Extending the proof methods and critics of a proof planner

Daniel Raggi

Master of Science
Cognitive Science
School of Informatics
University of Edinburgh
2011

Abstract

We analysed the failed proof attempts of the proof planner IsaPlanner, searching for patterns that would help us design methods and critics or extend those already implemented in IsaPlanner. The analysis led to a classification, of which two classes had already been discovered and discussed before. A broad novel class was found, in which multiple applications of the inductive hypothesis were required to complete the proofs. The fitted extension to IsaPlanner's methods was designed and implemented successfully. Our results are presented and discussed.

Acknowledgements

Many thanks to Alan and Gudmund for their guidance.
To my friends for the cheer, the stimulating and challenging conversations, and the food.
To those who stayed working beside me until sunrise.
To Ž for her support and inspiration.
To everyone I had already thanked before.
To the rest that deserve it.

I would also like to thank the Mexican National Council on Science and Technology (CONACYT) for funding my MSc. studies [309257].

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.

(*Daniel Raggi*)

Table of Contents

Chapter 1

Introduction

TheoryMine is a company that sells automatically generated mathematical theorems. It is built with technology developed mostly by the mathematical reasoning group at the University of Edinburgh. Its work is divided in three parts. First, it creates original formal inductive theories. Then it generates interesting conjectures for each theory and passes them through a filter that finds counterexamples and therefore refutes some of them. Finally, for each of the remaining conjectures, it guides a search for its proof. In the end some of those conjectures are left unproved. Conjectures that fail to be proven or falsified will be called *open conjectures*.

It has been observed that most of the open conjectures left by TheoryMine are actually true.[1] The goal of this project has been to identify patterns of proof for these conjectures and extend the techniques required for successfully proving them. Our methodology was divided into three major parts: analysis, design and implementation. The analysis consisted of examining the open conjectures and their failed proof attempts with the intention of classifying them according to the patterns of the proofs required. This led to the design of an extension of the techniques implemented in TheoryMine's prover, IsaPlanner. Then, taking time restrictions into consideration, some of the resulting designs would be implemented into IsaPlanner and evaluated according to success in proving open conjectures.

The basic assumptions on which this project stands are the following:

- There are some general reasons why IsaPlanner's is failing to prove the open conjectures. These reasons correspond to its proof techniques and the way they are applied in a proof attempt.

[1] As a matter of fact, all of the open conjectures that were examined for this project were true.

- Analysing the proof attempts can reveal these reasons and help us classify the open conjectures accordingly.
- Reasoning techniques can be designed based on some of the classes, that would help to prove the conjectures.
- IsaPlanner is a good framework in which to implement some of the designed techniques.
- There can be a significant decrease in the number of open conjectures if the techniques are implemented.

The work done for this project showed that the assumptions were correct. A particular extension of a method, which we will be referring to as *multiple fertilisation*, was designed as a solution for one class of failure and implemented successfully in IsaPlanner. As a result, the number of open conjectures went down considerably. Other causes of failure were spotted and analysed. Some of them point to proof patterns and techniques that have been discussed before Ireland and Bundy (1996); Johansson (2009). However, implementing them was not feasible in the timeline of this project. The class of conjectures for which the multiple fertilisation method was created was proved almost entirely by our extension, and success spreads even to other classes.

Chapter 2

Background

2.1 Automated Theorem Proving

Since the very beginning of computer science the automation of reasoning has been one of the main challenges. Despite extensive and exhaustive work in this field, it still has a long path to go if we want it to achieve the standards of human reasoning. Most of this work has been directed towards the automation of mathematical reasoning, or *automated theorem proving*. This is due to the formal qualities of mathematics and the very meticulous studies of mathematical logic that had already been done when computer science was created (from which computer science itself sprouted!).

Realising the formal characteristics of a mathematical proof involved careful examination of the process of reasoning. The most obvious aspect of this process is the characteristic step-by-step way in which valid logical arguments are constructed. This aspect can be formalised into first-order logic by the famous sequent calculus, which, in practical terms, involves unfolding a formal statement into a set of statements from which it can be inferred. For example, the goal $\Gamma \vdash \alpha[x]$ can be unfolded into $\Gamma \vdash \beta$ and $\Gamma, \beta \vdash \alpha[x]$, or into $\Gamma \vdash \alpha[t]$ and $\Gamma \vdash x = t$. As rules of a sequent calculus, they could be written respectively as follows:

$$\frac{\Gamma \vdash \beta \quad \Gamma, \beta \vdash \alpha[x]}{\Gamma \vdash \alpha[x]} \qquad \frac{\Gamma \vdash \alpha[t] \quad \Gamma \vdash x = t}{\Gamma \vdash \alpha[x]} \tag{2.1}$$

The sequent calculus, however, sheds no light on which is the actual sequence of steps required for a proof for, as it is evident, there is an infinite number of possible ways to unfold a statement (for example, in the first rule of (2.1), β could stand for any first order formula and the

rule remains logically valid). This problem can be attacked locally by a heuristic-driven search or globally by a *proof plan* Bundy et al. (1991). For example, good heuristics could find a way to simplify a term x into a provably equal term t, such that the proof for $\alpha[t]$ is simpler than that required by $\alpha[x]$. A proof plan could devise the general structure of the required proof using the goals, a higher level understanding of the techniques at hand and the proof attempt history.

A proof of statement α can be seen as a tree in which the root is α, the leaves are axioms or assumptions, and the edges represent applications of Gentzen rules. In this paradigm, a proof plan can be seen as a partial tree of this sort. That is, a tree with gaps to be filled by connecting Gentzen rules.

2.2 Induction

Induction is an essential part in theories regarding numbers, lists and recursively-built sets. For each of such sets, induction has to be custom-made and a well built induction rule expresses the structure of its set. The construction of such an induction rule has to be intimately related to the way the set is built. For example, consider the set of types T built by the following recursive rule:

- 0 and $0'$ are in T.
- If τ is in T then $s\tau$ is also in T.
- If τ is in T then $r\tau$ is also in T.

Assuming the injectivity of s and r and the fact that 0 and $0'$ are different, T can be said to represent a couple of binary trees. Based on such construction of the trees, we can build the inductive rule as follows:

$$\frac{\Gamma \vdash A[0] \quad \Gamma \vdash A[0'] \quad \forall\tau.\Gamma, A[\tau] \vdash A[s\tau] \quad \forall\tau.\Gamma, A[\tau] \vdash A[r\tau]}{\Gamma \vdash \forall\tau.A[\tau]} \tag{2.2}$$

where A stands for any formula of the theory Γ, whose universe is represented by T.

The parallelism between such inductive rule and the recursive construction of the set is evident. It points at the strategy required to derive a useful *scheme* of induction for a set, given its construction. This is a very useful heuristic for unfolding a conjecture into subgoals. However, inductive proofs are usually quite complex and require a more global control over the proof. For

example, they may require generalising the inductive hypothesis and this might be hinted by a failure to prove with the previous non-generalised hypothesis (which is higher in the proof tree, and thus requires global control and is not accounted by a step-by-step reasoning strategy). Nevertheless, there is structure shared by a lot of inductive proofs, and it is from this fact that proof planning holds. An inductive proof would typically look like this:

1. The goal is split into *base cases* and *step cases*.
2. The base cases are proved either by simplification or by induction on another variable of the formula.
3. The step case formulas are heuristically driven step by step to resemble the inductive hypothesis by a process called *rippling*.
4. Due to the resemblance the inductive hypothesis might be applied to the current goal. This step is called *fertilisation*. Failure to fertilise might be examined and hint at a change in the inductive hypothesis.
5. The result might be solved by simplification or induction, or require the calculation of a hidden lemma.

This structure is the base for the architecture of the proof strategies implemented in IsaPlanner used by TheoryMine.

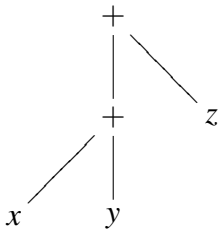
2.3 Rippling

Rippling is a pattern that many inductive proofs follow, but also the formal technique used for such proofs Bundy et al. (1993). Let us assume that we are trying to prove an equality $\sigma(x) = \tau(x)$ by induction over x (say, a natural number). For the step case we will want to prove the equality $\sigma(Suc(x)) = \tau(Suc(x))$. Often, a proof will exploit the similarity between the goal and the assumption. Ideally, we might want to de-construct the goal into a subgoal $\Xi(\sigma(x)) = \Xi(\tau(x))$ and thus apply the assumption. Something like this can often be done because of the nature of recursive definitions. For example, if we know a recursive definition for σ it means that $\sigma(Suc(x))$ could be seen in terms of applications of σ to x, thus getting us to an expression in which the assumption can be applied. In cases like these, the way $\sigma(x)$ is 'cleared' can be seen as waves

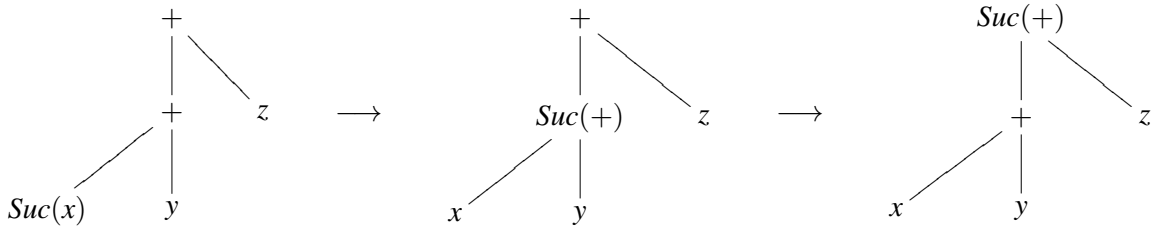
rippling outwards. This is what I will call Rippling Bundy et al. (1993). Rippling can also happen inwards and it is an analogous process, but for the sake of clarity and simplicity I'll only explain outward rippling. The rippling pattern is the inspiration for a heuristic of step by step reasoning. It is this heuristic that should guide the rewrite process of a term like $\sigma(Suc(x))$ into a term like $\Xi(\sigma(x))$.

Rippling involves finding a term from the assumption scattered within the goal and clearing it from a bunch of 'garbage' until there is an instance of the assumption term within the remaining goal. The term from the assumption is called the *skeleton*, the garbage is called the *wave front* and the rules for clearing on each step are called *wave rules* Bundy et al. (1993). If we think of the terms as trees, we can see the steps of Rippling as pushing the wave fronts to nodes closer to the roots.

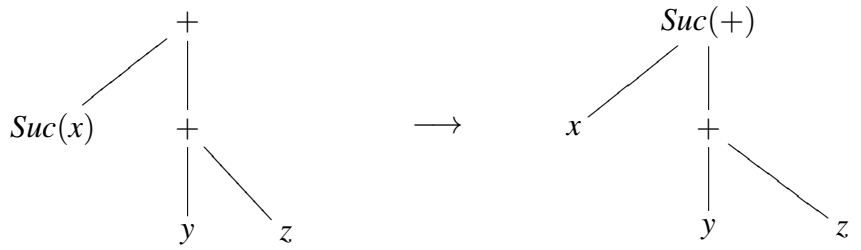
Consider, for example, the proof of the associativity of $+$. For the step case we want to prove $(Suc(x)+y)+z = Suc(x)+(y+z)$ from $(x+y)+z = x+(y+z)$. In tree form, the left-hand side of the assumption (the skeleton) looks like this:



We can match the skeleton to the goal and see the goal in the structure of the skeleton. This will be useful to identify the wave front. Then we can apply the wave rules (in this case the only wave rule is the recursive definition of $+$) and Ripple until the tree is clear. The process looks as follows:



For the right-hand side of the equation the procedure is done as follows:



As in this case, when it is no longer possible to push the wave front out of the skeleton, rippling is said to be *blocked*. If at one stage rippling is blocked and the skeleton has been cleared successfully, the goal can be fertilised and thus the problem solved. Sometimes the goal is an instance of the assumption, in which case fertilisation is said to be *strong*. In other cases strong fertilisation might not be possible. However, there might be an instance of the skeleton within the goal and, thus, the assumption may be applied. This is called a *weak* fertilisation. The result after such a fertilisation might then proceed to be proved by other techniques (this might be as easy as applying a logical rule like the reflexivity of $=$ in the case mentioned above).

2.4 Lemma calculation critic

The so called *cut-rule* of inference 2.3 formalises the use of an intermediate lemma β for the proof of α.

$$\frac{\Gamma \vdash \beta \quad \Gamma, \beta \vdash \alpha}{\Gamma \vdash \alpha} \tag{2.3}$$

However, the independence of β from α gives no clue as to where it might come from; every instance of the rule is logically valid. This creates the technical problem of infinite branching for the unguided application of inference rules in automatic search for proof. It has been noticed that, often, an analysis of the failed proof attempt and the reasoning methods at hand can give clues on how to build a useful lemma. Some techniques have been designed from that principle Ireland (1992); Ireland and Bundy (1996); Johansson et al. (2010). These techniques are called *critics*. I will discuss the *lemma calculation* critic in this section.

The lemma calculation critic works because of the curious fact that sometimes it is easier to prove a more general version of a theorem than the theorem itself. When the prover is trying to prove an equation and it gets stuck, the critic searches for a term, common in both sides of the equation, that contains garbage that might be an obstacle for the methods to work and achieve a proof. Then it proceeds to calculate a lemma that generalises such terms to a variable. After doing this generalisation, the prover might get unstuck and find a proof for the general version.

2.5 IsaPlanner

IsaPlanner was developed by Lucas Dixon as his PhD thesis Dixon (2006). It is a framework for proof planning written in Standard ML and it works over Isabelle's logics and proof tactics Paulson (1994).

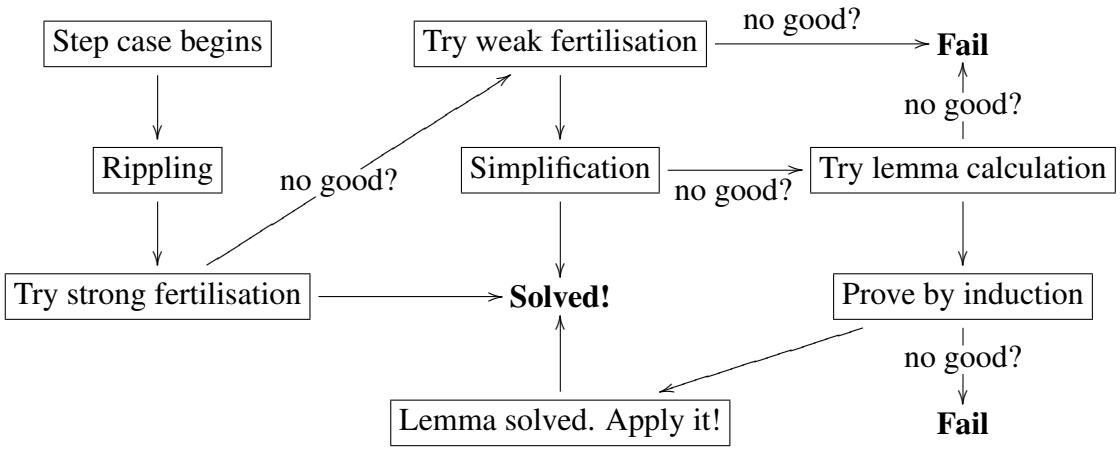
The central object for IsaPlanner's inner workings is what is called a *reasoning state*. A reasoning state contains a proof plan, a reasoning technique to be applied to the proof plan, and a context; which stores the information that may be used in the proof (for example, a set of wave rules). It can be understood to represent a moment in time, containing all the information available about the reasoning process. Its reasoning technique is a function such that given the current reasoning state it gives back a sequence of reasoning states, which in turn represent the possible 'future moments in time', one for each different way in which the technique could be applied. Each application of a reasoning technique would change the reasoning state's proof plan accordingly. For example, a technique might fertilise all the open goals that can be fertilised, closing the space in the gaps of the proof plan tree.

Stepping back we can see that an OR-tree is formed, such that the nodes are reasoning states and the edges are the applications of the reasoning techniques. Thus, a proof would be a directed path such that it ends in a reasoning state such that its proof plan has no gaps (where a gap is a subgoal that needs to be proven).

For inductive proofs, such as the ones studied in this project, IsaPlanner follows very closely the structure that was described before. Induction is divided into base cases and step cases, all of which have to be proved to finish the proof.

For the base cases, Isabelle's Simplification tactic is applied to all the base goals. If this fails, it proceeds to apply the lemma calculation critic, and to prove the resulting lemma by induction.

For each of the step cases, the structure of the proof procedure looks as follows:



2.6 TheoryMine's elements

TheoryMine is intended to create novel interesting theorems. It brings together four systems, each built on top of the last: Isabelle, IsaPlanner, IsaCoSy, IsaWannaThm. As a whole, this synthesis enriches the testing ground for IsaPlanner by giving it material to work with.

This project is centred on IsaPlanner. However, there are some basic assumptions about its work that rely on the other three systems that form TheoryMine. One can find detailed descriptions of each of them in Paulson (1994); Bundy et al. (2010); Dixon (2006); Johansson et al. (2009); Cavallo (2009). Here I give an overview of them, focusing on what is relevant to the project.

- **Isabelle**. This is the general frame over which IsaPlanner is built. It is generally used for interactive proofs. It can work in several logics (HOL, ZF, CTT) and has libraries of types and theorems, as well as automatic reasoning methods (like simplification) and a tool for deriving induction schemes for types. IsaPlanner makes extensive use of these methods Paulson (1994).

- **IsaCoSy**. This system, developed by Moa Johansson, generates conjectures when given an inductive theory. Its process relies on generating a lot of formulas and filtering out most of the false ones by running them through a counter-example finder. IsaCoSy generates its conjectures in irreducible form by not using terms which can be rewritten into other terms, thus getting rid of a lot of redundancy and therefore making conjectures more interesting Johansson et al. (2009).

- **IsaWannaThm**. This system is the program at the surface of TheoryMine. It was developed by Flaminia Cavallo as a final-year undergraduate project Cavallo (2009). It generates simple recursive types using Isabelle datatypes and constructs recursive functions over them using IsaCoSy. For the sake of novelty it filters out types which are already in Isabelle's libraries. A type set with its functions can be seen as an universe on which a theory can be created. Such theories are created by IsaWannaThm by using the definitions of the functions, logical axioms and inductive schemes, all within the framework of Isabelle. IsaWannaThm feeds such theories to IsaCoSy to get the conjectures which will be then passed to IsaPlanner for it to prove.

Chapter 3

Theories and conjectures

The theories of TheoryMine are generated as follows:

1. A type set is created in a recursive fashion, like the datatype of natural numbers, which is built out of 0 and the *constructor function Suc* (the symbol for the successor function). The types TheoryMine uses are built by taking naturals and booleans as base types and building more complex types from them. For example, by using $bool \times \mathbb{N}$ as a base and applying the constructor function C recursively. Then, it filters out known types, for the sake of originality. A type set can be seen as an universe for a theory. Like the natural numbers, each element of the type set can be thought to represent an element of the universe (and the injectivity of C is the assumption that makes each element unique). The generative structure of this universe is reflected in the theory by the axioms of induction, which are created automatically out of the recursive definition of the type.

2. For each type T, a set of functions f_i is defined in a recursive manner over the recursive definition of the type. For example, such functions might take as arguments an element of the type and a natural number, and compute a natural number from them. In the theory, the definition of the functions is reflected as axioms (rewrite rules).

Although five different theories were analysed for this project (T_1, T_2, T_3, T_6, T_{12}), all five can be seen as belonging to one of two classes. This is because even though the types over which they are built are unique, a lot of information is redundant because the functions that constitute the theory tend to use a very small fraction of the information of the types.

3.1 Funny lists of booleans (but, in fact, just a set of funny natural numbers)

Belonging to this class are theories T_2, T_6 and T_{12}. Let us use T_{12} to exemplify. Its criteria for belonging to the type set is, for $C_{23} : \mathbb{N} \to T_{12}$ and $C_{24} : T_{12} \to (bool \to T_{12})$ constructor functions:

- **base:** $C_{24}\, n$ for n any natural number
- **step:** $C_{23}\, \tau\, b$ for b boolean and τ in T_{12}

Thus, elements of the type set are of the form $C_{23} \ldots C_{23} C_{24}\, n\, b_1 \ldots b_k$ with $k \geq 0$, k C_{23}'s, n a natural and all b_i booleans. This makes them, basically, lists of booleans with a natural in the middle. However, all the functions over them are defined in a way such that the only information that matters is k; never using the booleans or the natural. For example, the function $f_0 : T_{12} \to (\mathbb{N} \to \mathbb{N})$ is defined as follows:

- **base:** $f_0(C_{24}\, a)\, b = b$
- **step:** $f_0(C_{23}\, a\, b)\, c = Suc(f_0\, a(Suc\,(f_0\, a\, c)))$

Notice that in the base case the only information comes from outside the type (the second argument), and in the step case one is only removing the outer layers while incorporating information from outside the type. Thus, this type might be partitioned into classes, one for each natural k, defined as follows:

$$\mathcal{C}_k = \{(C_{23} \ldots C_{23} C_{24}\, n\, b_1 \ldots b_k) \in T_{12} \,\textbf{:}\, n \text{ natural}, b_i \text{ booleans}\}$$

It is clear that, given the nature of the functions defined for this theory, every statement will be true for one element of the type if and only if it is true for all members of the class it belongs to. Thus, for the purpose of the analysis we can think of the type's quotient over the classes, which is only a copy of natural numbers, instead of thinking about the type itself.

Interpreting the type as natural numbers, a careful analysis of f_0 will show that for $a > 0$:

$$f_0\, a\, b = b + \sum_{i=1}^{a} 2^i = b + 2(2^a - 1)$$

Let us analyse what happens when IsaPlanner tries to prove a conjecture of this theory, specifically related to this function.

Consider the conjecture $Suc(Suc(f_0\,a\,b)) = f_0\,a(Suc(Suc\,b))$. From the analysis of the function above, the result is trivial because

$$f_0\,a(Suc(Suc\,b)) = Suc(Suc\,b) + 2(2^a - 1) = Suc(Suc(b + 2(2^a - 1))) = Suc(Suc(f_0\,a\,b)))$$

. This way we know that at least it is true (however, the actual proof that the function is the one I claim to be is much harder than the proof of this conjecture) and we can proceed to analyse the proof attempt taking that fact into consideration. Its proof attempt is reflected in IsaPlanner's proof plan, where it got stuck, shown in figure 1.

Even though IsaPlanner got stuck, we can see that goal (o) could still be fertilised by 'pushing' two of the the *Suc*'s of the left-hand side inwards into term d, proving the conjecture.

3.2 Funny lists of naturals (but, in fact, just a funny version of $\mathbb{N} \times \mathbb{N}$)

Belonging to this class are theories T_1 and T_3. Let us use T_1 to exemplify. The criteria for belonging to its type set is, for $C_1 : \mathbb{N} \to (\mathbb{N} \to T_1)$ and $C_2 : T_1 \to (\mathbb{N} \to T_1)$ constructor functions, as follows:

- **base:** $C_1\,n\,m$ for n and m any natural numbers.
- **step:** $C_2\,\tau\,n$ for τ in T_1 and n any natural number.

Thus, elements of the type set are of the form $C_2 \ldots C_2\,C_1\,m_1\,m_2\,n_1 \ldots n_k$ with $k \geq 0$, k C_2's, and all m_i's and n_i's natural numbers. Given that the amount of C_2's is the same as that of natural numbers $(+2)$, that information is redundant. Moreover, none of the functions defined for this type uses a number apart from one of them in the base (m_2 in the expression above). Thus, this type set can also be partitioned in classes, one for each element (m,k) of $\mathbb{N} \times \mathbb{N}$:

$$C_{m,k} = \{(C_2 \ldots C_2\,C_1\,m_1\,m\,n_1 \ldots n_k) \in T_1 \textbf{ : } m_1 \text{ and } n_i \text{ naturals}\}$$

in which any statement is true for a member of the type set if and only if it is true for every member of its class. Thus, for the sake of the analysis, the type set can be seen as $\mathbb{N} \times \mathbb{N}$.

Consider the function $f_6 : T_1 \to (\mathbb{N} \to \mathbb{N})$ defined as follows:

- **base:** $f_6(C_1\,a\,b)\,c = Suc(Suc(Suc(Suc(Suc(Suc\,c)))))$

{{ALL a : T_12, b : nat.
(g1): "$Suc(Suc(f\_0\,a\,b)) = f\_0\,a(Suc(Suc\,b))$"
[by_meth (Induction on: a) to: i, j]

{ALL nat : nat, c : nat.
(i): "$Suc(Suc(f\_0(C\_24\,nat)\,c)) = f\_0(C\_24\,nat)(Suc(Suc\,c))$"
[by_meth (simp (no_asm_simp))]}

{ALL c : T_12, bool : bool, d : nat.

{ALL e. (k): "$Suc(Suc(f\_0\,c\,e)) = f\_0\,c(Suc(Suc\,e))$"}
⊢

(j): "$Suc(Suc(f\_0(C\_23\,c\,bool)\,d)) = f\_0(C\_23\,c\,bool)(Suc(Suc\,d))$"
[by_meth (subst_w_thm: T_12.f_0.simps_2) to: l]

(l): "$Suc(Suc(f\_0(C\_23\,c\,bool)\,d)) = Suc(f\_0\,c(Suc(f\_0\,c(Suc(Suc\,d)))))$"
[by_meth (subst_w_thm: T_12.f_0.simps_2) to: m]

(m): "$Suc(Suc(Suc(f\_0\,c(Suc(f\_0\,c\,d))))) = Suc(f\_0\,c(Suc(f\_0\,c(Suc(Suc\,d)))))$"
[by_meth (subst_w_thm: Nat.nat.inject) to: n]

(n): "$Suc(Suc(f\_0\,c(Suc(f\_0\,c\,d)))) = f\_0\,c(Suc(f\_0\,c(Suc(Suc\,d))))$"
[by_meth (subst k) to: o]

(o): "$f\_0\,c(Suc(Suc(Suc(f\_0\,c\,d)))) = f\_0c(Suc(f\_0\,c(Suc(Suc\,d))))$"
[? gap]}}}

Figure 1 (proof plan): *Indentation represents the branching in the proof tree. For example, the goal is in the highest level and the base case (goal (i)) and the step case (goals (j) - (o)) are one level below. Each universal quantifier is enclosed within curly brackets to show their reach. Each step explains within square brackets the technique used to unfold the goal, and the subgoals into which it unfolds.*

In the first step, for goal (g1), the note [by_meth (Induction on: a) to: i, j] *shows that the technique is induction over a, and it branches into goal (i) and (j). In goals (j) - (m) the technique is simple rewriting with their corresponding rules (guided by rippling), and each goal leads to the next. In goal (n), the technique is a substitution by assumption (k); that is, fertilisation. In goal (i) there are no branching pointers, which means that the goal was solved (by Isabelle's simplification tactic; in this case it is just applying the definition of f_0, plus the reflexivity of equality). In goal (o) there is no branching but there is a gap, which means that the goal wasn't solved but, rather, the prover got stuck and can't unfold the goals any more. A proof attempt has gaps if and only if it is incomplete.*

- **step:** $f_6(C_2\,a\,b)\,c = Suc(Suc(f_6\,a(f_6\,a\,c)))$

A careful analysis of such function reveals that it works as follows: if $l > 0$ is the number of C_2's,

$$f_6(C_2\,a\,b)\,c = c + \sum_{i=1}^{l} 2^i + 6(2^l) = c + \sum_{i=1}^{l} 2^i + 2(2^l) + 4(2^l) = c + \sum_{i=1}^{l+2} 2^i = c + 2(2^{l+2} - 1)$$

Thus it can be seen that this particular function behaves as a function of just one of the two natural numbers the type contains (the one formed by C_2's). However, other functions of the same theory make use of the other natural number. This function was picked for its interesting proof attempt, shown in figure 2.

In the same way as the previous example, IsaPlanner tries to prove by induction over a. However, it gets stuck at a point in which it could fertilise further and prove the conjecture.

Both examples shown reveal a pattern that is to be seen all over IsaPlanner's failed proof attempts. This will be discussed further in the next section.

{{ALL a : T_ 1, b : nat.
(g1): "$Suc(f\_6\,a\,b) = f\_6\,a(Suc\,b)$"
[by_ meth (Induction on: a) to: i, k]

{ALL c : T_ 1, nat : nat, d : nat.

{ALL e. (j): "$Suc(f\_6\,c\,e) = f\_6\,c(Suc\,e)$"}
⊢

(i): "$Suc(f\_6(C_2\,c\,nat)\,d) = f\_6(C_2\,c\,nat)(Suc\,d)$"
[by_ meth (subst_ w_ thm: T_ 1.f_ 6.simps_ 2) to: l]

(l): "$Suc(f\_6(C_2\,c\,nat)\,d) = Suc(Suc(f\_6\,c(f\_6\,c(Suc\,d))))$"
[by_ meth (subst_ w_ thm: T_ 1.f_ 6.simps_ 2) to: m]

(m): "$Suc(Suc(Suc(f\_6\,c(f\_6\,c\,d)))) = Suc(Suc(f\_6\,c(f\_6\,c(Suc\,d))))$"
[by_ meth (subst_ w_ thm: Nat.nat.inject) to: n]

(n): "$Suc(Suc(f\_6\,c(f\_6\,c\,d))) = Suc(f\_6\,c(f\_6\,c(Suc\,d)))$"
[by_ meth (subst_ w_ thm: Nat.nat.inject) to: o]

(o): "$Suc(f\_6\,c(f\_6\,c\,d)) = f\_6\,c(f\_6\,c(Suc\,d))$"
[by_ meth (subst j) to: p]

(p): "$f\_6\,c(Suc(f\_6\,c\,d)) = f\_6\,c(f\_6\,c(Suc\,d))$"
[? gap]}

{ALL nat1 : nat, nat2 : nat, c : nat.
(k): "$Suc(f\_6(C_1\,nat1\,nat2)\,c) = f\_6(C_1\,nat1\,nat2)(Suc\,c)$"
[by_ meth (simp (no_ asm_ simp))]}}}

Figure 2 (proof plan): *In the first step, goal g1 is unfolded into j and k by induction over a. In goals i - n the technique is simple rewriting with their corresponding rules (guided by rippling), and each goal leads to the next. In goal o, the technique is a substitution by assumption j; that is, fertilisation. In goal k there are no branching pointers, which means that the goal was solved (by Isabelle's simplification tactic; in this case it is just applying the definition of f_0 plus the reflexivity of equality). In goal p there is no branching but there is a gap, which means that the goal wasn't solved but, rather, the prover got stuck and can't unfold the goals any more.*

Chapter 4

Analysis and classification of failed proofs

For each of the five theories mentioned, a list of open conjectures and proved theorems was analysed. In this section we explain what the analysis shows and how it hints at the design for the reasoning techniques required.

There is a large class of conjectures, central to this project, which require multiple fertilisations, described in section 4.3. However, other distinct causes of failure were recognised, two of which have already been discussed before Ireland and Bundy (1996); Johansson et al. (2010). Those failures can be thought of in terms of the techniques required to actually prove the conjectures. However, in its current version, IsaPlanner is lacking the means to apply such techniques.

The other causes of failure are not as easily understandable in terms of the techniques required to fix them. Those other classes will be just briefly mentioned.

4.1 Generalisation critic

Like the lemma calculation critic, the competence of the generalisation critic relies on the fact that it is often easier to prove a more general version of a theorem than the theorem itself. The lemma calculation critic takes the goal at a point the prover is stuck and generalises it. However, in inductive proofs it often happens that an analysis at the point of failure hints at a change in a previous step; specifically, in the inductive hypothesis. The analysis may reveal that the goal would be available for fertilisation if only the inductive hypothesis had been more general from the start. This apparently paradoxical fact arises because the same complexity that makes a more

general theorem seem harder to prove gives the prover more power using the inductive hypothesis as an assumption. Thus, a special critic that generalises the inductive assumption instead of the goal, is necessary in such cases.

Suppose that we are trying to prove the goal $\forall A.\, f(A,0) = f(0,A)$ by induction over A, with f a function such that, after rippling, we get subgoal (4.1)

$$f(A,g(0)) = f(g(0),A) \tag{4.1}$$

In that case, fertilisation cannot happen. However, if one tried to prove the more general version (4.2), one is likely to get (4.3) as the subgoal, and then the hypothesis of induction can be applied by instantiating the variable B as $g(Q)$.

$$\forall A\,B.\, f(A,B) = f(B,A) \tag{4.2}$$

$$\forall Q.\, f(A,g(Q)) = f(g(Q),A) \tag{4.3}$$

To exemplify, let us consider theory T_{12} with function $f_2 : T_{12} \to (\mathbb{N} \to \mathbb{N})$ defined as follows:

- **base:** $f_2(C_{24}\,a)\,b = Suc(Suc\,b)$
- **step:** $f_2(C_{23}\,a\,b)\,c = f_2\,a(Suc(Suc\,c))$

Consider the conjecture $f_2\,a(Suc\,0) = Suc(f_2\,a\,0)$. IsaPlanner's proof plan at the point it gets stuck is shown in 3.

As the example shows, one can recognise the need for a generalisation from the proof failure. The critic to do so has been implemented in the proof planner Clam 3 Ireland and Bundy (1996), but it hasn't been done IsaPlanner. Roughly, an implementation of such technique has to follow the next guideline:

1. When rippling is blocked but one cannot fertilise, search for a term in the inductive assumption that doesn't contain the inductive variable or doesn't contain a subterm into which the wave front will be thrown.

2. Check that, if the term is replaced with a new variable, the resulting formula can fertilise the rippled goal.

3. Change the inductive hypothesis in the proof plan.

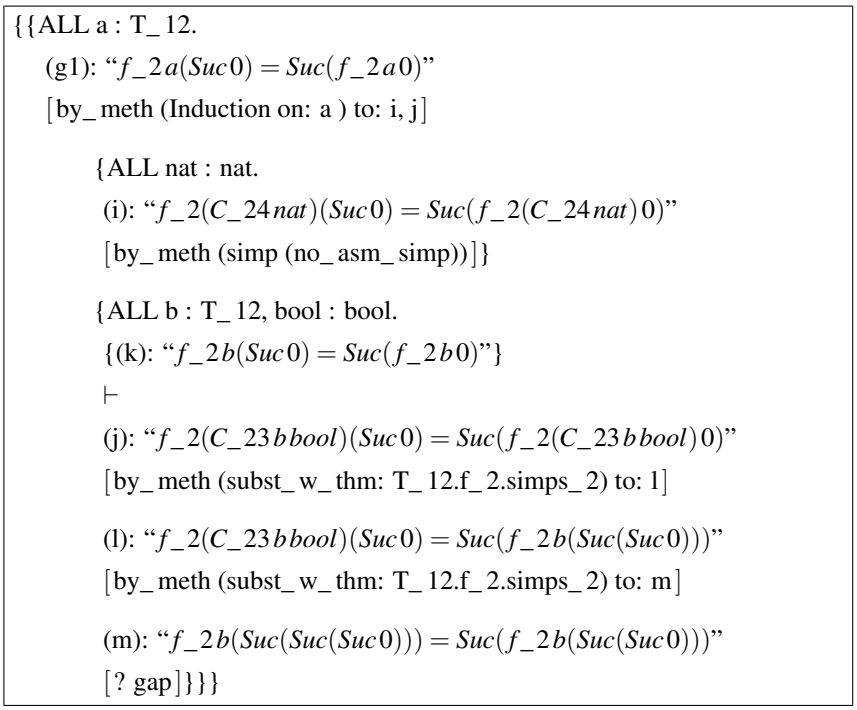


Figure 3 (proof plan): *As it can be seen, the inductive assumption (k) doesn't fit the goal (m) and thus fertilisation cannot happen. At this stage IsaPlanner gets stuck and the proof attempt fails.*

The reason why this would work is that the first condition ensures that changing the term will not tamper with the way in which rippling works. After rippling, the resulting goal will have the same structure as the goal which the second condition ensured fertilisation could occur. Therefore, fertilisation will be possible in the new proof attempt.

About one fourth of the conjectures of TheoryMine that were analysed would benefit from the application of this technique. However, it also happens that for all the open conjectures in this class, their general version (the one that the generalisation critic would have to discover) was also in the list (and in a few cases it was even in the list of proved theorems).

4.2 Lemma speculation critic

In the search for the useful middle lemma in a proof, the lemma calculation critic tries to find and prove a generalisation of the goal in which the prover got stuck, and the generalisation critic tries to prove a more general version of the inductive hypothesis. Both do this by analysing a part of the proof attempt at the point the prover got stuck. Similar to lemma calculation, the lemma speculation critic Johansson (2009) tries to find a way to prove the goal in which the prover got stuck. However, it doesn't try to generalise it; it tries to find and prove a theorem that can be used as a wave rule for the skeleton to be cleared for fertilisation.

A limited number of examples were found in which one would arrive to a subgoal like (4.5), after rippling, when the inductive assumption is (4.4).

$$\forall BC.\, f(A, f(B,C)) = f(B, f(A,C)) \tag{4.4}$$

$$\forall QP.\, g(f(A, g(f(Q,P)))) = f(Q, g(f(A, g(P)))) \tag{4.5}$$

In this case, although the goal is very similar to the inductive assumption, there are no possible fertilisations and the lemma calculation critic will not work because the only common terms in both sides of the equation are the variables. An actual lemma that would clear the skeleton in goal (4.5) is (4.6).

$$\forall AQ.\, f(A, g(Q)) = g(f(A,Q)) \tag{4.6}$$

When applying (4.6) as a wave rule instantiating Q as $g(f(Q,P))$ on the left side and as P on the right side, we get (4.7) and fertilisation can happen, getting (4.8); a goal for which the lemma

calculation critic will calculate (4.9), which might be more easily solvable.

$$\forall QP.\, g(g(f(A, f(Q,P)))) = f(Q, g(g(f(A,P)))) \tag{4.7}$$

$$\forall QP.\, g(g(f(Q, f(A,P)))) = f(Q, g(g(f(A,P)))) \tag{4.8}$$

$$\forall QPN.\, g(g(f(Q,N))) = f(Q, g(g(N))) \tag{4.9}$$

Consider the conjecture of theory T_{12}, $f_5\, a(f_5\, b\, c) = f_5\, b(f_5 a\, c)$, where f_5 is defined as follows:

- **base:** $f_5(C_2 4\, a)b = Suc(Suc(Suc(Suc(Suc\, b))))$
- **step:** $f_5(C_2 3\, a\, b)\, c = Suc(Suc(f_5\, a(Suc(Suc(Suc\, c)))))$

The proof attempt is shown in figure 4.

{ALL b : T_ 12, bool : bool, e : T_ 12, f : nat.
{(k): "$f\_5\, d(\_\ 5\, g\, h) = f\_5\, g(f\_5\, d\, h)$"}
⊢
(j): "$f\_5(C\_23\, d\, bool)(f\_5\, e\, f) = f\_5\, e(f\_5(C\_23\, d\, bool)\, f)$"
[by_ meth (subst_ w_ thm: T_ 12.f_ 5.simps_ 2) to: l]

(l): "$f\_5(C\_23\, d\, bool)(f\_5\, e\, f) = f\_5\, e(Suc(Suc(f\_5\, d(Suc(Suc(Suc\, f))))))$"
[by_ meth (subst_ w_ thm: T_ 12.f_ 5.simps_ 2) to: s]

(s): "$Suc(Suc(f\_5\, d(Suc(Suc(Suc(f\_5\, e\, f)))))) = f\_5\, e(Suc(Suc(f\_5\, d(Suc(Suc(Suc\, f))))))$"
[? gap]}

Figure 4 (partial proof plan): *In the proof plan to which this piece belongs, the base case is proved successfully but the step case ripples into something that can't be fertilised or rippled any more.*

The goal of the lemma speculation critic is to find a wave rule that clears the skeleton successfully and makes fertilisation possible Johansson (2009); Johansson et al. (2010). For example, in proof attempt 4, the lemma

$$Suc(Suc(Suc(f\_5\, p\, q))) = f\_5\, p(Suc(Suc(Suc\, q)))$$

would rewrite the goal into the fertilisable goal

"$Suc(Suc(f\_5\, d(f\_5\, e(Suc(Suc(Suc\, f))))))) = f\_5\, e(Suc(Suc(f\_5\, d(Suc(Suc(Suc\, f))))))$"

from which the lemma calculation critic would calculate

$$\text{“}Suc(Suc(f\_5\,d\,n)) = f\_5\,e(Suc(Suc\,n))\text{”}$$

, a lemma that can actually be proven (it doesn't appear as an open conjecture nor as a theorem, but our experiments have shown that it can indeed be proved). Another lemma that would solve the problem is “$Suc(f\_5\,p\,q) = f\_5\,p(Suc\,q)$”, which is actually in the list of theorems proved by IsaPlanner (without our extension).

Interestingly, it was observed that, in many cases, a lemma that would solve the problem appears either in the list of theorems or in the list of open conjectures. Thus, if the required lemma were to be solved and used as a wave rule for rippling, the theorem might be proved. This, of course, is particular to the way in which IsaCoSy generates its conjectures and does not shed light on the actual process of lemma speculation as a method or critic for theorem proving.

There is an implementation of the lemma speculation critic in IsaPlanner. However, our experiments with it didn't prove anything (nor did they seem to give any obvious clue at what was happening. Unfortunately, the time constraint of this project didn't let us analyse the reason for these failures.

In any case, our analysis shows that the number of cases that require lemma speculation is very small, compared to those that require multiple fertilisations or generalisation. It should also be noted that some of these cases can actually be solved by adding as wave rules the theorems our multiple fertilisation method proves. This is discussed in detail in chapter 6.

4.3 Multiple fertilisation method

We have shown two examples in which multiple fertilisations are required. In about a third of the open conjectures analysed, the requirement for this technique was spotted. Here I show some generic forms of how this would happen and discuss the technique in more detail.

Suppose that we are trying to prove conjecture $\forall A\,B\,C.\,f(A,f(B,C)) = f(B,f(A,C))$ by induction over A and f is a function such that, after rippling, we get the subgoal (4.10).

$$\forall P\,Q.\,f(A,f(A,f(P,Q))) = f(P,f(A,f(A,Q))) \tag{4.10}$$

If we fertilise instantiating the variable C as Q we will get goal (4.11). At this point a second

fertilisation is required, instantiating C as $f(A,Q)$, getting the equality (4.12).

$$\forall P\,Q.\,f(A,f(P,f(A,Q))) = f(P,f(A,f(A,Q))) \tag{4.11}$$

$$\forall P\,Q.\,f(P,f(A,f(A,Q))) = f(P,f(A,f(A,Q))) \tag{4.12}$$

Thus, the conjecture gets solved.

Let us consider the following slightly different example. Suppose we are trying to prove $\forall A\,B.\,g(f(A,B)) = f(A,g(B))$ by induction on A and it gets rippled to (4.13)

$$\forall P.\,g(f(A,g(f(A,P)))) = f(A,g(f(A,g(P)))) \tag{4.13}$$

As in the first example, we need to 'push' g two times, as shown in (4.14) and (4.15), thus completing the proof.

$$\forall B.\,g(f(A,f(A,g(B)))) = f(A,g(f(A,g(B)))) \tag{4.14}$$

$$\forall B.\,f(A,g(f(A,g(B)))) = f(A,g(f(A,g(B)))) \tag{4.15}$$

The above examples divide proofs by multiple fertilisations into two classes, one in which what needs to be 'pushed' is a variable (as in the first example), and one in which it is a function (as in the second example). In both cases it seems to occur quite a lot with functions like the ones generated by IsaCoSy because of their nested nature. Examples of the first class require even one more nesting, which doesn't occur in the definition of the functions, but happens when a nested definition is combined with a conjecture in which there is already a nested function.

Let us consider the following examples for function $f_4 : T_{12} \to (\mathbb{N} \to \mathbb{N})$ defined as follows:

- **base:** $f_4(C_{24}\,a)\,b = Suc(Suc(Suc(Suc\,b)))$
- **step:** $f_4(C_{23}\,a\,b)\,c = Suc(f_4\,a(f_4\,a\,c))$

The first conjecture to be proved is $f_4\,a(Suc\,b) = Suc(f_4\,a\,b)$. Figure 5 shows the proof plan at the point IsaPlanner gets stuck. It is clear that such example belongs to the second class, in which a function needs to be pushed.

Now, let us consider the proof attempt for conjecture $f_4\,a(f_4\,b\,c) = f_4\,b(f_4\,a\,c)$, shown in figure 6. As it can be seen, this proof attempt belongs to the class in which it is a variable that needs to be pushed.

Example 6 shows clearly the possibility for a rippling technique that fertilises unrestrictedly to fall into an infinite loop. However, it would be inelegant to restrict the number of possible

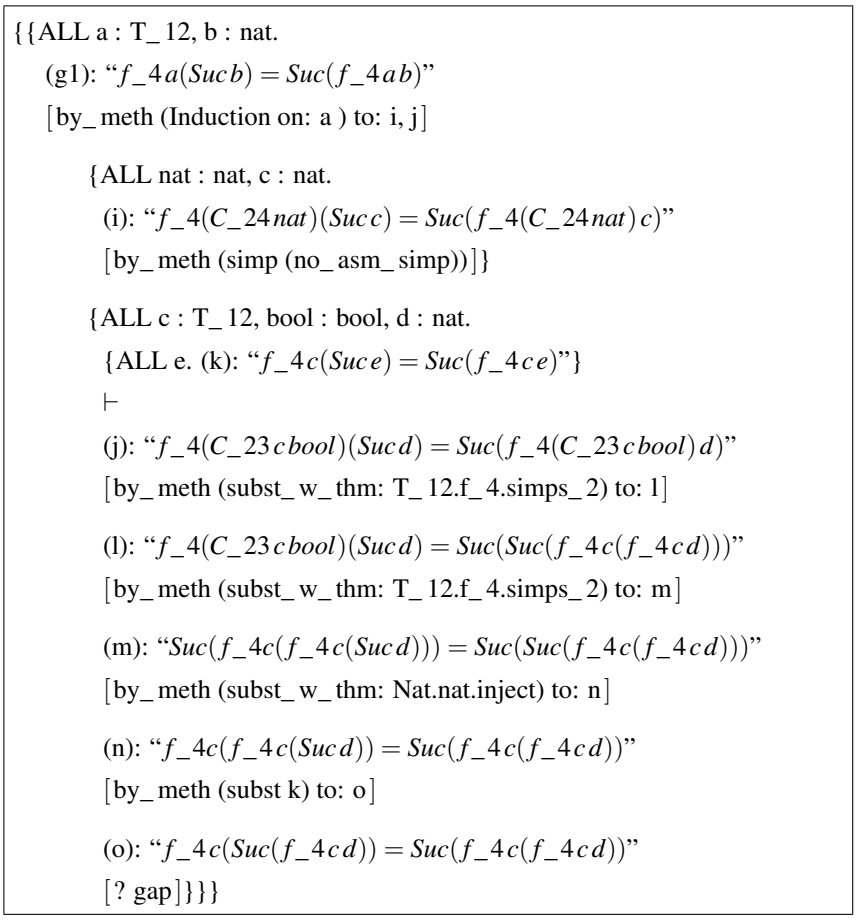


Figure 5 (proof plan): *(g1) is the main goal, (i) is the base step and (k) is the inductive assumption. From (j) to (n) rewriting is guided by rippling, and (o) is the remaining goal after one fertilisation. Like in the examples from section 3, it just takes another fertilisation to solve the problem.*

{ALL d : T_ 12, bool : bool, e : T_ 12, f : nat.
{ALL g h. (k): "$f\_4\,d\,(f\_4\,g\,h) = f\_4\,g\,(f\_4\,d\,h)$"}
⊢
(j): "$f\_4\,(C\_23\,d\,bool)\,(f\_4\,e\,f) = f\_4\,e\,(f\_4\,(C\_23\,d\,bool)\,f)$"
[by_ meth (subst_ w_ thm: T_ 12.f_ 4.simps_ 2) to: l]

(l): "$f\_4\,(C\_23\,d\,bool)\,(f\_4\,e\,f) = f\_4\,e\,(Suc\,(f\_4\,d\,(f\_4\,d\,f)))$"
[by_ meth (subst_ w_ thm: T_ 12.f_ 4.simps_ 2) to: u]

(u): "$Suc\,(f\_4\,d\,(f\_4\,d\,(f\_4\,e\,f))) = f\_4\,e\,(Suc\,(f\_4\,d\,(f\_4\,d\,f)))$"
[by_ meth (subst k) to: v]

(v): "$Suc\,(f\_4\,d\,(f\_4\,e\,(f\_4\,d\,f))) = f\_4\,e\,(Suc\,(f\_4\,d\,(f\_4\,d\,f)))$"
[? gap]}}}

Figure 6 (partial proof plan): *Here, only the step case is shown. It is clear that it only takes another fertilisation, instantiating h of the assumption as* $(f\_4\,d\,f)$ *to lead the goal to "*$Suc\,(f\_4\,e\,(f\_4\,d\,(f\_4\,d\,f))) = f\_4\,e\,(Suc\,(f\_4\,d\,(f\_4\,d\,f)))$*". This won't solve the problem directly, but the lemma calculator will find the common term* $(f\_4\,d\,(f\_4\,d\,f))$ *and calculate the generalised lemma "*$Suc\,(f\_4\,e\,n) = f\_4\,e\,(Suc\,n)$*", which, as we know from example 5, only requires one extra fertilisation to be solved. However, let us we go back and consider our options in goal (u). At this point, IsaPlanner lets us instantiate h as* $(f\_4\,e\,f)$*. If this is done, the goal will not change. For a single fertilisation there is no problem. However, for infinite capacity to fertilise, the prover could fall into a loop. This suggests a depth-first search strategy would be problematic. This is discussed further in sections 4.3 and 6.2.*

fertilisations, because it is easy to imagine, for any number n, a case in which $n+1$ fertilisations are required (define a function with $n+1$ nested instances of itself in the recursive). Most of the proofs analysed require no more than two, but it would be ugly and arbitrary to choose two as a limit.

However, let us analyse the possible problems arising from a method which fertilises unboundedly. Consider the conjecture $f_1\, a(Suc\, b) = Suc(f_1\, a\, b)$ from theory T_{12}, with f_1 defined as follows:

- **base:** $f_1(C_{24}\, a)\, b = Suc\, b$
- **step:** $f_1(C_{23}\, a\, b)\, c = Suc(Suc(f_1\, a(f_1\, a(Suc\, c))))$

The proof attempt for that conjecture is shown in figure 7

Example 7 shows that not only do some paths of fertilisation lead to a dead end, but that there is also such thing as over-fertilisation.

Summarising the analysis of the conjectures in which multiple fertilisations are required:

1. A lot of TheoryMine's open conjectures require multiple fertilisations
2. The number of fertilisations required varies for each proof.
3. There are options to fertilise a goal and, thus, many possible paths.
4. Some paths might be infinite loops.
5. Some paths might be dead ends.
6. Some paths are only successful if stopped at the right time.

These facts put constraints on the design for a method. First of all, the strategies "fertilise when possible" and "always chose the first option" are out of the question because of facts 4, 5 and 6. Thus, there seem to be three options for designing a method:

1. Restricting the branching in a structured way.
2. Opening all different paths in the search tree (including all the "stop fertilising" paths).
3. Opening all paths, but guiding the search.

{{ALL a : T_12, b : nat.
(g1): "$f\_1\,a(Suc\,b) = Suc(f\_1\,a\,b)$"
[by_meth (Induction on: a) to: i, j]

{ALL nat : nat, c : nat.
(i): "$f\_1(C\_24\,nat)(Suc\,c) = Suc(f\_1(C\_24\,nat)\,c)$"
[by_meth (simp (no_asm_simp))]}

{ALL c : T_12, bool : bool, d : nat.
{ALL e. (k): "$f\_1\,c(Suc\,e) = Suc(f\_1\,c\,e)$"}
⊢
(j): "$f\_1(C\_23\,c\,bool)(Suc\,d) = Suc(f\_1(C\_23\,c\,bool)\,d)$"
[by_meth (subst_w_thm: T_12.f_1.simps_2) to: l]

(l): "$f\_1(C\_23\,c\,bool)(Suc\,d) = Suc(Suc(Suc(f\_1\,c(f\_1\,c(Suc\,d)))))$"
[by_meth (subst_w_thm: T_12.f_1.simps_2) to: m]

(m): "$Suc(Suc(f\_1\,c(f\_1\,c(Suc(Suc\,d))))) = Suc(Suc(Suc(f\_1\,c(f\_1\,c(Suc\,d)))))$"
[by_meth (subst_w_thm: Nat.nat.inject) to: n]

(n): "$Suc(f\_1\,c(f\_1\,c(Suc(Suc\,d)))) = Suc(Suc(f\_1\,c(f\_1\,c(Suc\,d))))$"
[by_meth (subst_w_thm: Nat.nat.inject) to: o]

(o): "$f\_1\,c(f\_1\,c(Suc(Suc\,d))) = Suc(f\_1\,c(f\_1\,c(Suc\,d)))$"
[by_meth (subst k) to: p]

(p): "$f\_1\,c(Suc(f\_1\,c(Suc\,d))) = Suc(f\_1\,c(f\_1\,c(Suc\,d)))$"
[? gap]}}}

Figure 7 (proof plan): *In this proof attempt the base case is solved and the step case (j) is rippled to goal (o), which is then fertilised to goal (p). It can be seen that two other fertilisations can occur at this point, one instantiating e of the assumption as d, and the other as* $(f\_1\,c(Suc\,d))$. *If the first one is done we get the goal "*$f\_1\,c(Suc(Suc(f\_1\,c\,d))) = Suc(f\_1\,c(f\_1\,c(Suc\,d)))$*", which cannot be solved because there is no way to match the term* $(Suc\,d)$ *from the right side on the left side, and that term itself cannot be changed by fertilisation. Thus, one path of fertilisations can lead to a dead end. Moreover, if the right fertilisation is done and we get the goal "*$Suc(f\_1\,c(f\_1\,c(Suc\,d))) = Suc(f\_1\,c(f\_1\,c(Suc\,d)))$*" there is still a chance to undo the success up if fertilisation is unbounded, because then it would fertilise, instanciating e of the assumption as d, obtaining "*$Suc(f\_1\,c(Suc(f\_1\,c\,d))) = Suc(f\_1\,c(f\_1\,c(Suc\,d)))$*", raising the difference between both sides of the equation.*

I imagine a rippling-like heuristic could be designed to guide fertilisation. However, that is out of the scope of this project. Thus, we need to open all the paths and search in a space where there are loops and dead ends. For this reason, depth-first search is not an option.

Considering that there might be many paths that solve the problem breadth-first search might be extremely inefficient and we might better exploit the power of iterative-deepening search.

There is another question: should 0 fertilisations be allowed? That is, should the search branch to “end fertilisation" before it has fertilised? I think the answer is yes, and the reason is the following: first of all, the fact that the functions IsaCoSy gives us are defined recursively doesn't necessarily mean that induction is needed for the proof. However, it does mean that they are defined in two parts, independently of each other (one definition for the bases and one definition for the step cases). Thus, different proofs might be required for the base case and the step case (even if it is not an inductive proof!). As a matter of fact, such conjectures were found. They require 0 fertilisations, which means that induction just acts as a case split for the base cases and the rest.

Consider, for example theory T_3, with its type defined as follows:

- **base:** $C_5\ b\ n$, for any b boolean, and n natural.
- **step:** $C_6\ \tau\ m$, for any m natural and τ in T_3

Take the function $f_0 : T_3 \to (\mathbb{N} \to \mathbb{N})$ defined as follows:

- **base:** $f_0(C_5\, a\, b)\, c = c$
- **step:** $f_0(C_6\, a\, b)\, c = f_0\, a(Suc(f_0\, a(Suc(Suc\, b))))$

The conjecture to be proved is $f_0\, a(f_0\, a(f_0\, a\, 0)) = f_0\, a\, 0$, by induction over a. The base case is trivial and the step case

$$f_0(C_6\, b\, nat)(f_0(C_6\, b\, nat)(f_0(C_6\, b\, nat)\, 0)) = f_0(C_6\, b\, nat)\, 0$$

is simplified by the definition of f_0 as the following steps show:

$$\begin{aligned} f_0(C_6\, b\, nat)(f_0(C_6\, b\, nat)(f_0\, b(Suc(f_0\, b(Suc(Suc\, nat)))))) &= f_0\, b(Suc(f_0\, b(Suc(Suc\, nat)))) \\ f_0(C_6\, b\, nat)(f_0\, b(Suc(f_0\, b(Suc(Suc\, nat))))) &= f_0\, b(Suc(f_0\, b(Suc(Suc\, nat)))) \\ f_0\, b(Suc(f_0\, b(Suc(Suc\, nat)))) &= f_0\, b(Suc(f_0\, b(Suc(Suc\, nat)))) \end{aligned}$$

thus solving the problem.

Even though induction was not necessary, its procedure of splitting into base and step cases solves the problem because the this resembles the structure of the function's definition. Thus, branching before fertilising the first accounts for these kinds of proofs.

4.4 Other causes of failure

There are other more elusive causes of failure, briefly explained below.

Some of the proof attempts seem to just be too long, pointing at some kind of divergence created by the lemma calculation critic. Divergence has already been spotted, and a critic designed accordingly, which generalises over the lemmas that form the the divergent sequence Walsh (1994). However, we could not spot much of a pattern over such lemmas because they grew quickly and the time rippling took to work on them grew even more (and this growth was reflected in the base case, in the step case and in every sub-proof). Thus, only about two lemmas of a single line of divergence could be seen and that was not enough to know what was going on.

In a larger subset of open conjectures (about one fifth of the total amount of open conjectures) the prover just fails to fertilise in steps where it is actually obvious that fertilisation is possible, but the prover just does not see it. These cases might be due to bugs or could be pointing at a need for improving the way such techniques are applied.

4.5 Summary of the analysis

The two classes with most open conjectures were multiple fertilisations and generalisation. It turns out that, for all conjectures belonging to the generalisation class, their generalised version also appears in the list of open conjectures (except in a couple of cases where it actually appears in the list of theorems). This is very particular to the way IsaCoSy generates the conjectures.

Of the lemma speculation class there is also a dependency on theorems that can be proved by the multiple fertilisations technique.

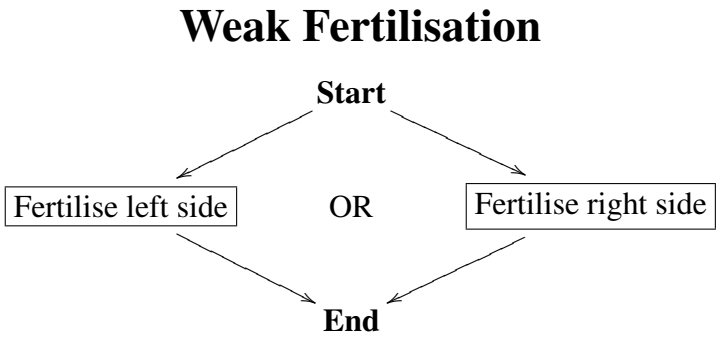
The basic constraints required for the multiple fertilisations extension to rippling were delineated in this section. It was out of this analysis that the method in which this project centred was designed and implemented. Some ideas that sprung out from the analysis, like guiding fertilisation through heuristics, were not implemented and are left open for further work.

Chapter 5

Design and implementation of the multiple fertilisations extension

Our concern for the implementation of our extension to IsaPlanner is in the functions in charge of the step case, whose structure has been explained in section 2.5:

Particularly, our work is in the weak fertilisation process, structured as follows:



IsaPlanner is written in the functional programming language ML. Its techniques are represented by functions that take a reasoning state and return the sequence of reasoning states which result from the different possible applications of the proof technique in the proof plan.

The code for the function that chooses the side where fertilisation is going to occur (represented by the *start* node in the diagram above, is as follows:

```
fun subst_skel goal skelnm rst =
    (orr_list [subst_in_lhs skelnm goal,
               subst_in_rhs skelnm goal]) rst;
```

The technique itself is *subst_skel goal skelnm*, which takes the reasoning state rst. The function *orr_list* takes a list of reasoning techniques and returns the technique that, given a reasoning state, applies to it all the techniques in the list and appends all the sequences of reasoning states they return. In this case, it appends the sequences that *subst_in_lhs skelnm goal* and *subst_in_rhs skelnm goal* return when given the reasoning state rst. These two functions are, respectively, the techniques that fertilise either the left side or the right side of the goal. Each has been built by a skeleton and a goal and, when applied to the reasoning state, changes the proof plan accordingly (unfold the goal into the fertilised subgoal).

The particular way weak fertilisation occurs is that the left side of the assumption is compared to the terms in the left side of the goal and, if there is a match (with the option to instantiate freely as terms the universal variables [not the inductive one!]), the matching term is substituted by the right side of the assumption equation. The right side of the assumption, on its part, is matched to the right side of the equation analogously.

The implementation of the unbounded multiple fertilisations technique was done by creating a recursion for each of the functions *subst_in_lhs skelnm goal* and *subst_in_rhs skelnm goal*. It was on these, instead of on *subst_skel goal skelnm*, to prevent mishaps like the following:

Suppose we are on the step case of conjecture $f\,a(Suc\,b) = Suc(f\,a\,b)$ and the goal gets rippled to $f\,a(f\,a(Suc\,b)) = Suc(f\,a(f\,a\,b))$. By fertilising twice on the left side we get steps (5.1) and (5.2)

$$f\,a(Suc(f\,a\,b)) = Suc(f\,a(f\,a\,b)) \tag{5.1}$$

$$Suc(f\,a(f\,a\,b)) = Suc(f\,a(f\,a\,b)) \tag{5.2}$$

the problem is solved. However, if the recursion was on *subst_skel goal skelnm*, $Suc(f\,a(f\,a\,b)) = f\,a(f\,a(Suc\,b))$ would be reached by fertilising all the way through.[1]

The code for our *middle-man* function, *subst_in_lhs_until_fail skelnm goal*, who manages the recursion, looks as follows:

[1] At the time of implementation we didn't know branching was going to be necessary and the search strategy changed. This problem wouldn't have been a big problem considering the facts. In any case, it would have made the search bigger.

```
fun subst_in_lhs_until_fail skelnm goal rst =
   let
     val newseq =
          (givename weak_fertN (RTechnEnv.subst_in_lhs skelnm false goal)) rst
   in
     case (Seq.pull newseq) of
       NONE => Seq.single (end_rst
                           (RstName.str ("end fertilisation on goal: " ^ goal))
                           rst |> RState.set_goalnames [goal] )
     | SOME _ => Seq.append
                 (Seq.map (RState.set_rtechn
                             (SOME (map_then (subst_in_lhs_until_fail skelnm))))
                          newseq)
                 (Seq.single (end_rst
                              (RstName.str ("end fertilisation on goal: " ^ goal))
                                           rst |> RState.set_goalnames [goal] ))
   end;
```

What the technique related to this function does is to create the sequence of reasoning states *newseq* out of the sequence that the original function *subst_in_lhs* returns. If this sequence is empty it means that there are no possible fertilisations. In such case, it will return the sequence with just one reasoning state, which has instructions to proceed to the next step in the proof (in the current structure of IsaPlanner, to try to solve by simplification or calculating a new lemma). If, on the other hand, the sequence is not empty, it will return a sequence with all the possible left-side fertilisations plus the reasoning state that ends fertilisation. This last reasoning state is there to have the option of interrupting fertilisation at any point. The function for the right side is analogous.

IsaPlanner already had functions for interactive and automated proving which were adapted to our tests. Its automated proving function, which was running on depth-first search, was changed to breadth-first search. The code for that function is the following:

```
fun automatic_rippling ctx goal =
    let
      val rst_opt =
        PPInterface.init_rst_of_strings ctx [goal]
         |> RState.set_rtechn (SOME (RTechnEnv.map_then
                                      MyRippling.induct_ripple_lemcalc))
         |> GSearch.breadth_fs
                 (fn rst => is_none (RState.get_rtechn rst)) RState.unfold
         |> Seq.filter RstPP.solved_all_chk
         |> Seq.pull;
      val string_thrm_opt =
            case rst_opt of
              NONE => NONE
            | SOME (rst,altrsts) =>
                 SOME (goal, SynthPrfTools.name_thrm_from_rst "g1" rst)
    in
      string_thrm_opt
    end;
```

IsaPlanner's interface is run on our extended techniques (*MyRippling.induct_ripple_lemcalc*), for a single goal, on breadth-first search. A reasoning state (*rst_opt*) is returned in case the proof plan has no gaps (and it is therefore a proof), of which the theorem is extracted and returned. Otherwise, a *NONE* is returned.

In this section, IsaPlanner's code and our extension were described. The general procedure of IsaPlanner with our extension, when taking a goal and a context, is to use the interface to do breadth-first search in the tree of reasoning states with applications of the reasoning techniques as edges. The order of the search is induced by the structure of the techniques, which is roughly as follows:

1. Begin proof by induction; unfold the goal into base cases and step cases into a single proof plan.

2. Try the base cases by simplification. If this fails try lemma calculation and try the resulting lemma by induction.

3. Start the step cases by rippling.

4. Try fertilisation (proceeding to step 5), or end step (prodceeding to step 6).

5. Repeat step 4.

6. Solve the remaining goals by simplification. If this fails proceed to lemma calculation and try the resulting lemma by induction.

Chapter 6

Evaluation

Evaluation was done by running the prover with our extension on a large sample of open conjectures. A number of experiments were done, to test for the success rate of our extension's proof capacity alone, as well as with different set-ups of the capacity to use the theorems proved, as lemmas in further proofs.

For such an enterprise, we needed a program that runs our extension of IsaPlanner through a list of conjectures. Due to the possibility of loops, we need a time limit for each proof. We devised two strategies for the prover to use theorems in further proofs:

1. Add the theorems as wave rules for rippling. As it was shown by the analysis, sometimes the failure leads to a goal which could be rippled successfully if using a lemma that appears in the list of open conjectures (this is particularly present in those conjectures that seemed to require lemma speculation).

2. Try direct proofs of the theorems, by checking if the conjecture is an instance of a previously proved theorem.

For the experiments, we tried all possible combinations of those strategies ($\emptyset$,{1},{2} and {1,2}).

To satisfy the first condition, our standard procedure would have been to build a program that updates the context of the reasoning state (the set of rewrite rules during rippling) every times it proves something new and reruns through the list until the number of remaining open conjectures reaches a fixed point. However, experiments with a program with such conditions revealed a problem with IsaPlanner's rippling: slowness. With more theorems available as wave rules, rippling takes more time. Given that there is a time limit for each conjecture, slowness in

rippling has an impact on the proof (and this is shown in the data). Nevertheless, experiments also showed that adding wave rules is not entirely useless. For some conjectures the time required is not as high, and cases exist, where a conjecture is proved with added wave rules but not otherwise (these cases are reflected in the data of our experiments, which will be shown in the 6.1 section). To get the best out of adding theorems as wave rules and avoiding the problem of time, we built our program in a way that it would add new wave rules only after going through the whole list without them.

To satisfy the second condition the program must keep the list of all proven theorems and try to prove each conjecture, before induction, by direct resolution with each of the theorems.

The code for such a program is divided into four functions:

1. A function that tries to prove a single conjecture by IsaPlanner's rippling technique with the multiple fertilisation extension, with a time limit.

2. A function that tries to prove a single conjecture directly from a list of theorems (as an instance of one of them).

3. A function that receives a list of conjectures and runs program 1 for each element of the list.

4. A function that runs 2 and 3 repeatedly, starting with the whole list of conjectures of a theory, updating the context at the end of each run, and feeding the remaining open conjectures to functions 2 and 3 in the next run, until its length reaches a fixed point.

5.

Functions 1, 2, 3, 4 are shown in figures 8, 9, 10 and 11, respectively.

Slight variations of these functions were done for each of the experiments. The conditions and results of such experiments are shown and explained in the next section.

```
fun a_rippling_one_goal_timeout ctx goal =
    let
      val prf_time = Unsynchronized.ref Time.zeroTime;
      val timeout = Time.fromSeconds 10;
      val ripple_res = Unsynchronized.ref NONE;
      val timenow = Timer.startRealTimer();
      val nap = Time.fromReal 0.25;
      val ripple = Thread.fork (fn () =>
                                  ripple_res :=
                                    (automatic_rippling ctx goal))
      fun timeout_chk timer =
          if (Timer.checkRealTimer timer) >= timeout then
            let val _ = if (Thread.isActive ripple)
                        then Thread.kill ripple else ()
                val _ = prf_time := !prf_time + timeout
            in NONE
            end
          else
            if (Thread.isActive ripple)
            then ((OS.Process.sleep nap); timeout_chk timer)
            else ripple_res
    in
       timeout_chk timenow
    end;
```

Figure 8: *This function creates a thread in which it applies automatic rippling (with the multiple fertilisations extension). The timer kills the fork if the time limit is exceeded (in the case of this particular code, it is 10 seconds per proof). It returns the same as the automatic rippling program, if this one finds a proof in due time, or a* NONE *if it runs out of time before finding a proof.*

```
fun prove_directly_one ctx ann_thms goal =
    let
      fun annthms_to_thms [] thm_list = thm_list
        | annthms_to_thms ((_,x)::t) thm_list = annthms_to_thms t (x::thm_list)
      fun rule_thm_dtac1 th =
          DTac.mk (K (Pretty.str
                        ("Resolution backward using thm "
                                 ^ (Thm.get_name_hint th))),
                    GTacs.of_bckf (PPlan.apply_rule_thm th));
      fun prove_with_list_of_thms thms g =
          (RTechnEnv.orr_list
              (map (fn th => (RTechnEnv.apply_dtac (rule_thm_dtac1 th))) thms)) g
      val thms = annthms_to_thms ann_thms []
      val rtechn = prove_with_list_of_thms thms
      val rst_opt =
      PPInterface.init_rst_of_strings ctx [goal]
           |> RState.set_rtechn (SOME rtechn)
           |> GSearch.depth_fs
                 (fn rst => is_none (RState.get_rtechn rst)) RState.unfold
           |> Seq.filter RstPP.solved_all_chk
           |> Seq.pull;
      val string_thrm_opt =
          case rst_opt of
              NONE => NONE
            | SOME (rst,altrsts) =>
              SOME (goal, SynthPrfTools.name_thrm_from_rst "g1" rst)
    in
      string_thrm_opt
    end;
```

Figure 9: *The function* one_step_updt_ctx_end *(figure 10) feeds it a list with theorems with annotations. The first thing this function does is to take that annotation out (*annthms_to_thms*). Then, it invokes a tactic to prove a goal with a given theorem (*PPlan.apply_rule_thm in rule_thm_dtac1*), and tries such tactic for every theorem in the goal (*prove_with_list_of_thms*). This is done through the IsaPlanner's interface's automatic search. It returns* NONE *if it finds no proof or the theorem (taken out of the reasoning state) if it finds a proof.*

```
fun one_step_updt_ctx_end ctx provedconjs [] openconjs =
                                                (provedconjs,openconjs)
  | one_step_updt_ctx_end ctx provedconjs (g::gs) openconjs =
    let
      val ann_thm_opt =
          case prove_directly_one ctx provedconjs g
            of NONE => let val (x,_) = a_rippling_one_goal_timeout ctx g
                       in case x of
                          NONE => NONE
                        | SOME (_,thm_rmf) => SOME ("RipplingWithMF",thm_rmf)
                       end
             | SOME (_,thm_d) => SOME ("Directly",thm_d)
      val (newprovedconjs,newopenconjs) =
        case ann_thm_opt
          of NONE => (provedconjs, g::openconjs)
           | SOME ann_thm => (ann_thm::provedconjs, openconjs)
    in
      one_step_updt_ctx_end ctx newprovedconjs gs newopenconjs
    end;
```

Figure 10: *This function runs through the list of goals it is fed with, keeping two lists (proved conjectures and open conjectures) which it updates simultaneously. For each goal, it first tries to prove it directly by function* prove_directly_one. *If it is proven it annotates the theorem with the label "Directly". Otherwise, it proceeds with rippling by function* a_rippling_one_goal_timeout *and labels the theorem with "RipplingWithMF" if proved. Otherwise, it returns a NONE. If proven, it adds it to the list of proved theorems, and otherwise it adds it to the list of open conjectures. It then feeds itself the updated lists and proceeds with the next goal in the list.*

```
fun top_level_updt_ctx_end ctx proved gs =
    let
      val (newproved, newgs) = one_step_updt_ctx_end ctx proved gs []
      fun add_ann_thms_to_ctx ctxt [] = ctxt
        | add_ann_thms_to_ctx ctxt (h::t) =
          let
            val (_,thm) = h
            val newctxt = SynthPrfTools.add_to_wrules thm ctxt
          in add_ann_thms_to_ctx newctxt t
          end;
      val newctx = add_ann_thms_to_ctx ctx newproved
    in
      if length gs = length newgs then (newgs,newproved)
      else top_level_updt_ctx_end newctx newproved newgs
    end;
```

Figure 11: *This is the main function for the evaluation. It takes a context*[a]*, a list of conjectures and a list of proved theorems. Each run it sends the context and both lists to function* one_step_updt_ctx_end*, which returns the updated lists of proved theorems and remaining open conjectures. Then it processes the list of proved theorems through a function that updates the context*[b] *given a list of theorems (*add_ann_thms_to_ctx*). It repeats the process while there are new theorems being proved (expecting that, in the next run, something might be proven by them either by direct application or by the wave rules generated from them in the updated context).*

An important feature of this function is that it only updates the context once it has gone through the whole list without updating it and thus saving us from the problem of rippling turning slow and not proving stuff.

[a] The context is created at first by the information that may be used in the proof, i.e. the axioms of the theory.

[b] Updating the context consists of adding the proved theorems as wave rules.

6.1 Results

First of all, it was required to check if the extension built into IsaPlanner was at least as successful as IsaPlanner without such an extension. It is not obvious that this would be so because of the extra branching that our methods add plus the disadvantages breadth-first search carries (previous to our work, IsaPlanner was running on depth-first search). The time limit could have chopped a previously successful proof before it finished because of unnecessary branching and inefficient search. Fortunately, all theorems previously proved by IsaPlanner were proved by its extension. So, if there are any ugly effects out of the inefficiencies of our program at least they are not reflected there. That being said, we can concentrate on the real results.

Four different experiments were made, testing for specific information about the performance of our program. Each is explained next:

1. **Rippling with multiple fertilisations** There are two reasons for making this experiment. The first one is the same as the reason for the previous experiment; taking advantage of IsaCoSy's particular way of working might be cheating. The second one is just to test for how much of the proofs actually require multiple fertilisations.

2. **Rippling with multiple fertilisations, adding proved theorems as wave rules.** It helps to see the power of the method, considering that direct proofs might have not let us see if the methods would have proven the conjectures anyway. This experiment also accounts for the fact that the logical dependencies (the fact that a lot of conjectures are generalisations of others) are "artificial"; caused by the IsaCoSy's particular way of working.

3. **Rippling with multiple fertilisations plus direct application of theorems.** This shows, first of all, the strength of direct application because of the logical correlation between the theorems and, secondly, shows the difference between direct application and adding of proved theorems as wave rules.

4. **Rippling with multiple fertilisations adding proved theorems as wave rules, plus direct application of theorems.** The full power of our methods is seen, though we might be losing some fine information about the power of the specific parts.

Experiments 1, 2 and 3 were done for theories all conjectures belonging to theories T_2, T_3 and T_{12} (notice that T_2 and T_{12} belong to the class of funny naturals while T_3 belongs of the class funny $\mathbb{N} \times \mathbb{N}$; this, to get some variety). Experiment 4 was done to all conjectures of all theories.

Theory	# OC	(R + T) + D			R + D	R + T	R
T_2	102	56	R+T	D	52	37	28
			31	25			
T_{12}	68	38	R+T	D	36	28	15
			17	21			
T_3	36	19	R+T	D	19	17	17
			17	2			
T_6	64	36	R+T	D			
			19	17			
T_1	40	24	R+T	D			
			16	18			

Table 1: *#OC is the number of open conjectures. (R + T) + D is the result for experiment 1. R + D is the result for experiment 2. R + T is the result for experiment 3. R is the result of experiment 4. Inside each cell we show the number of conjectures proved for each of the trials. For example, in theory T_2 there were 102 open conjectures, of which the prover with all techniques active proved 56, 31 of them by rippling and 25 directly. For the same theory the prover with rippling plus theorems proved 37 and just with just rippling it proved 28. Note that R + T inside (R + T) + D is smaller than R + T alone. That means that direct proving prevented the prover from trying rippling for some goals which would have proven anyway. All of these experiments were done with 10 seconds for each conjecture.*[a]

For the experiments that were not done, the space was left blank.

[a] All experiments were done on a PC with processor Intel(R) Core(TM)^2 Duo CPU P8700 @ 2.53GHz, with 4Gb of RAM

Furthermore, experiments were done with 30 seconds for each conjecture for theories T_2, T_{12} and T_3 getting getting the following results:

Theory	**# OC**	**10s**	**30s**
T_2	102	56	60
T_{12}	68	38	42
T_3	36	19	19

This shows that time is indeed a problem for the prover and improving its search strategy could improve its performance.

On all counts (all experiments) there is a considerable success for the extension. The total amount of conjectures proved by this extension is $\sim$56%. Moreover, of the number of theorems proved only by the method itself is above one third.

If we compare our results with TheoryMine's success rate before the extension we get some interesting numbers. Before the extension TheoryMine had proved $\sim$14% of the conjectures of the samples analysed for this project. After the extension it proves $\sim$60% of them.

Most of the conjectures, in which the need to fertilise multiple times had been spotted, were proved. Some of them were not proved because the multiple fertilisation helped only prove a lemma, but some other gaps were left in the proof. A much bigger number of conjectures in which a proof wasn't expected, were proved. Most of them by direct application. However, it is also interesting to see that a big number of them were proved with the theorems added as wave rules (as is shown by the difference between the results for R and for R + T), and some of them couldn't have been proved directly (as shown by the difference between the results for (R + T) + D and R + D).

6.2 Discussion & further work

Even though there is a large success for our methods in proving TheoryMine's open conjectures, we have to question how general the work is. The issue at hand is whether our methods work only because of IsaCoSy's very particular way of generating functions and conjectures, or because of an actual broad requirement for multiple fertilisations in inductive proofs. Finding out whether this is true needs time and open eyes. Given that this pattern had not been spotted before, I believe it is more of a peculiarity of TheoryMine's conjectures and functions. However, from the analysis of the functions (as shown in section 3) we can see that they can be understood to

represent interesting number-theoretic functions. This might imply that, even though this proof pattern is not so common, it has a broader potential; a large set of number-theoretic theorems might be proved with these methods even if there is a standard proof without them by using definitions of functions in nested form. For example, the function $2^x + y$ can be expressed in a nested way as follows:

- **Base:** $f(0,y) = y+1$
- **Step:** $f(x+1,y) = f(x, f(x,y)+1)$

and thus, theorems regarding the function might follow the pattern we spotted.

We have mentioned before that the fact that our technique raises the size of the search space by branching might be a problem. Before implementing our extension into IsaPlanner, the search space was quite small. For that case, the search strategy made not much of a difference. However, with our extension, the search strategy has become an issue.

It was stated how depth-first search was out of the question. Nevertheless, for the extra branching that our method introduces, a better search strategy is required. The breadth-first search we used temporally filled a requirement depth-first just couldn't fulfil. However, the fact that a significant number of theorems were proved in the experiments with 30 seconds, that wouldn't otherwise be proved, suggests that it is necessary to improve it further. Moreover, one example in which a proof was found interactively but not automatically (even with 30 seconds) was found. We proposed iterative deepening search, which wouldn't have the disadvantages of memory and slowness that breadth first search poses, considering the search space probably has multiple win-paths.

As an alternative, we briefly mentioned the possibility of guided fertilisations much in the style of rippling. It is clear how such a method would be helpful in cases like those shown in figures 6 and 7 where loops, over-fertilisation and dead ends were possible. It could also bring back the possibility of using depth-first search. I hypothesise that rippling could done with the inductive assumption as a wave rule, trying to make one side of the equation look like the other, thus skipping the middle step of making the goal look like the skeleton. This hypothesis could be tested in future work.

Furthermore, an analysis analogous to the one done for this project could be done on the new list of open conjectures and the new proof attempts generated by the extended IsaPlanner. This work could be used as a starting point. A limited amount of analysis has been done for the new

open conjectures. Most of them seem to belong to the class of section 4.4, as expected. This is because, as mentioned before, for the generalisation and the lemma speculation class there is a large logical dependency with those theorems proved by multiple fertilisations and, thus, they are proved by the methods implemented to use theorems in new proofs. To analyse that class it is necessary to look closely at the fertilisation methods at the lowest level of IsaPlanner's architecture.

The remaining cause of failure that remains open, then, is the elusive divergence. I have mentioned in section 4.4 how its analysis has proved very difficult because of the slowness in rippling when terms get big (regardless of the amount of lemmas added as wave rules, which was the other reason for the slowness of rippling). Further work is necessary to understand divergence.

Chapter 7

Conclusion

The initial assumption, that by analysing the failed proof attempts a useful classification could be developed, out of which proof techniques could be devised, was true. We found three classes out of which the techniques required for each were outlined. Two of them had already been discussed in the literature. One of them had not been implemented in IsaPlanner and the other one wasn't in working order. For the new one; the one that hadn't been discussed before, we designed the corresponding technique and implemented it into IsaPlanner with success.

Although the technique implemented strictly raised the success rate of IsaPlanner in the tests done, some problems were spotted and discussed, along with outlines of their solutions and improvements. This means there is further work to be done. Thus, the work presented in this dissertation should be understood as an experiment on the principle of multiple fertilisations. The results of the experiment point the way for the future.

Bibliography

Bundy, A., Cavallo, F., Dixon, L., Johansson, M., and McCasland, R. (2010). The theory behind theorymine. In *Automatheo*. FLoC.

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

Bundy, A., van Harmelen, F., Hesketh, J., and Smaill, A. (1991). Experiments with proof plans for induction. *Journal of Automated Reasoning*, 7:303–324. Earlier version available from Edinburgh as DAI Research Paper No 413.

Cavallo, F. (2009). Vanity theorem proving. Undergraduate Dissertation, University of Edinburgh.

Dixon, L. (2006). *A Proof Planning Framework for Isabelle*. PhD thesis, University of Edinburgh.

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

Ireland, A. and Bundy, A. (1996). Productive use of failure in inductive proof. *Journal of Automated Reasoning*, 16(1–2):79–111. Also available from Edinburgh as DAI Research Paper No 716.

Johansson, M. (2009). *Automated Discovery of Inductive Lemmas*. PhD thesis, School of Informatics, University of Edinburgh.

Johansson, M., Dixon, L., and Bundy, A. (2009). Isacosy: Synthesis of inductive theorems. Workshop on Automated Mathematical Theory Exploration (Automatheo).

Johansson, M., Dixon, L., and Bundy, A. (2010). Dynamic rippling, middle-out reasoning, and lemma discovery. In *Walther Festschrift*, number 6463 in Springer. LNCS.

Paulson, L. C. (1994). *Isabelle: A generic theorem prover*. Springer-Verlag.

Walsh, T. (1994). A divergence critic. *Automated Deduction–CADE-12*, pages 14–28.