesis

The limitations of lemma speculation motivated the development of a conjecture synthesis program, called IsaCoSy. To make synthesis computationally feasible, we turn rewriting upside down, and only allow the synthesis of terms that do not match any of the current rewrite rules. This is enforced by generating constraints from theorems. The constraint limits further synthesis by restricting where functions and variables are allowed to occur. IsaCoSy filters out false conjectures by counter-example checking and passes the remaining conjectures to IsaPlanner for proof. As new theorems are found, more constraints are generated from these.

IsaCoSy was evaluated on several inductive theories about natural numbers, lists and binary trees. Compared to a naive version of synthesis, it manages an exponential reduction of the search space size. We compared the theorems found by IsaCoSy with those in the Isabelle's libraries (when available). IsaCoSy produces many good theorems, resulting in high recall of 83% for natural numbers and 100% for lists. It does however produce a number of less interesting theorems too, so precision is lower: 63% for natural numbers and 38% for lists. Using a synthesised background theory, we also showed that IsaPlanner is able to prove all but one of Ireland's examples of theorems previously requiring lemma speculation.

10.2.3 Verification of the Hypothesis

We consider these results as a good verification of part 1 of the hypothesis in §10.1. Our conjecture synthesis program does not produce nearly as many terms as a naive version and is able to generate large terms before encountering any memory issues. We showed that many useful theorems, occurring in libraries and useful in automated proofs, can be synthesised in a tractable fashion. However, we believe IsaCoSy can be made more efficient by considering the improvements suggested in chapter 9. Using a synthesised background theory allowed IsaPlanner to prove more of the theorems from the evaluation set for lemma speculation, than using the critic itself.

To conclude, theory formation by conjecture synthesis or extensions to lemma calculation seem to be the more promising areas to further explore automating lemma discovery for inductive theorem proving.

10.3 Case-Analysis

We implemented a case-analysis technique in IsaPlanner, capable of introducing splits when encountering an if- or case-statement for which the condition cannot be proved.

The second part of the hypothesis §10.1 was verified by evaluating the critic on a test set of 87 theorems, 47 of which were automatically proved. We compared this to a simple proof technique based on Isabelle's simplifier, which managed to prove 37 theorems. Our technique performed better on proofs requiring splits on a datatype. Isabelle's simplifier cannot perform such splits, as they may lead to non-termination. Our case-analysis technique is incorporated with rippling, and can thus retain termination.

We did not expect IsaPlanner's new case-analysis technique to prove all of the 40 remaining theorems as these require more sophisticated reasoning about, for example, side-conditions and the ability to construct conditional lemmas, which was suggested as further work.

10.4 Summary

Our experimental results support the hypotheses, showing that conjecture synthesis is capable of finding many useful theorems. Using a synthesised background theory allows IsaPlanner to prove theorems that would otherwise require lemma speculation, including some where the critic fails. As lemma speculation is rarely applicable, the-

ory formation by conjecture synthesise is currently a more powerful technique for automated discovery of inductive lemmas. Given the wide range of possible improvements, proposed in chapter 9, conjecture synthesis also seems to be the more promising direction for further work.

We also showed that our case-analysis technique allowed IsaPlanner to automatically prove a number of theorems involving conditional statements. IsaPlanner's capabilities for dealing with such theorems could be further improved by implementing a mechanism for constructing conditional lemmas, as well as more sophisticated reasoning about side-conditions.

Appendix A

Function Definitions

A.1 Natural Numbers

A.1.1 Arithmetic

fun $+$ :: $nat \Rightarrow nat \Rightarrow nat$

$$\text{add-zero:} \quad 0 + y = y$$

$$\text{add-suc:} \quad (Suc\ x) + y = Suc(x + y)$$

fun $*$ :: $nat \Rightarrow nat \Rightarrow nat$

$$\text{mult-zero:} \quad 0 * y = y$$

$$\text{mult-suc:} \quad (Suc\ x) * y = y + (x * y)$$

fun exp :: $nat \Rightarrow nat \Rightarrow nat$

$$\text{exp-zero:} \quad x^0 = Suc\ 0$$

$$\text{exp-suc:} \quad x^{Suc\ y} = x * x^y$$

fun $-$ :: $nat \Rightarrow nat \Rightarrow nat$

$$\text{minus-zero:} \quad 0 - y = 0$$

$$\text{minus-suc:} \quad (Suc\ x) - y = case\ y\ of\ 0 \Rightarrow Suc\ x \mid Suc\ z \Rightarrow x - z$$

A.1.2 Orders and Max

fun $<$ :: $nat \Rightarrow nat \Rightarrow bool$

$$\text{less-zero:} \quad x < 0 = False$$

$$\text{add-suc:} \quad x < Suc\ y = case\ x\ of\ 0 \Rightarrow True \mid Suc\ z \Rightarrow z < y$$

fun $\leq$ :: $nat \Rightarrow nat \Rightarrow bool$

lesseq-zero: $0 \leq y = True$

lesseq-suc: $Suc\ x \leq y = case\ y\ of\ 0 \Rightarrow False \mid Suc\ z \Rightarrow x \leq z$

fun max :: $nat \Rightarrow nat \Rightarrow nat$

max-zero: $max\ 0\ y = y$

max-suc: $max\ (Suc\ x)\ y = case\ y\ of\ 0 \Rightarrow (Suc\ x) \mid Suc\ z \Rightarrow Suc(max\ z\ y)$

A.1.3 Even

fun $even$ :: $nat \Rightarrow bool$

even-zero: $even\ 0 = True$

even-suc: $even\ (Suc(Suc\ x)) = even\ x$

A.2 Lists

A.2.1 Basics

fun @ :: $\alpha\ list \Rightarrow \alpha\ list \Rightarrow \alpha\ list$

app-nil: $[\,] @ l = l$

app-cons: $(h \# t) @ l = h \# (t @ l)$

fun len :: $\alpha\ list \Rightarrow nat$

len-nil: $len\ [\,] = 0$

len-cons: $len\ (h \# t) = Suc\ (len\ t)$

fun rev :: $\alpha\ list \Rightarrow \alpha\ list$

rev-nil: $rev\ [\,] = [\,]$

rev-cons: $rev(h \# t) = (rev\ t) @ [h]$

fun $qrev$ :: $\alpha\ list \Rightarrow \alpha\ list \Rightarrow \alpha\ list$

qrev-nil: $qrev\ [\,]\ l = l$

qrev-cons: $qrev(h \# t)\ l = qrev\ t\ (h \# l)$

fun $member :: \alpha \Rightarrow \alpha\ list \Rightarrow bool$

$$\begin{aligned} \text{mem-nil:} \quad & x\ member\ [\,] = False \\ \text{mem-cons:} \quad & x\ member\ (h \# t) = if\ (x = h)\ then\ True\ else\ (x\ member\ t) \end{aligned}$$

fun $count :: \alpha \Rightarrow \alpha\ list \Rightarrow nat$

$$\begin{aligned} \text{count-nil:} \quad & count\ x\ [\,] = 0 \\ \text{count-cons:} \quad & count\ x\ (h \# t) = if\ (x = h)\ then\ (1\ +\ (count\ x\ t))\ else\ (count\ x\ t) \end{aligned}$$

fun $concat :: \alpha\ list\ list \Rightarrow \alpha\ list$

$$\begin{aligned} \text{concat-nil:} \quad & concat\ [\,] = [\,] \\ \text{concat-cons:} \quad & concat\ (h \# t) = h\ @\ (concat\ t) \end{aligned}$$

fun $zip :: \alpha\ list \Rightarrow \beta\ list \Rightarrow (\alpha * \beta)\ list$

$$\begin{aligned} \text{zip-nil:} \quad & zip\ l\ [\,] = [\,] \\ \text{zip-cons:} \quad & zip\ l\ (h_2 \# t_2) = case\ l\ of\ [\,] \Rightarrow [\,] \mid (h_1 \# t_1) \Rightarrow ((h_1, h_2) \# (zip\ t_1\ t_2)) \end{aligned}$$

A.2.2 Higher-Order Functions

fun $map :: (\alpha \Rightarrow \beta) \Rightarrow \alpha\ list \Rightarrow \beta\ list$

$$\begin{aligned} \text{map-nil:} \quad & map\ f\ [\,] = [\,] \\ \text{map-cons:} \quad & map\ f\ (h \# t) = (f\ h) \# t \end{aligned}$$

fun $maps :: (\alpha \Rightarrow \beta\ list) \Rightarrow \alpha\ list \Rightarrow \beta\ list$

$$\begin{aligned} \text{maps-nil:} \quad & maps\ f\ [\,] = [\,] \\ \text{maps-cons:} \quad & maps\ f\ (h \# t) = (f\ h)\ @\ t \end{aligned}$$

fun $filter :: (\alpha \Rightarrow bool) \Rightarrow \alpha\ list \Rightarrow \alpha\ list$

$$\begin{aligned} \text{filter-nil:} \quad & filter\ P\ [\,] = [\,] \\ \text{filter-cons:} \quad & filter\ P\ (h \# t) = if\ (P\ h)\ then\ (h \#\ filter\ P\ t)\ else\ (filter\ P\ t) \end{aligned}$$

fun $foldl :: (\alpha \Rightarrow \beta \Rightarrow \alpha) \Rightarrow \alpha \Rightarrow \beta\ list \Rightarrow \alpha$

$$\begin{aligned} \text{foldl-nil:} \quad & foldl\ f\ a\ [\,] = a \\ \text{foldl-cons:} \quad & foldl\ f\ a\ (h \# t) = foldl\ f\ (f\ a\ h)\ t \end{aligned}$$

fun $foldr :: (\alpha \Rightarrow \beta \Rightarrow \beta) \Rightarrow \alpha\ list \Rightarrow \beta \Rightarrow \beta$

foldr-nil: $foldr\ f\ [\,]\ a = a$

foldr-cons: $foldr\ f\ (h \# t)\ a = f\ h\ (foldr\ f\ t\ a)$

A.2.3 Insertertion and Deletion

fun $insert :: nat \Rightarrow (nat\ list) \Rightarrow (nat\ list)$

insert-nil: $insert\ x\ [\,] = [x]$

insert-cons: $insert\ x\ (h \# t) = if\ (x < h)\ then\ (x \# h \# t)\ else\ (insert\ x\ t)$

fun $insert\_1 :: nat \Rightarrow (nat\ list) \Rightarrow (nat\ list)$

insert_1-nil: $insert\_1\ x\ [\,] = [x]$

insert_1-cons: $insert\_1\ x\ (h \# t) = if\ (x = h)\ then\ (x \# t)\ else\ (h \#\ (insert\_1\ x\ t))$

fun $delete :: \alpha \Rightarrow \alpha\ list \Rightarrow \alpha\ list$

del-nil: $delete\ x\ [\,] = [\,]$

del-cons: $delete\ x\ (h \# t) = if\ (x = h)\ then\ (delete\ x\ t)\ else\ (h \#\ delete\ x\ t)$

A.2.4 Sorting

fun $sort :: nat\ list \Rightarrow nat\ list$

sort-nil: $sort\ [\,] = [\,]$

sort-cons: $sort\ (h \# t) = insert\ h\ (sort\ t)$

fun $sorted :: nat\ list \Rightarrow bool$

sorted-nil: $sorted\ [\,] = True$

sorted-cons: $sorted\ (h \# t) = case\ t\ of\ [\,] \Rightarrow True \mid (h_2 \# t_2) \Rightarrow (h \leq h_2) \wedge sorted\ (h_2 \# t_2)$

A.2.5 Last and Butlast

fun $last :: \alpha\ list \Rightarrow \alpha$

last-cons: $last\ (h \# t) = if\ (t = [\,])\ then\ h\ else\ (last\ t)$

fun $butlast :: \alpha\ list \Rightarrow \alpha\ list$

butlast-nil: $butlast\ [\,] = [\,]$

butlast-cons: $butlast\ (h \# t) = if\ (t = [\,])\ then\ [\,]\ else\ (h \#\ butlast\ t)$

A.2.6 Take and Drop

fun $drop$ :: $nat \Rightarrow \alpha\ list \Rightarrow \alpha\ list$

drop-nil: $drop\ n\ [\,] = [\,]$

drop-cons: $drop\ n\ (h \# t) = case\ n\ of\ 0 \Rightarrow (h \# t) \mid (Suc\ m) \Rightarrow drop\ m\ t$

fun $dropWhile$ :: $(\alpha \Rightarrow bool) \Rightarrow \alpha\ list \Rightarrow \alpha\ list$

dropWhile-nil: $dropWhile\ P\ [\,] = [\,]$

dropWhile-cons: $dropWhile\ P\ (h \# t) = if\ (P\ h)\ then\ (dropWhile\ P\ t)\ else\ (h \# t)$

fun $take$ :: $nat \Rightarrow \alpha\ list \Rightarrow \alpha\ list$

take-nil: $take\ n\ [\,] = [\,]$

take-cons: $take\ n\ (h \# t) = case\ n\ of\ 0 \Rightarrow [\,] \mid (Suc\ m) \Rightarrow h \# take\ m\ t$

fun $takeWhile$ :: $(\alpha \Rightarrow bool) \Rightarrow \alpha\ list \Rightarrow \alpha\ list$

takeWhile-nil: $takeWhile\ P\ [\,] = [\,]$

takeWhile-cons: $takeWhile\ P\ (h \# t) = if\ (P\ h)\ then\ (h \# takeWhile\ P\ t)\ else\ [\,]$

A.3 Binary Trees

datatype $\alpha\ Tree =$

Leaf

| Node $of\ \alpha\ Tree * \alpha * \alpha\ Tree$

fun $mirror$ :: $\alpha\ Tree \Rightarrow \alpha\ Tree$

mirror-leaf: $mirror\ Leaf = Leaf$

mirror-node: $mirror\ (Node\ l\ data\ r) = Node\ (mirror\ r)\ data\ (mirror\ l)$

fun $nodes$ :: $\alpha\ Tree \Rightarrow nat$

nodes-leaf: $nodes\ Leaf = 0$

nodes-node: $nodes\ (Node\ l\ data\ r) = (Suc\ 0) + (nodes\ l) + (nodes\ r)$

fun $height$ :: $\alpha\ Tree \Rightarrow nat$

height-leaf: $height\ Leaf = 0$

height-node: $height\ (Node\ l\ data\ r) = Suc(max\ (height\ l)\ (height\ r))$

Appendix B

Experimental Results for Case Analysis

The evaluation corpus consists of 87 theorems about function defined with if- and case-statements. The 47 theorems in table B.1 can be proved by IsaPlanner by rippling with our case-analysis technique, while table B.2 shows the remaining 40 theorems where IsaPlanner fails. For the experiments, rippling (and simplification) was only supplied with the definitions given in Appendix A, and no extra lemmas.

The run-times are given in seconds. The 'Src' column indicates where the theorem came from: 'Isa' is Isabelle's library, 'Ire' is the paper about proof-critics by Ireland and Bundy [41] and 'Wil' is the paper by Wilson and Fleuriot about inductive proofs arising from dependent types [76]. Theorems with no such label have been added by the author to evaluate additional properties.

No	Theorem	Time	Cond	Src
1	$m - m = 0$	0.068	case	Isa
2	$n - (n + m) = 0$	0.174	case	Isa
3	$(n + m) - n = m$	0.177	case	Isa
4	$(k + m) - (k + n) = m - n$	0.079	case	Isa
5*	$(i - j) - k = i - (j + k)$	0.270	case	Isa
6	$n \leq 0 \leftrightarrow (n = 0)$	0.065	case	Isa
7	$n \leq (n + m)$	0.168	case	Isa
8	$i < Suc\ (i + m)$	0.251	case	Isa
9*	$max\ a\ b = max\ b\ a$	0.254	case	
10*	$max\ (max\ a\ b)\ c = max\ a\ (max\ b\ c)$	0.632	case	

11*	$(max\ a\ b = a) = b \leq a$	0.262	case	
12*	$max\ a\ b = b) = a \leq b$	0.270	case	
13*	$min\ a\ b = min\ b\ a$	0.259	case	
14*	$min\ (min\ a\ b)\ c = min\ a\ (min\ b\ c)$	0.646	case	
15*	$(min\ a\ b = a) = a \leq b$	0.265	case	
16*	$min\ a\ b = b) = b \leq a$	0.261	case	
17	$drop\ 0\ xs = xs$	0.068	case	Isa
18	$drop\ (Suc\ n)\ (x\ \#\ xs) = drop\ n\ xs$	0.073	case	Isa
19*	$drop\ n\ (map\ f\ xs) = map\ f\ (drop\ n\ xs)$	0.200	case	Isa
20*	$len\ drop\ n\ xs = (len\ xs) - n$	0.360	case	Isa
21	$take\ 0\ xs = []$	0.068	case	Isa
22	$take\ (Suc\ n)\ (x\ \#\ xs) = x\ \#\ take\ n\ xs$	0.072	case	Isa
23*	$take\ n\ (map\ f\ xs) = map\ f\ (take\ n\ xs)$	0.218	case	Isa
24*	$take\ n\ xs\ @\ drop\ n\ xs = xs$	0.284	case	Isa
25	$zip\ []\ ys = []$	0.078	case	Isa
26	$zip\ (x\ \#\ xs)\ ys = case\ ys\ of\ [] \Rightarrow []$ $\mid (z\ \#\ zs) \Rightarrow (x, z)\ \#\ zip\ xs\ zs)$	0.222	case	Isa
27	$zip\ (x\ \#\ xs)\ (y\ \#\ ys) = (x, y)\ \#\ zip\ xs\ ys$	0.109	case	Isa
28*	$height\ (mirror\ t) = height\ a$	0.282	case	
29	$x\ member\ (l\ @\ [x])$	0.023	if	Isa
30	$\neg\ (x\ member\ (delete\ x\ l))$	0.028	if	
31	$x\ member\ l \rightarrow x\ member\ (l\ @\ t)$	0.069	if	Ire
32	$x\ member\ t \rightarrow x\ member\ (l\ @\ t)$	0.025	if	Ire
33	$x\ member\ (insert\ x\ l)$	0.029	if	Ire
34	$x\ member\ (insert\_1\ x\ l)$	0.027	if	Ire
35	$len\ insert\ x\ l = Suc\ (len\ l)$	0.035	if	Ire
36	$len\ (sort\ l) = len\ l$	0.047	if	Ire
37	$xs = [] \Longrightarrow last\ (x\ \#\ xs) = x$	0.012	if	Isa
38	$(Suc\ 0) + (count\ n\ l) = count\ n\ (n\ \#\ l)$	0.086	if	
39	$n = x \Longrightarrow (Suc\ 0) + count\ n\ l = count\ n\ (x\ \#\ l)$	0.132	if	
40	$count\ n\ l + count\ n\ m = count\ n\ (l\ @\ m)$	0.131	if	
41	$count\ n\ (x\ @\ [n]) = Suc\ (count\ n\ x)$	0.064	if	Wil
42	$count\ n\ [h] + count\ n\ t = count\ n\ (h\ \#\ t)$	0.144	if	Wil
43	$count\ n\ l \leq count\ n\ (l\ @\ m)$	0.200	if	
44	$dropWhile\ (\lambda\ x.\ False)\ xs = xs$	0.011	if	

45	*takeWhile (λ x. True) xs = xs*	0.009	if	
46	*takeWhile P xs @ dropWhile P xs = xs*	0.500	if	Isa
47	*filter P (xs @ ys) = filter P xs @ filter P ys*	0.069	if	Isa

Table B.1: Theorems proved automatically by IsaPlanner using the case-analysis. Run-times are given in seconds. The 14 theorems marked with * require a case-split on a datatype, of the kind Isabelle's simplifier cannot perform.

No	Theorem	Time	Cond	Src
48	$(m + n) - n = m$	0.184	case	Isa
49	$((Suc\ m) - n) - (Suc\ k) = (m - n) - k$	0.464	case	Isa
50	$i < Suc\ (m + i)$	0.490	case	Isa
51	$n \leq (m + n)$	0.174	case	Isa
52	$m \leq n \Longrightarrow m \leq Suc\ n$	0.071	case	
53	$drop\ n\ (drop\ m\ xs) = drop\ (n + m)\ xs$	0.294	case	Isa
54	$drop\ n\ (xs\ @\ ys) = drop\ n\ xs\ @\ drop\ (n - (len\ xs))\ ys$	1.968	case	Isa
55	$drop\ n\ (take\ m\ xs) = take\ (m - n)\ (drop\ n\ xs)$	0.761	case	Isa
56	$drop\ n\ (zip\ xs\ ys) = zip\ (drop\ n\ xs)\ (drop\ n\ ys)$	0.860	case	Isa
57	$rev\ (drop\ i\ xs) = take\ ((len\ xs) - i)\ (rev\ xs)$	0.783	case	Isa
58	$rev\ (take\ i\ xs) = drop\ ((len\ xs) - i)\ (rev\ xs)$	0.777	case	Isa
59	$rev\ (filter\ P\ xs) = filter\ P\ (rev\ xs)$	0.096	if	Isa
60	$take\ n\ (xs\ @\ ys) = take\ n\ xs\ @\ take\ (n - (len\ xs))\ ys$	1.986	case	Isa
61	$take\ n\ (drop\ m\ xs) = drop\ m\ (take\ (n + m)\ xs)$	0.691	case	Isa
62	$take\ n\ (zip\ xs\ ys) = zip\ (take\ n\ xs)\ (take\ n\ ys)$	0.726	case	Isa
63	$(len\ filter\ P\ xs) \leq (len\ xs)$	0.336	if	Isa
64	$zip\ (xs\ @\ ys)\ zs = zip\ xs\ (take\ (len\ xs)\ zs)\ @$ $zip\ ys\ (drop\ (len\ xs)\ zs)$	0.730	case	Isa
65	$zip\ xs\ (ys\ @\ zs) = zip\ (take\ (length\ ys)\ xs)\ ys\ @$ $zip\ (drop\ (length\ ys)\ xs)\ zs$	9.439	case	Isa
66	$(len\ xs = len\ ys) \Longrightarrow zip\ (rev\ xs)\ (rev\ ys) =$ $rev\ (zip\ xs\ ys)$	0.427	case	Isa
67	$(len\ delete\ x\ l) \leq (len\ l)$	0.352	if	
68	$x < y \Longrightarrow x\ member\ (insert\ y\ l) = x\ member\ l$	0.03	if	
69	$x \neq y \Longrightarrow x\ member\ (insert\ y\ l) = x\ member\ l$	0.084	if	Ire

70	$sorted\ l \Longrightarrow sorted\ (insert\ x\ l)$	0.023	if/case	Ire
71	$sorted\ (sort\ l)$	0.005	if/case	Ire
72	$last\ (xs\ @\ [x]) = x$	0.029	if	Isa
73	$xs \neq [] \Longrightarrow last\ (x\ \#\ xs) = last\ xs$	0.08	if	Isa
74	$ys = [] \Longrightarrow last\ (xs\ @\ ys) = last\ xs$	0.07	if	Isa
75	$ys \neq [] \Longrightarrow last\ (xs\ @\ ys) = last\ ys$	0.038	if	Isa
76	$last\ (xs\ @\ ys) = if\ (ys = [])$ $then\ (last\ xs)\ else\ (last\ ys)$	0.249	if	Isa
77	$n < (len\ xs) \Longrightarrow last\ (drop\ n\ xs) = last\ xs$	0.147	if/case	Isa
78	$butlast\ (xs\ @\ [x]) = xs$	0.026	if	Isa
79	$xs \neq [] \Longrightarrow butlast\ xs\ @\ [last\ xs] = xs$	0.005	if	Isa
80	$butlast\ (xs\ @\ ys) = if\ ys = []$ $then\ (butlast\ xs)\ else\ (xs\ @\ butlast\ ys)$	0.268	if	Isa
81	$butlast\ xs = take\ ((len\ xs)\ -\ (Suc\ 0))\ xs$	2.410	if/case	Isa
82	$len\ butlast\ xs = (len\ xs)\ -\ (Suc\ 0)$	0.756	if/case	Isa
83	$(len\ delete\ x\ l) \leq (len\ l)$	0.352	if	
84	$count\ n\ t + count\ n\ [h] = count\ n\ (h\ \#\ t)$	1.827	if	Wil
85	$count\ n\ l = count\ n\ (rev\ l)$	0.075	if	
86	$count\ x\ l = count\ x\ (sort\ l)$	0.075	if	Ire
87	$n \neq h \Longrightarrow count\ n\ (x\ @\ [h]) = count\ n\ x$	0.216	if	Wil

Table B.2: 40 theorems IsaPlanner fails to prove. Run-times (until failure) are given in seconds.

Appendix C

Experimental Results for Conjecture Synthesis

The tables below show the theorems synthesised by IsaCoSy for the six evaluation theories from chapter 8, about natural numbers, lists and binary trees. The theorems marked 'Pre-Processing' in the tables below have been discovered by IsaCoSy's heuristic which attempts to find associativity and commutativity properties prior to synthesis.

Label	Size	Theorem
T1	Pre-processing	*max a b = max b a*
T2	Pre-processing	*max (max a b) c = max a (max b c)*
T3	5	$max\ a\ a = a$
T4	5	$mirror\ (mirror\ a) = a$
T5	6	$nodes\ (mirror\ a) = nodes\ a$
T6	6	$height\ (mirror\ a) = height\ a$

Table C.1: Theorems found about binary trees with functions max, mirror, nodes and height. Isabelle has no library for binary trees.

Label	Size	Theorem
N1*	Pre-processing	$a + 0 = a$
N2*	Pre-processing	$a + Suc\ b = Suc(a + b)$
N3*	Pre-processing	$a * 0 = 0$
N4*	Pre-processing	$a * Suc\ b = a + (a * b)$
N5*	Pre-processing	$a + b = b + a$
N6*	Pre-processing	$a * b = b * a$
N7*	Pre-processing	$(a + b) + c = a + (b + c)$
N8*	Pre-processing	$(a * b) * c = a * (b * c)$
N9*	13	$(a * b) + (c * b) = (a + c) * b$
N10	13	$(a * b) + (c * a) = (b + c) * a$
N11	13	$(a * b) + (c * a) = (c + b) * a$
N12	13	$(a * b) + (c * b) = (c + a) * b$
N13*	13	$(a * b) + (a * c) = (b + c) * a$
N14	13	$(a * b) + (a * c) = (c + b) * a$
N15	13	$(a * b) + (b * c) = (a + c) * b$
N16	13	$(a * b) + (b * c) = (c + a) * b$

Table C.2: Theorems discovered about addition and multiplication on the natural numbers. Theorems marked with * are included in Isabelle's library for natural numbers. Note that Isabelle has the equations orientated in the opposite direction for N9 and N13. N13 is furthermore commuted over multiplication, e.g. the RHS of N13 is $(b + c) * a$, while in Isabelle it is $a * (b + c)$.

Label	Size	Theorem
L1*	Pre-processing	$(a \ @ \ b) \ @ \ c = a \ @ \ (b \ @ \ c)$
L2*	5	$rev(rev\ a) = a$
L3*	5	$a \ @ \ [\,] = a$
L4*	6	$len(rev\ a) = len\ a$
L5	9	$len(a \ @ \ b) = len(b \ @ \ a)$
L6	10	$rev(a \ @ \ (rev\ b)) = b \ @ \ (rev\ a)$
L7*	10	$(rev\ a) \ @ \ (rev\ b) = rev\ (b \ @ \ a)$
L8	10	$rev((rev\ a) \ @ \ (rev\ b)) = b \ @ \ a$
L9	11	$rev(a \ @ \ [b]) = b \ \# \ (rev\ a)$
L10	11	$rev((rev\ a) \ @ \ [b]) = b \ \# \ a$
L11	14	$rev(a \ @ \ (b \ @ \ (rev\ c))) = c \ @ \ (rev(a \ @ \ b))$
L12	14	$rev(a \ @ \ (b \ \# \ (rev\ c))) = c \ @ \ (b \ \# \ (rev\ a))$
L13	14	$rev((rev\ a) \ @ \ (b \ \# \ c)) = (rev\ c) \ @ \ (b \ \# \ a)$
L14	14	$(rev\ a) \ @ \ ((rev\ b) \ @ \ c) = (rev\ (b \ @ \ a)) \ @ \ c$
L15	14	$(rev\ a) \ @ \ (b \ \# \ (rev\ c)) = rev(c \ @ \ (b \ \# \ a))$
L16	14	$rev((rev\ a) \ @ \ b) \ @ \ c = (rev\ b) \ @ \ (a \ @ \ c)$
L17	14	$rev((rev\ a) \ @ \ (b \ \# \ (rev\ c))) = c \ @ \ (b \ \# \ a)$
L18	14	$a \ @ \ (rev(rev\ b) \ @ \ c) = a \ @ \ ((rev\ c) \ @ \ b)$

Table C.3: Theorems discovered about append, reverse and length on lists. Theorems marked with * are included in Isabelle's list library. Note that L18 is allowed to be synthesised as its simpler version, $rev(rev\ b) \ @ \ c = (rev\ c) \ @ \ b$, could not be proved when it was first synthesised, and thus did not generate constraints. Its proof require L7 as a lemma, which was not yet available. The lemma is however available when attempting to prove L18, so this succeeds.

Label	Size	Theorem
L19	9	$map\ a\ (rev\ b) = rev(map\ a\ b)$
L20*	9	$rev(map\ a\ b) = map\ a(rev\ b)$
L21	9	$rev(map\ a\ (rev\ b)) = map\ a\ b$
L22*	13	$(map\ a\ b) \ @ \ (map\ a\ c) = map\ a\ (b \ @ \ c)$

Table C.4: Additional theorems discovered about append, reverse and map on lists. The theorems about append and reverse from table C.3 (theorems L1 - L3 and L6 - L18) were re-discovered. Theorems marked with * are included in Isabelle's list library.

Label	Size	Theorem
L42*	14	*foldl a (foldl a b c) d = foldl a b (c @ d)*
L43*	14	*foldr a b (foldr a c d) = foldr a (b @ c) d*

Table C.5: Additional theorems discovered about foldl and foldr. Both theorems are included in Isabelle's list library. The theorems about append and reverse from table C.3 (theorems L1 - L3 and L6 - L18) were re-discovered.

Label	Size	Theorem
L23	8	$qrev\ (rev\ a)\ b = a\ @\ b$
L24	8	$(rev\ a)\ @\ b = qrev\ a\ b$
L25	9	$qrev\ (qrev\ a\ b)[\,] = qrev\ b\ a$
L26	11	$qrev\ a\ (qrev\ b\ c) = qrev\ (b\ @\ a)\ c$
L27	11	$qrev\ a\ (b\ @\ c) = qrev\ (qrev\ b\ a)\ c$
L28	11	$qrev\ a\ (b\ @\ c) = (qrev\ a\ b)\ @\ c$
L29	11	$qrev\ (qrev\ a\ b)\ c = qrev\ b\ (a\ @\ c)$
L30	11	$qrev\ (a\ @\ b)\ c = qrev\ b\ (qrev\ a\ c)$
L31	11	$(qrev\ a\ b)\ @\ c = qrev\ a\ (b\ @\ c)$
L32	12	$rev\ (qrev\ a\ (b\ \#\ c)) = qrev\ c\ (b\ \#\ a)$
L33	12	$a\ @\ (rev\ (qrev\ b\ [\,])) = rev\ (qrev\ b\ (rev\ a))$
L34	13	$rev\ (a\ @\ (b\ @\ c)) = qrev\ c\ (rev\ (a\ @\ b))$
L35	13	$rev\ (a\ @\ (b\ \#\ c)) = qrev\ c\ (b\ \#\ (rev\ a))$
L36	13	$qrev\ a\ (rev\ (b\ @\ c)) = rev\ (b\ @\ (c\ @\ a))$
L37	13	$qrev\ a\ (b\ \#\ (rev\ c)) = rev\ (c\ @\ (b\ \#\ a))$
L38	13	$rev\ (qrev\ a\ (rev\ (b\ @\ c))) = b\ @\ (c\ @\ a)$
L39	13	$a\ @\ (rev\ (b\ @\ c)) = a\ @\ qrev\ c\ (rev\ b)$
L40	13	$a\ @\ qrev\ b\ (rev\ c) = a\ @\ (rev\ (c\ @\ b))$
L41	13	$a\ @\ rev\ (qrev\ b\ (rev\ c)) = a\ @\ (c\ @\ b)$

Table C.6: Additional theorems discovered about append, reverse and qrev on lists. The theorems about append and reverse from table C.3 (theorems L1 - L3 and L6 - L18) were re-discovered. The qrev-function is not defined in Isabelle's list library, so no comparison can be made. Note that theorems L39 - L41 are allowed to be synthesised as their simpler versions (excluding the $a\ @\ldots$ on both sides) could not be proved. However, other proofs will later discover the required lemmas by lemma calculation. These lemmas are stored, so IsaPlanner will be able to re-use them to prove L39 - L41.

Bibliography

[1] M. Alderhold. Improvements in formula generalisation. In F. Pfenning, editor, *Proceedings of CADE'07*, volume 4603 of *Lecture Notes in Artificial Intelligence*, pages 231–246. Springer-Verlag, 2007.

[2] D. Basin and T. Walsh. A calculus for and termination of rippling. *Journal of Automated Reasoning*, 16(1-2):147–180, 1996.

[3] C. Benzmüller, L. Cheikhrouhou, D. Fehrer, A. Fiedler, X. Huang, M. Kerber, M. Kohlhase, K. Konrad, E. Melis, A. Meier, W. Schaarschmidt, J. Siekmann, and V. Sorge. Omega: Towards a mathematical assistant. In *Proceedings of 14th International Conference on Automated Deduction (CADE-14)*, pages 252–255. Springer Verlag, 1997.

[4] R. Boulton, K. Slind, A. Bundy, and M. Gordon. An interface between CLAM and HOL. In *Theorem Proving in Higher Order Logics*, Lecture Notes in Computer Science, pages 87–104. Springer Verlag, 1998.

[5] R.S. Boyer and J.S. Moore. *A Computational Logic*. Academic Press, 1979.

[6] B. Buchberger, A. Craciun, T Jebelean, L. Kovacs, T. Kutsia, K. Nakagawa, F. Piroi, N. Popov, J. Robu, M. Rosenkrantz, and W. Windsteiger. Theorema: Towards computer-aided mathematical theory exploration. *Journal of Applied Logic*, 4(4):470–504, 2006.

[7] A. Bundy. *Computer Modelling of Mathematical Reasoning*. Academic Press, 1983.

[8] A. Bundy. The use of explicit plans to guide inductive proofs. In *9th International Conference on Automated Deduction*, pages 111–120, 1988.

[9] A. Bundy. The automation of proof by mathematical induction. In A. Voronkov and A. Robinson, editors, *Handbook of Automated Reasoning*, chapter 13. MIT

Press, 2001. Also available from the University of Edinburgh as research report EDI-INF-RR-0002.

[10] A. Bundy, D. Basin, D. Hutter, and A. Ireland. *Rippling: Meta-level Guidance for Mathematical Reasoning*. Cambridge University Press, 2005.

[11] A. Bundy, A. Smaill, and J. Hesketh. Turning Eureka steps into calculations in automatic program synthesis. In *Proceedings of UK IT 90*, pages 221–226, 1990. Also available from the University of Edinburgh as DAI research paper 448.

[12] A. Bundy, F. van Harmelen, J. Hesketh, and A. Smaill. Experiments with proof plans for induction. *Journal of Automated Reasoning*, 7(3):303–324, 1992.

[13] A. Bundy, F. van Harmelen, C. Horn, and A. Smaill. The Oyster-Clam system. In *10th International Conference on Automated Deduction*, volume 449 of *Lecture Notes in Artificial Intelligence*, pages 647–648, 1990.

[14] F. Cantu, A. Bundy, A. Smaill, and D. Basin. Experiments in automating hardware verification using inductive proof planning. In *First International Conference on Formal Methods in Computer-Aided Design*, volume 1166 of *Lecture Notes in Computer Science*, pages 94–108. Springer-Verlag, 1996.

[15] C. Castellini and A. Smaill. Proof planning for first-order temporal logic. In *Automated Deduction - CADE 2005*, pages 235–249, 2005.

[16] F. Cavallo. Vanity theorem proving. Final Year Undergaduate Dissertation, University of Edinburgh, 2009.

[17] A. Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2), 1936.

[18] S. Colton. *Automated Theory formation in pure mathematics*. Springer-Verlag, 2002.

[19] S. Colton and G. Sutcliffe. Automatic generation of benchmark problems for automated theorem proving systems. In *Proceedings of the 7th Symposium on Artificial Intelligence and Mathematics*, 2002.

[20] R.L. Constable, S.F. Allen, H.M. Bromley, W.R. Cleaveland, J.F. Creamer, 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 Proof Development System*. Prentice Hall, 1985.
http://www.cs.cornell.edu/Info/Projects/NuPrl/book/doc.html.

[21] A. Cook, A. Ireland, and G. Michaelson. Higher order function synthesis through proof planning. In *Proceedings of 16th Annual International Conference on Automated Software Engineering ASE 2001*, pages 307–310, 2001.

[22] L. Dennis, A. Bundy, and I. Green. Using a generalisation critic to find bisimulations for coinductive proofs. In *Proceedings of CADE'97*, pages 276–290, 1997.

[23] L. Dennis, I. Green, and A. Smaill. Embeddings as a higher-order representation of annotations for rippling. Technical Report Computer Science No. NOTTCS-WP-SUB-0503230955-5470, University of Nottingham, 2005.

[24] N. Dershowitz. Orderings for term-rewriting systems. In *20th Annual Symposium on Foundations of Computer Science*, pages 123–131, 1979.

[25] L. Dixon. *A generic approach to proof planning*. PhD thesis, School of Informatics, University of Edinburgh, 2005.

[26] L. Dixon and J. Fleuriot. IsaPlanner: A prototype proof planner in Isabelle. In *Proceedings of CADE'03*, pages 279–283, 2003.

[27] L. Dixon and J. Fleuriot. Higher-order rippling in IsaPlanner. In *Proceedings of TPHOLs'04*, pages 83–98, 2004.

[28] G. Dowrek. Higher-order unification and matching. In A Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, chapter 16, pages 1011–1061. Elsevier, 2001.

[29] J. Gow. *The dynamic creation of Induction Rules using proof planning*. PhD thesis, University of Edinburgh, 2004.

[30] J. Hesketh. *Using middle-out reasoning to guide inductive theorem proving*. PhD thesis, University of Edinburgh, 1991.

[31] J. Hesketh, A. Bundy, and A. Smaill. Using middle-out reasoning to control the synthesis of tail-recursive programs. In *Autoamted Deduction - CADE-11*, pages 310–324. Springer-Verlag, 1992.

[32] M. Hodorog and A. Craciun. Scheme-based systematic exploration of natural numbers. In *Proceedings of the 8th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pages 26–34, 2006.

[33] G. Huet. A unification algorithm for typed lambda-calculus. *Journal of Theoretical Computer Science*, 1(1):27–57, 1975.

[34] G. Huet. The zipper. *Journal of Functional Programming*, 7(5):549–554, 1997.

[35] B. Hummel. An investigation of formula generalization heuristics for inductive proofs. Interner Bericht Nr, 6/87, Universität Karlsruhe, 1987.

[36] D. Hutter. Coloring terms to control equational reasoning. *Journal of Automated Reasoning*, 18(3):399–442, 1997.

[37] D. Hutter, H. Mantel, G. Rock, W. Stephan, A. Wolpers, M. Balser, W. Reif, G. Schellhorn, and K. Stenzel. VSE: Controlling the complexity in formal software developments. In *Applied Formal Methods - FM-Trends 98*, volume 1641 of *Lecture Notes in Computer Science*, pages 351–358. Springer Verlag, 1999.

[38] D. Hutter and C. Sengler. INKA: The next generation. In *Conference on Automated Deduction – CADE-13*, number 1104 in Lecture Notes in Computer Science, pages 288–292. Springer-Verlag, 1996.

[39] A. Ireland. The use of planning critics in mechanizing inductive proofs. In *LPAR '92: Proceedings of the International Conference on Logic Programming and Automated Reasoning*, pages 178–189, London, UK, 1992. Springer-Verlag.

[40] A. Ireland and A. Bundy. Extensions to a generalization critic for inductive proof. In *CADE-13: Proceedings of the 13th International Conference on Automated Deduction*, pages 47–61, London, UK, 1996. Springer-Verlag.

[41] A. Ireland and A. Bundy. Productive use of failure in inductive proof. *Journal of Automated Reasoning*, 16(1-2):79–111, 1996.

[42] A. Ireland, B. Ellis, A. Cook, R. Chapman, and J. Barnes. An integrated approach to program reasoning. *Journal of Automated Reasoning*, 36(4):379–410, 2006.

[43] A. Ireland, B. Ellis, and T. Ingulfsen. Invariant patterns for program reasoning. In *MICAI-03: Proceedings of the 3rd Mexican International Conference on Artificial Intelligence*, pages 190–201. Springer-Verlag, 2004.

[44] A. Ireland, M. Jackson, and G. Reid. Interactive proof critics. *Formal Aspects of Computing*, 11(3):302–325, 1999.

[45] M. Jackson. *Interaction with semi-automated theorem provers*. PhD thesis, School of Computing, Napier University, 1999.

[46] M. Johansson, A. Bundy, and L. Dixon. Best-first rippling. In O. Stock and M. Schaerf, editors, *Reasoning, Action and Interaction in AI theories and Systems: Essays dedicated to Luigia Carlucci Aiello*, number 4155 in Lecture Notes in Artificial Intelligence. Springer-Verlag, 2006.

[47] D. Kapur and N. A. Sakhanenko. Automatic generation of generalization lemmas for proving properties of tail-recursive definitions. In *Proceedings of TPHOLs'03*, pages 136–154, 2003.

[48] D Kapur and M. Subramaniam. Lemma discovery in automating induction. In *Proceedings of CADE'96*, pages 538–552, 1996.

[49] M. Kaufmann and J.S. Moore. An industrial strength theorem prover for a logic based on common LISP. *IEEE Transactions on Software Engineering*, 23(4):203–213, 1997.

[50] D.E. Knuth and P.B. Bendix. Simple word problems in universal algebras. *Computational Problems in Abstract Algebra*, 1967.

[51] I. Kraan, D. Basin, and A. Bundy. Middle-out reasoning for synthesis and induction. *Journal of Automated Reasoning*, 16(1-2):113–145, 1996.

[52] D. Lacey, J. Richardson, and A. Smaill. Logic program synthesis in a higher-order setting. In *CL '00: Proceedings of the First International Conference on Computational Logic*, pages 87–100, London, UK, 2000. Springer Verlag.

[53] D.B. Lenat. AM: Discovery in mathematics as heuristic search. In D. Heiberg and D.A. Damstra, editors, *Knoledge-based systems in Artificial Intelligence*, chapter 1. McGraw-Hill, 1982.

[54] H. Lowe and D. Duncan. XBarnacle: Making theorem provers more accessible. In *Proceedings of CADE'97*, pages 404–407, 1997.

[55] E. Maclean. Generalisation as a critic. Master's thesis, School of Informatics, University of Edinburgh, 1999.

[56] E. Maclean and J. Fleuriot. Proof-planning non-standard analysis. In *7th International Symposium on AI and Mathematics*, 2002.

[57] E. Maclean, A. Ireland, R. Atkey, and L. Dixon. Refinement and term synthesis in loop invariant generation. In *WING 09: Workshop on Invariant Generation*, 2009.

[58] R. McCasland and A. Bundy. MATHsAiD: a mathematical theorem discovery tool. In *Proceedings of the 8th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pages 17–22. IEEE Computer Society Press, 2006.

[59] R. McCasland, A. Bundy, and S. Autexier. Automated discovery of inductive theorems. *Special Issue of Studies in Logic, Grammar and Rhetoric on Computer Reconstruction of the Body of Mathematics: From Insight to Proof: Festschrift in Honor of A. Trybulec*, 10(23):135–149, 2007.

[60] A. Meier. *Proof-Planning with Multiple Strategies*. PhD thesis, Technischen Fakultät, Universität des Saarlandes, 2004.

[61] E. Melis and A. Meier. Proof planning with multiple strategies. In *Proceedings of the First International Conference on Computational Logic*, volume 1861 of *Lecture Notes in Computer Science*, pages 644 – 659. Springer Verlag, 2000.

[62] E. Melis and J. Siekmann. Knowledge-based proof planning. *Artificial Intelligence*, 115(1):65–105, 1999.

[63] R. Monroy-Borja. The use of abduction to correct faulty conjectures. Master's thesis, University of Edinburgh, 1993.

[64] O. Montano Rivas. Thesis proposal: Scheme-based theorem discovery and concept invention. University of Edinburgh.
homepages.inf.ed.ac.uk/s0793667/publications.html.

[65] T. Nipkow, L.C. Paulson, and M. Wenzel. *Isabelle/HOL - A proof assistant for higher-order logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[66] L. C. Paulson. Isabelle: The next seven hundred theorem provers. In *9th International Conference on Automated Deduction*, volume 310 of *Lecture Notes in Computer Science*, pages 772–773. Springer-Verlag, 1988.

[67] C. Prehofer. Higher-order narrowing. In *Proceedings of the Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 507–516. IEEE Computer Society Press, 1994.

[68] Y. Puzis, Y. Gao, and G. Sutcliffe. Automated generation of interesting theorems. In *Proceedings of the 19th International FLAIRS Conference*, pages 49–54, 2006.

[69] J. Richardson, A. Smaill, and I. Green. System description: Proof planning in higher-order logic with Lambda-Clam. In *15th International Conference on Automated Deduction*, volume 1421 of *Lecture Notes in Artificial Intelligence*, pages 129–133, 1998.

[70] A. Smaill and I. Green. Higher-order annotated terms for proof search. In *TPHOLs '96: Proceedings of the 9th International Conference on Theorem Proving in Higher Order Logics*, pages 399–413, London, UK, 1996. Springer-Verlag.

[71] G. Sutcliffe and C.B. Suttner. The TPTP Problem Library: CNF Release v1.2.1. *Journal of Automated Reasoning*, 21(2):177–203, 1998.

[72] T. Walsh. A divergence critic for inductive proof. *Journal of Artificial Intelligence Research*, 4:209–235, 1996.

[73] T. Walsh, A. Nunes, and A. Bundy. The use of proof plans to sum series. In *11th Conference on Automated Deduction*, number 607 in Lecture Notes in Computer Science, pages 325–339. Springer Verlag, 1992.

[74] S. Walther. About VeriFun. In *Conference on Automated Deduction – CADE-19*, volume 2741 of *Lecture Notes in Computer Science*, pages 322–327. Springer-Verlag, 2003.

[75] M. Wenzel. Isar - a generic interpretative approach to readable formal proof documents. In *Proceedings of TPHOLs'99*, volume 1690 of *Lecture Notes in Computer Science*, pages 167–184. Springer Verlag, 1999.

[76] S. Wilson, J. Fleuriot, and A. Smaill. Automation for dependently typed functional programming. *To appear in: Special Issue on Dependently Typed Programming of Fundamental Informaticae*, 2009.

[77] S. Wilson, J. Fleuriot, and A. Smaill. Supporting dependently typed functional programming with testing and user-assisted proof automation, 2009. Submitted to The Third International Conference on Tests and Proofs.