Automated Lemma Generation and Multi-Context Schemes for Rewriting Induction

Chengcheng JI

A DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

INFORMATION SCIENCE AND TECHNOLOGY

at the

GRADUATE SCHOOL OF INFORMATION SCIENCE AND TECHNOLOGY,

HOKKAIDO UNIVERSITY

December 2018

Acknowledgements

I would like to express my gratitude to all those who helped me during the writing of this dissertation. I am very grateful to my supervisor Prof. Masahito Kurihara for his assistance and advices. I also thank my co-supervisors, Prof. Satoshi Oyama, Prof. Masahito Yamamoto, Prof. Hidenori Kawamura and Prof. Tetsuo Ono for their valuable comments. I am especially appreciative of Prof. Haruhiko Sato for his assistance and advices. Their contribution was indispensable for the progress of this research. I would like to thank to my colleagues in our laboratory and all my friends for supporting me. Finally, I would like to thank to my beloved family for their loving considerations and great confidence in me through all these years.

Contents

List of Tables

Chapter 1

Introduction

Algebraic structure often appears in various theories in computer science as a means of representing systems in a formal way and reasoning about their properties such as their correctness. Related technologies include term rewriting, equational reasoning, and their application to formal methods in software engineering.

Executable implementation of these technologies often involves *nondeterministic computation*. In nondeterministic computational processes, users and/or algorithms make a series of choices (or decisions) at the beginning and subsequent temporal points (or choice points). In this dissertation, we refer to such a series of choices as a *context* for such a process. Naturally, one hopes that the context will lead to success defined for such a computation, but in general, it is not an easy task to make a right decision to lead the nondeterministic computation to success. Surprisingly, however, a lot of researchers have managed to make their computation lead to success by setting parameters and strategies beforehand to control the nondeterminism 'appropriately'. This often involves a lot of handmade trial-and-errors and/or tricks to suppress the nondeterminism in their experiments. However, the author believes that most of them will admit that there was a fairly amount of accumulation of inappropriate settings and failures before their 'success'. Thus, we should develop a highly universal technology to solve this problem to lead nondeterministic computation to a real success.

In a simple computational system, what is necessary is just backtracking, go-

ing back to the previous choice point when we have had a failure. In a complex system with an unlimited search space, however, backtracking is often impossible, because the system may be run indefinitely without success or failure. Therefore, a *concurrent computation* (or a sequential computation simulating concurrency) is necessary here. However, a naive implementation of such a concurrency often learns the hard way, facing the reality in which the number of processes grows exponentially too large to be practical. To solve this problem, Kurihara and other researchers [28, 35, 36, 37] have been developing a computational system in which a lot of computation and reasoning can be done efficiently in a single process. Its idea is based on an empirical knowledge that processes with 'similar' contexts often have a lot of mutually-related computational tasks which can be carried out simultaneously as they involve exactly common computation. Though the average computational complexity may be still exponential even in such a system, one can handle a larger size of problems in practice by suppressing the base of the exponential function with the special mechanism implementing the idea described above.

In this dissertation, the systems for running mutually-related virtual processes efficiently as a physical single process is referred to as *multi-context reasoning systems* [28] [35] [23] [24] [22]. Those systems are used to reason about algebraic computational systems such as *term rewriting systems* (TRSs), which are a concise and rigorous representation of computational systems in terms of rewrite rules. In fact, TRSs are studied and used in various areas of computer science, including automated theorem proving, analysis and implementation of abstract data types, and decidability of word problems.

The number of manipulations over the database increases when the size of problems gets larger. Although most of the common mutually-related manipulations can be merged into a single process, there may still exist plenty of duplicated calculations which can degrade the performance. *Lazy evaluation* is a powerful evaluation scheme for dealing with large databases, because it evaluates an executable expression only when it is called for the first time and then stores the result to avoid dupli-

cated calculations. We exploit this mechanism in our implementations to improve the efficiency.

1.1 Multi-Context Completion System

The well-known procedure for the completion of TRS was invented by Knuth and Bendix [27] in 1970 and affected a lot of researchers since then. Given a set of equations and a reduction ordering on a set of terms, the procedure (called KB in this dissertation) uses the ordering to orient equations (either from left to right or from right to left to transform them into rewrite rules) and tries to generate a complete TRS equationally equivalent to the input set of equations. The resultant TRS can be used to decide the equational consequences (word problems) of the input equations.

Actually, however, the KB leads to three possible results: success, failure, or divergence. In the success case, the procedure stops and outputs a complete TRS. In the failure case, the procedure stops but only returns a failure message with an unorientable equation. In the divergence case, the procedure falls into an infinite loop, trying to generate an infinite set of rewrite rules. The result of KB seriously depends on the given reduction ordering. With a good ordering, it would lead to a success, but otherwise, it would cause the failure or the divergence. In the latter case, we could try to avoid them by changing the ordering to appropriate one, but the problem is that it is very difficult for ordinary software designers and AI researchers to design or choose an appropriate ordering.

Therefore, automatic search for appropriate orderings is desired. But according to the possibility of divergence, we cannot try candidate orderings one by one. Also, it is not efficient to simply create processes for each different ordering and run them in parallel on a machine, because the number of candidate orderings normally exceeds ten thousands even for a small problem.

In 1999, this problem was partially solved by a multi-context completion procedure called MKB [28]. MKB is a single procedure that efficiently simulates execution

of multiple processes each running KB with a different reduction ordering. The key idea of MKB lies in a data structure called node. The node contains a pair $s : t$ of terms and three sets of indices to orderings to show whether or not each process contains rules $s \to t$, $t \to s$, or an equation $s = t$. The well-designed inference rules of MKB allows an efficient simulation of multiple inferences in several processes all in a single operation.

1.2 Multi-Context Rewriting Induction

An algebraic inductive theorem is a proposition for algebraic specifications defined on inductively-defined data structures such as natural numbers and lists. The proof of such inductive theorems based on equational logics has been studied for decades. The frameworks for inductive theorem proving are classified into two categories: the *explicit* induction which directly follows the paradigm of inductions, and the *implicit* induction in which some part of the paradigm is implicit in the sense that, typically, the induction rule and/or the well-founded induction order need not be provided explicitly by the users [43]. These two types of induction both have to find an induction pattern (a finite cyclic representation) for an infinite deductive proof and an induction order for ensuring the termination. The theorem prover Nqthm developed by Boyer&Moore [10] stands for the former one, because it requires that the induction patterns be provided as inference rules. Generally speaking, however, providing such induction patterns is difficult in practice. The *inductionless induction* proposed and extended by [29] [17] falls into the latter category, because, with no induction patterns provided, it implicitly tries to prove inductive theorems based on the principle of ground completion. However, this method was claimed to be so inefficient and practically useless, because, based on the notion of "proof by consistency", it has to search all over the search space for an inconsistency before it concludes that there is no inconsistency and thus the given formula should be certainly an inductive theorem [43]. On the other hand, the *term rewriting*

induction (RI for short) proposed by Reddy [34] implicitly obtains the induction patterns by the inference procedure itself and in this sense, it is a practically promising framework for implicit induction.

However, there are several kinds of strategic issues in constructing successful proofs by RI:

- which reduction order should be applied
- which (axiomatic or hypothetical) rules should be applied during rewriting
- which variables should be instantiated for induction

It is not a trivial task to choose appropriate strategies in general, because of the non-determinism during the induction procedure. Since inappropriate ones can easily lead the procedure to divergence (i.e., infinite computation), we cannot physically create and run a number of parallel processes because such naive parallelization would cause serious inefficiency. This makes it really hard to fully automate the RI-based inductive theorem proving.

Aoto [2] proposed a variant of RI, called the rewriting induction with termination checker (RIt), which partially solved the problem by using an external automated termination checker instead of a specific reduction order. Therefore, the users no longer had to provide promising reduction orders and they could implicitly exploit modern termination proving methods more powerful than the simply parameterized reduction orders (e.g., recursive path orders and polynomial orders). However, a new issue came out: in which direction the hypothetical equations should be oriented. Since the use of the termination checker increased the possibility of success in the orientation and we could decide the direction of the equations dynamically, more flexibility in the orientation strategy was given, from the viewpoint of strategy. Based on a multi-context strategy a procedure called *multi-context rewriting induction with termination checker* (MRIt) has been proposed by Sato [35] in order to exploit such flexibility in orientation and solve the other strategic issues of RI.

1.3 Lemma Generation

In general, an execution of RI leads to one of the following three results: success, failure, or divergence. The *divergence* occurs when the procedure generates in vain an infinite sequence of conjectures which cannot be proved automatically as lemmas for establishing the target theorem. To avoid the divergence, the users often need to supply appropriate lemmas which can be proved and used to solve the overall problems, but in practice, requiring their mathematical intuition and experience, this is so difficult to general users.

Automated lemma generation, therefore, is desired. Generally, there are two categories for lemma generation methods: *bottom-up* and *target-aimed* (top-down). The bottom-up methods generate lemmas from the given equational axioms with no consideration of the target theorem [25]. These methods have outstanding ability of generating conjectures, but their computational cost is extremely high. Meanwhile, the target-aimed methods work in a different way by considering the candidate conjectures to speculate appropriate lemmas [41] [31]. Even though the generating power is limited, the computational cost is comparatively low. Thus it is desirable to strengthen the power of the target-aimed method while preserving its acceptable cost.

Target-aimed methods are classified into *sound* and *unsound* ones. The sound methods [31] generate only correct conjectures in the sense that the goal is an inductive theorem if and only if the generated conjectures are inductive theorems. These methods have a very low computational cost, but the ability of generating appropriate lemmas is extremely low. On the other hand, the unsound methods [41] try to generate useful conjectures without being restricted by the soundness. This gives higher ability of generating appropriate lemmas with a modest computational cost. We focus on the latter one to keep balance between power and cost. Basically, the unrestricted use of classic generalization techniques for the unsound methods based on replacing constants and ground terms with universally-quantified vari-

ables limits the practical usefulness. The framework based on divergence-detection proposed by Walsh [41] greatly improves the practical usefulness by the following steps: 1) detect a potential divergence from the sequence of generated conjectures; 2) generate candidate lemmas by locating the differences between two consecutive conjectures in the diverging sequence.

We put into the Walsh's framework a new heuristic lemma generation method, *peripheral sculpture*, to make the theorem prover more powerful without introducing significant cost increase. Since the method is unsound, there is no guarantee of the correctness of the generated candidate lemmas. If the system accepts a wrong lemma, it could cause an infinite sequence of wasteful inferences. Therefore, we need to separate into parallel contexts the choices of whether to accept the candidate lemma or not. From this point of view, the combination of multi-context induction and lemma generation can be a new research area of automated reasoning. As a means to study this area, we extended the efficient *multi-context rewriting induction system* of Sato and Kurihara [35] with our lemma generation method. The experimental results show that, with no much redundant costs, we have succeeded in solving several lemma-required benchmark problems which encountered complex differences (i.e., parallel and nested differences) and which the original systems [35] [41] could not solve.

1.4 Overview

In this dissertation, we proposed new automated lemma generation methods for multi-context schemes. We also present: 1) new implementations of multi-context completion system *lz-mkb* based on MKB, 2) new implementations of multi-context algebraic inductive theorem prover *lz-itp* based on MRIt, both of which efficiently simulate the execution of parallel KB/RIt processes by dynamically dealing with the nondeterministic choices. Because these systems rely on the manipulation of the *node* database, we exploit the lazy evaluation schemas [23] [24] to gain more

efficiency.

In Chapter 1, we introduce the background of our research which, in the field of Artificial Intelligence, is a subject including term rewriting, equational reasoning, and their application to formal methods.

In Chapter 2, we introduced the basis of term rewriting systems (TRSs) and two TRS-based automated reasoning systems. Starting from a well-known completion procedure KB, we show the details of rewriting induction procedure RI/RIt.

In Chapter 3, we start with an unsound automated lemma generation method with divergence detection, then propose a new automated lemma generation method: *peripheral sculpture* to handle the lemma-generation problems for rewriting induction.

There are several problems of KB or RI/RIt in practice. Firstly, the problem of KB is that it is difficult to set an appropriate reduction orders before the procedure starts. Secondly, the problem of RI/RIt is that the nondeterministic choices may cause the procedures to end up with failing results (i.e., divergence or fail). Thirdly, appropriate lemmas are often required in practice for RI/RIt to achieve successful proofs. The first problem can be partially solved by the multi-context reasoning system MKB which simulates the mutually related processes in a single process efficiently. As for the second problem, MRIt pursues all nondeterministic choices trying to lead the procedure to a successful result by exploit the schema of MKB. We also expand our method into multi-context schema to offer a practically useful framework for inductive theorem provers. We introduce these two systems in Chapter 4.

In Chapter 5, we discuss the lazy evaluation technique we embed in our system as well as other details of implementation of multi-context reasoning systems: *lz-mkb* and *lz-itp*.

In Chapter 6, we summarize and show the results of our experiments. These results are pleasing in the sense that 1) the newly proposed lemma generation method worked well with other methods in multi-context framework and solved some lemma-

required problems that could not be solved by the original methods; 2) the new implementation of MKB and MRIt with lazy evaluation are more effective than the original ones. We give conclusions in Chapter 7.

Chapter 2

Preliminary

We briefly review the basic notions for term rewriting systems(TRS) [5] [33] [6] [12] [40]. We start with the basic definitions.

2.1 Term Rewriting System

A signature Σ is a set of *function symbols*, where each $f \in \Sigma$ is associated with a non-negative integer n, the *arity* of f. The elements of Σ with arity n=0 are called *constants*. Let V denote a set of *variables* such that $\Sigma \cap V = \emptyset$.

Definition 2.1.1. A *term* is either a variable or a function symbol (of arity $n \geq 0$) followed by n terms as arguments (possibly delimited by commas and enclosed by parentheses).

Definition 2.1.2. The set $T(\Sigma, V)$ of all *terms* over Σ and V is recursively defined as follows: $V \subseteq T(\Sigma, V)$ (i.e., all variables are terms) and if $t_1, \ldots, t_n \in T(\Sigma, V)$ and $f \in \Sigma$, then $f(t_1, \ldots, t_n) \in T(\Sigma, V)$, where n is the arity of f.

Example 2.1.3. Let f be a function symbol with arity 2 and $\{x, y\}$ are variables, then $f(x, y)$ is a term.

We write $s \equiv t$ when the terms s and t are identical. The set of variables occurring in a term t is represented by $\mathcal{V}(t)$. When the function symbol is binary (i.e., of arity 2)

and its name consists of special characters (such as $+$ and :), the term may be written in infix form (such as $0 + x$ and $x : nil$).

Definition 2.1.4. A term s is a *subterm* of t, if either $s \equiv t$ or $t \equiv f(t_1, \ldots, t_n)$ and s is a $subterm$ of some $t_k (1 \leq k \leq n)$.

Definition 2.1.5. A *substitution* σ is a mapping from V to $T(\Sigma, V)$ such that $\sigma(x) \neq x$ for only finitely many xs, and is extended to a mapping from $T(\Sigma, V)$ to $T(\Sigma, V)$ by defining $\sigma(f(s_1, \ldots, s_m)) = f(\sigma(s_1), \ldots, \sigma(s_m))$, where m is the arity of f.

Definition 2.1.6. The *domain* of σ is the set $Dom(\sigma) = \{x \in V \mid \sigma(x) \neq x\}$. We write a substitution σ as $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ if $\sigma(x_i) = t_i$ for $x_i \in \{x_1, \ldots, x_n\}$ and $\sigma(x) = x$ for $x \notin \{x_1, \ldots, x_n\}$.

The application $\sigma(s)$ of σ to s is often written as $s\sigma$. A term t is an *instance* of a term s if there exists a substitution σ such that $s\sigma \equiv t$. If $s\sigma$ is a ground term (containing no variables), it is a *ground instance* of s.

Definition 2.1.7. A substitution σ that replaces distinct variables by distinct variables (i.e. σ is injective and $x\sigma$ is a variable for every x) is called a *renaming*.

Definition 2.1.8. Two terms s and t are *variants* of each other and denoted by $s \cong t$, if s is an instance of t and vice versa (i.e., s and t are syntactically the same up to renaming variables).

Now we can define TRS as follows:

Definition 2.1.9. A *rewrite rule* $l \to r$ is an ordered pair of terms such that l is not a variable and every variable contained in r is also in l.

Definition 2.1.10. A *term rewriting system (TRS)*, denoted by $\mathcal{R}$, is a set of rewrite rules.

When we use TRS to solve completion problems, some properties such as *termination* and *confluence* are expected to hold most of the time. To talk about those properties, we need more definitions as follows.

Definition 2.1.11. Let $\Box$ be an extra constant called a *hole*. A *context* C is a term in $T(\Sigma \cup \{\Box\}, V)$. If C is a context with *n* occurrences of holes and $t_1, \ldots, t_n$ are terms, then $C[t_1, \ldots, t_n]$ is the result of replacing the holes by $t_1, \ldots, t_n$ from left to right. The *empty* context consists of only a single hole.

Definition 2.1.12. The *reduction relation* $\rightarrow_R \subseteq T(\Sigma, V) \times T(\Sigma, V)$ is defined by $s \rightarrow_{\mathcal{R}} t$ iff there exists a rule $l \rightarrow r \in \mathcal{R}$, a context C, and a substitution σ such that $s \equiv C[l\sigma]$ and $C[r\sigma] \equiv t$. A term s is *reducible* if $s \rightarrow_{\mathcal{R}} t$ for some t; otherwise, s is a *normal form*.

Definition 2.1.13. A TRS $\mathcal{R}$ *terminates* if there is no infinite rewrite sequence $s_0 \rightarrow_{\mathcal{R}} s_1 \rightarrow_{\mathcal{R}} \cdots$. We also say that $\mathcal{R}$ has the $termination$ property or R is $terminating$.

The termination property of TRS can be proved by the following definition and theorem.

Definition 2.1.14. A strict partial order $\succ$ on $T(\Sigma, V)$ is called a *reduction order* if it possesses the following properties.

- $closed\ under\ substitution$:

  $s \succ t$ implies $s\sigma \succ t\sigma$ for any substitution σ.

- $closed\ under\ context$:

  $s \succ t$ implies $C[s] \succ C[t]$ for any context C.

- $well\text{-}founded$:

  there exist no infinite decreasing sequences $t_1 \succ t_2 \succ t_3 \succ \cdots$.

Theorem 2.1.15. A term rewriting system $\mathcal{R}$ terminates iff there exists a reduction order $\succ$ that satisfies $l \succ r$ for all $l \rightarrow r \in \mathcal{R}$.

After termination we talk about confluence, which is also an important property often expected.

Definition 2.1.16. Two terms s, t in TRS $\mathcal{R}$ are *joinable* (notation $s \downarrow t$), if there exists a term v such that $s \rightarrow_{\mathcal{R}}^* v$ and $t \rightarrow_{\mathcal{R}}^* v$, where $\rightarrow_{\mathcal{R}}^*$ is the reflexive transitive closure of $\rightarrow_{\mathcal{R}}$.

The reflexive, symmetric, transitive closure of $\rightarrow_{\mathcal{R}}$ is denoted by $\leftrightarrow^*_{\mathcal{R}}$.

Theorem 2.1.17. A TRS $\mathcal{R}$ is *confluent* iff for all terms $s, t, u \in T(\Sigma, V)$, $u \rightarrow^*_{\mathcal{R}} s$ and $u \rightarrow^*_{\mathcal{R}} t$ implies $s \downarrow t$.

Definition 2.1.18. The composition $\sigma\tau$ of two substitutions σ and τ is defined as $s(\sigma\tau) = (s\sigma)\tau$. A substitution σ is *more general* than a substitution σ' if there exists a substitution δ such that $\sigma' = \sigma\delta$. For two terms s and t, if there is a substitution σ such that $s\sigma \equiv t\sigma$, σ is a unifier of s and t. We denote the *most general unifier* of s and t by $mgu(s, t)$.

Definition 2.1.19. Consider two rewrite rules $l_1 \rightarrow r_1$ and $l_2 \rightarrow r_2$ in a TRS R with no common variables. (If they have common variables, we can rename them properly.) If a term s is a subterm of l_1 denoted by $l_1[s]$, and if there exists an $mgu(s, l_2) = \sigma$, then the pair $\langle l_1\sigma[r_2\sigma], r_1\sigma\rangle$ of terms is called a *critical pair* of $l_1 \rightarrow r_1$ and $l_2 \rightarrow r_2$.

Example 2.1.20. Let f be a function symbol, $\{a, b, c\}$ be terms, and consider two rewrite rules $f(a) \rightarrow b$ and $a \rightarrow c$. By setting $s = a$ (the argument of $f(a)$) and $l_2 = a$ (the left-hand side of the second rule) , we have the empty mgu (or the identical mapping, meaning that no variables need to be replaced). Since $l_1[r_2] = f(c)$ and $r_1 = b$, we obtain $\langle f(c), b\rangle$ as a critical pair.

In TRS, confluence can be decided with $critical\ pairs$.

Theorem 2.1.21. A terminating TRS is *confluent* iff all critical pairs (p, q) satisfy $p \downarrow q$.

Theorem 2.1.22. If a TRS $\mathcal{R}$ satisfies termination and confluence, we say $\mathcal{R}$ is *complete* (or *convergent*) or $\mathcal{R}$ has the $completion$ property.

We fix some definitions and notations on term rewriting induction as follows.

Definition 2.1.23. The set of all *defined symbols* of $\mathcal{R}$ is defined as $D_{\mathcal{R}} = \{root(l) \mid l \rightarrow r \in \mathcal{R}\}$, where the *root symbol* of a term $s \equiv f(s_1, \ldots, s_n)$ is f, denoted by $root(s)$.

Example 2.1.24. Let a TRS $\mathcal{R} = \{f(a) \rightarrow b, g(c) \rightarrow d\}$, the defined symbols $D_{\mathcal{R}} = \{f, g\}$.

Definition 2.1.25. The function symbols other than defined symbols are *constructors*. A term consisting of only constructors and variables is a *constructor term*.

Definition 2.1.26. A term is a *basic term* if its root symbol is a defined symbol and its arguments are constructor terms. We denote all basic subterms of a term t by $\mathcal{B}(t)$.

Definition 2.1.27. A TRS $\mathcal{R}$ is *ground-reducible* (also called quasi-reducible) if every ground basic term is reducible in $\mathcal{R}$. Plaisted [32] proved that ground-reducibility is decidable.

An *equation* is expressed in the form $s \leftrightarrow t$. We do not distinguish between $s \leftrightarrow t$ and $t \leftrightarrow s$.

Theorem 2.1.28. An equation $s \leftrightarrow t$ is an *inductive theorem* of $\mathcal{R}$ if all its ground instances $s\sigma = t\sigma$ are equational consequences of the equational axioms $\mathcal{R}$ (i.e., $s\sigma \leftrightarrow^*_{\mathcal{R}} t\sigma$).

2.2 Equational Reasoning with Term Rewriting

We recall the standard automated reasoning systems on term rewriting systems.

2.2.1 Equational Completion

Given a set of equations $\mathcal{E}_0$ and a reduction order $\succ$, the standard completion procedure *KB* (or *Knuth-Bendix completion*) [27] tries to generate a convergent set $\mathcal{R}_c$ of rewrite rules that is contained in $\succ$ and that induces the same equational theory as $\mathcal{E}_0$. The KB procedure implements the following six inference rules.

DELETE: $(\mathcal{E} \cup \{s \leftrightarrow s\}; \mathcal{R}) \vdash (\mathcal{E}; \mathcal{R})$.

COMPOSE: $(\mathcal{E}; \mathcal{R} \cup \{s \to t\}) \vdash (\mathcal{E}; \mathcal{R} \cup \{s \to u\})$,
if $t \to_{\mathcal{R}} u$.

SIMPLIFY: $(\mathcal{E} \cup \{s \leftrightarrow t\}; \mathcal{R}) \vdash (\mathcal{E} \cup \{s \leftrightarrow u\}; \mathcal{R})$,
if $t \to_{\mathcal{R}} u$.

ORIENT: $(\mathcal{E} \cup \{s \leftrightarrow t\}; \mathcal{R}) \vdash (\mathcal{E}; \mathcal{R} \cup \{s \to t\})$,

if $s \succ t$.

DEDUCE: $(\mathcal{E}; \mathcal{R}) \vdash (\mathcal{E} \cup \{s \leftrightarrow t\}; \mathcal{R})$,

if $u \to_{\mathcal{R}} s$ and $u \to_{\mathcal{R}} t$.

COLLAPSE: $(\mathcal{E}; \mathcal{R} \cup \{t \to s\}) \vdash (\mathcal{E} \cup \{u \leftrightarrow s\}; \mathcal{R})$,

if $l \to r \in \mathcal{R},\ t \to_{\{l \to r\}} u$, and $t \rhd l$.

The inference rule DELETE removes a trivial equation. The inference rule ORIENT select from $\mathcal{E}$ an equation that can be oriented by the reduction order $\succ$ and adds the resultant rule to $\mathcal{R}$. The inference rule SIMPLIFY reduces an equation using $\mathcal{R}$. The inference rule COMPOSE reduces the right-hand side of a rule. The inference rule DEDUCE calculates and adds a critical pair in $\mathcal{R}$ to $\mathcal{E}$ as an equation. The inference rule COLLAPSE reduces the left-hand side of a rule and adds the results as an equation to $\mathcal{E}$, where the new symbol $\rhd$ here denotes the *encompassment order* defined as follows.

Definition 2.2.1. An *encompassment order* $\rhd$ on a set of terms is defined by $s \rhd t$ iff some subterm of s is an instance of t and $s \neq t$.

Example 2.2.2. Let $\{f, g\}$ be function symbols and $\{x, y, z\}$ be variables, then $f(x, g(x)) \rhd f(y, g(z))$ but $f(x, g(y)) \not\rhd f(z, g(z))$.

KB starts from the initial state $(\mathcal{E}_0, \mathcal{R}_0)$ where $\mathcal{R}_0 = \emptyset$. The procedure changes the states in a possibly infinite completion sequence $(\mathcal{E}_0; \mathcal{R}_0) \vdash (\mathcal{E}_1; \mathcal{R}_1) \vdash \cdots$ by its inference rules. The result of the completion sequence is the sets $\mathcal{E}_c$ and $\mathcal{R}_c$. When $\mathcal{E}_c = \emptyset$, $\mathcal{R}_c$ will be a confluent and terminating TRS satisfying $\leftrightarrow^*_{\mathcal{R}_c} = \leftrightarrow^*_{\mathcal{E}_0}$, which means KB procedure has succeeded. And the sequence has failed if $\mathcal{E}_c \neq \emptyset$.

Example 2.2.3. Let $\mathcal{E}_0$ be a set of equations representing the axioms of the group

theory as follows:

$$\mathcal{E}_0 = \left\{ \begin{array}{rcl} f(1,x) & \leftrightarrow & x, \\ f(i(x),x) & \leftrightarrow & 1, \\ f(f(x,y),z) & \leftrightarrow & f(x,f(y,z)) \end{array} \right\}$$

We obtain the following convergent TRS $\mathcal{R}_c$ equivalent to $\mathcal{E}_0$ by applying the standard completion procedure KB.

$$\mathcal{R}_c = \left\{ \begin{array}{rcl} i(1) & \rightarrow & 1, \\ i(i(x)) & \rightarrow & x, \\ f(x,1) & \rightarrow & x, \\ f(1,x) & \rightarrow & x, \\ f(x,i(x)) & \rightarrow & 1, \\ f(i(x),x) & \rightarrow & 1, \\ f(x,f(i(x),y)) & \rightarrow & y, \\ f(i(x),f(x,y)) & \rightarrow & y, \\ i(f(x,y)) & \rightarrow & f(i(y),i(x)), \\ f(f(x,y),z) & \rightarrow & f(x,f(y,z)) \end{array} \right\}$$

It is known that KB procedure may fail or diverge unless it succeeds. The result of KB seriously depends on the given reduction order. With a good order, it would lead to a success, but otherwise, it would cause the failure or the divergence. In the latter case, we could try to avoid them by changing the ordering to appropriate one, but the problem is that it is very difficult for ordinary software designers and AI researchers to design or choose an appropriate order. One may think concurrently running processes with each possible reduction order in one of the processes is a solution. However, when the number of those orders gets very large, it is clear that the direct implementation of this idea would cause serious inefficiency.

2.2.2 Rewriting Induction

The rewriting induction (RI) [1] [34] works on two sets $\langle \mathcal{E}, \mathcal{H} \rangle$, where $\mathcal{E}$ denotes the *conjectures* containing the equations to be proved, while $\mathcal{H}$ indicates the inductive *hypotheses* generated during the inferences. Given as input a ground-reducible and convergent (terminating and confluent) TRS $\mathcal{R}$, a reduction order $\succ$ covering $\mathcal{R}$, and a set $\mathcal{E}_0$ of target inductive theorems or related lemmas, the RI theorem prover starts from $\langle \mathcal{E}_0, \mathcal{H}_0 \rangle$, where $\mathcal{H}_0 = \emptyset$, and generates a derivation sequence $\langle \mathcal{E}_0, \mathcal{H}_0 \rangle \vdash \langle \mathcal{E}_1, \mathcal{H}_1 \rangle \vdash \cdots$ until it (hopefully) stops with success at $\langle \mathcal{E}_f, \mathcal{H}_f \rangle$ such that $\mathcal{E}_f = \emptyset$. The inference rules of RI are summarized as follows:

DELETE: $\langle \mathcal{E} \cup \{s \leftrightarrow s\}, \mathcal{H} \rangle \vdash \langle \mathcal{E}, \mathcal{H} \rangle$.

SIMPLIFY: $\langle \mathcal{E} \cup \{s \leftrightarrow t\}, \mathcal{H} \rangle \vdash \langle \mathcal{E} \cup \{s' \leftrightarrow t\}, \mathcal{H} \rangle$
if $s \rightarrow_{\mathcal{R} \cup \mathcal{H}} s'$.

EXPAND: $\langle \mathcal{E} \cup \{s \leftrightarrow t\}, \mathcal{H} \rangle \vdash$
$\langle \mathcal{E} \cup \mathrm{Expd}_u(s,t), \mathcal{H} \cup \{s \rightarrow t\} \rangle$
if $u \in \mathcal{B}(s)$ and $s \succ t$,

where:

$$\mathrm{Expd}_u(s,t) = \{C[r]\sigma = t\sigma \mid s \equiv C[u], l \rightarrow r \in \mathcal{R}, \sigma = mgu(u,l), l : \text{basic}\}.$$

POSTULATE: $\langle \mathcal{E}, \mathcal{H} \rangle \vdash \langle \mathcal{E} \cup \mathcal{E}', \mathcal{H} \rangle$.

The DELETE rule removes meaningless conjectures. The SIMPLIFY rule rewrites a conjecture in $\mathcal{E}$ by applying a rewrite rule taken from $\mathcal{R} \cup \mathcal{H}$.

The EXPAND rule generates conjectures and hypotheses from the current conjectures and $\mathcal{R}$. The new conjectures are generated by the function $\mathrm{Expd}_u(s,t)$, which overlaps a basic subterm u of s with each basic left-hand side of rewrite rules $l \rightarrow r$ in $\mathcal{R}$. Those conjectures will become the subgoals of the proof for the original conjec-

ture $s = t$, while the original conjecture is transformed into an inductive hypothesis $s \to t$ usable in the succeeding inferences.

The POSTULATE rule adds a set of equations $\mathcal{E}'$ to $\mathcal{E}$. Logically speaking, the equations in $\mathcal{E}'$ can be any equations, but in practice, they should be the *lemmas* that will be necessary or useful for leading the theorem prover to success. Generally, such lemmas should be added manually by highly experienced users' intuitions or generated mechanically by some heuristic algorithms.

In general, it is not straightforward to provide a suitable reduction order and to choose appropriate inference rules to be applied in the reasoning steps. Aoto [2] proposed a variant of the rewriting induction, using an arbitrary termination checker instead of a reduction order. The new system, called RIt, is defined by modifying the expand rule as follows.

EXPAND: $\langle \mathcal{E} \cup \{s = t\}, \mathcal{H} \rangle \vdash$
$\langle \mathcal{E} \cup \mathrm{Expd}_u(s,t), \mathcal{H} \cup \{s \to t\} \rangle$
if $u \in \mathcal{B}(s)$ and
$\mathcal{R} \cup \mathcal{H} \cup \{s \to t\}$ terminates.

It allows us to use more powerful termination checking techniques. However, the neccessity of approprite choice of the direction of the equation in applying the expand rule arises, because we can often orient an equation in both directions.

Chapter 3

Lemma Generation

There are two categories for lemma generation methods: *bottom-up* and *target-aimed* (top-down). The bottom-up methods generate lemmas from the given equational axioms with no consideration of the target theorem [25]. These methods have outstanding ability of generating conjectures, but their computational cost is extremely high. Meanwhile, the target-aimed methods work in a different way by considering the candidates for conjecture to generate potentially appropriate lemmas [41, 31]. Even though the generating power is limited, the computational cost is comparatively low. Thus it is desirable to strengthen the power of the target-aimed method while preserving its acceptable cost.

Target-aimed methods are classified into *sound* and *unsound* ones. The sound methods [31] generate only correct conjectures in the sense that the goal is an inductive theorem if and only if the generated conjectures are inductive theorems. These methods have a very low computational cost, but the ability of generating appropriate lemmas is extremely low. On the other hand, the unsound methods [41] try to generate useful conjectures without being restricted by the soundness. This gives higher ability of generating appropriate lemmas with a modest computational cost. We focus on the latter one to keep balance between power and cost. Basically, the unrestricted use of classic generalization techniques for the unsound methods based on replacing constants and ground terms with universally-quantified variables limits the practical usefulness. The framework based on divergence-detection

proposed by Walsh [41] greatly improves the practical usefulness by the following steps: 1) detect a potential divergence from the sequence of generated conjectures; 2) generate candidates for lemma by locating the differences between two consecutive conjectures in the diverging sequence. We put into the Walsh's framework a new heuristic lemma generation method, *peripheral sculpture*, to make the theorem prover more powerful without introducing significant cost increase.

In this chapter, we first discuss the divergence detection method and then introduce our new method.

3.1 Divergence Detection

In this section, we take simple examples for illustrating a successful proof and a divergence case in RI.

3.1.1 A Simple Successful Proof

Consider the following axioms given as a TRS $\mathcal{R}_1$ defining the binary function *append* (@ for short) on lists constructed from the $cons$ (: for short) operator, where the constant nil denotes an empty list.

$$\mathcal{R}_1 = \left\{ \begin{array}{ll} nil@xs \to xs, & (1) \\ (x : ys)@zs \to x : (ys@zs) & (2) \end{array} \right\}.$$

The target inductive theorem denoted by $\mathcal{T}_1$ is the associativity of *append*.

$$\mathcal{T}_1 : (xs@ys)@zs \leftrightarrow xs@(ys@zs). \quad (3)$$

With a reduction order $\succ$ such as the lexicographic path order based on a precedence $@ > :$, the derivation starts from $\langle \mathcal{E}_0, \mathcal{H}_0 \rangle$, where $\mathcal{E}_0$ consists of the target theorem (3) and $\mathcal{H}_0$ is the empty set. The first step of the derivation is conducted by the EXPAND

rule applied on the basic subterm $u \equiv xs@ys$ of (3) to get $\langle \mathcal{E}_1, \mathcal{H}_1 \rangle$, where $\mathcal{E}_1$ consists of the two equations

$$xs@zs1 \leftrightarrow nil@(xs@zs1),$$
$$(x : (ys@zs))@zs1 \leftrightarrow (x : ys)@(zs@zs1),$$

created by the expand function, and $\mathcal{H}_1$ consists of a single rewrite rule

$$(xs@ys)@zs \rightarrow xs@(ys@zs)$$

created by orienting the equation (3). To be more specific, the function $\text{Expd}_u(s, t)$ was invoked with $s \equiv (xs@ys)@zs$, $t \equiv xs@(ys@zs)$ and $C = \Box@zs$. The first and the second equations of $\mathcal{E}_1$ were obtained by overlapping u with the left-hand sides of (1) and (2), respectively. Then, several SIMPLIFY rules were applied to $\langle \mathcal{E}_1, \mathcal{H}_1 \rangle$ for normalization until no more applications were possible. As the result, we obtain $\langle \mathcal{E}_2, \mathcal{H}_2 \rangle$, where $\mathcal{E}_2$ consists of two equations

$$xs@zs1 \leftrightarrow xs@zs1,$$
$$x : (ys@(zs@zs1)) \leftrightarrow x : (ys@(zs@zs1)),$$

and $\mathcal{H}_2 = \mathcal{H}_1$. After that, the DELETE rule was invoked twice to remove the self-evident conjectures from $\mathcal{E}_2$ to get $\langle \mathcal{E}_3, \mathcal{H}_3 \rangle$, where $\mathcal{E}_3 = \{\}$, $\mathcal{H}_3 = \mathcal{H}_2$. Therefore, the proof succeeds, because $\mathcal{E}_3$ is empty.

3.1.2 A Divergence Case

Execution of automated RI theorem provers may *diverge* due to the accumulation of the generated "unprovable" conjectures. We illustrate such a case in the following:

Example 3.1.1.

$$\mathcal{R}_2 = \left\{ \begin{array}{ll} nil@xs \to xs, & \\ (x:ys)@zs \to x:(ys@zs), & \\ r(nil) \to nil, & (4) \\ r(x:xs) \to r(xs)@(x:nil) & (5) \end{array} \right\}.$$

Two rewrite rules (4), (5) were added to $\mathcal{R}_1$ in this problem, where the new function r defines the *reverse* operation on a list. The target inductive theorem is

$$\mathcal{T}_2 : r(r(xs)) \leftrightarrow xs.$$

The derivation starts from applying the EXPAND rule to get $\langle \mathcal{E}_1, \mathcal{H}_1 \rangle$, where

$$\mathcal{E}_1 = \left\{ \begin{array}{ll} r(nil) \leftrightarrow nil, & (6) \\ r(r(xs)@(x:nil)) \leftrightarrow x:xs & (7) \end{array} \right\}$$

and $\mathcal{H}_1 = \{r(r(xs)) \to xs\}$. Obviously, (6) is normalized by (4) to get the equation $nil \leftrightarrow nil$ to be removed by DELETE. However, since no rewrite rule of $\mathcal{R}_2 \cup \mathcal{H}_1$ can rewrite (7), it is expanded, and then the derivation goes to $\langle \mathcal{E}_2, \mathcal{H}_2 \rangle$, where

$$\mathcal{E}_2 = \left\{ \begin{array}{ll} r(nil@(x1:nil)) \leftrightarrow x1:nil, & (8) \\ r((r(xs)@(x:nil))@(x1:nil)) \leftrightarrow x1:(x:xs) & (9) \end{array} \right\},$$

$$\mathcal{H}_2 = \left\{ \begin{array}{ll} r(r(xs)) \to xs, & \\ r(r(xs)@(x:nil)) \to x:xs & (10) \end{array} \right\}.$$

Note that (7) was turned into (10), when (8) and (9) were generated by EXPAND. The good thing is that (8) will be simplified and removed after several steps of rewriting. However, the bad thing is that (9) still cannot be simplified. The derivation

continues in this way for several steps before getting to $\langle \mathcal{E}_3, \mathcal{H}_3 \rangle$, where

$$\mathcal{E}_3 = \left\{ \begin{array}{l} r((nil@(x1 : nil))@(x2 : nil)) \leftrightarrow x2 : (x1 : nil), \\ r(((r(xs)@(x : nil))@(x1 : nil))@(x2 : nil)) \leftrightarrow x2 : (x1 : (x : xs)) \end{array} \right\},$$

$$\mathcal{H}_3 = \left\{ \begin{array}{ll} r(r(xs)) \rightarrow xs, & (11) \\ r(r(xs)@(x : nil)) \rightarrow x : xs, & (12) \\ r((r(xs)@(x : nil))@(x1 : nil)) & \\ \rightarrow x1 : (x : xs) & \end{array} \right\}.$$

Clearly, there exists a regular pattern of growth in the accumulation of hypotheses. In fact, this process will continue indefinitely and generate an infinite set of hypotheses, meaning that this derivation is diverging.

It is known that in this case we can suppress the divergence by using the POSTULATE rule to provide the following equations as conjectures (becoming lemmas when proved):

$$r(xs@ys) \leftrightarrow r(ys)@r(xs), \quad (13)$$

$$(xs@ys)@zs \leftrightarrow xs@(ys@zs). \quad (14)$$

To be more specific, we can use the POSTULATE rule to put conjecture (13) into the set of equations, hoping that it may be useful for ending the divergence. However, we see that the proof of (13) itself requires a new postulation, because it will turn out that the derivation for proving (13) causes another divergence. To solve this problem, we can use the POSTULATE rule again to additionally put conjecture (14) into the set. Fortunately, conjecture (14) can be proved without any help of extra lemmas, and thus it is established as a lemma. It will turn out that lemma (14) is helpful for ending the divergence generated when trying to prove (13), and thus (13) is established as a lemma. Now lemma (13) can be used to end the divergence mentioned in the previous paragraph to finally establish the target $\mathcal{T}_2$ as a theorem. Of course, the real problem here is how we can come up with (13), (14), or any other conjectures leading our proof to success. This is the main topic of this paper.

3.1.3 Term Annotation and Difference Match

As discussed in [7] [11] [18], a crucial point in proving inductive theorems is to transform the induction conclusion to enable the use of the induction hypothesis. This can be often done by controlling the deduction so that it will remove (or "ripple out") the "difference" between the conclusion and the hypothesis. The difference is also called the "wave-front". In the following, we present some basic definitions and notations related to this subject. (Actually, we present them in a formal way suitable for the term rewriting community.)

In [7] [11], a *wave-front* is described as a term t with a proper subterm t' deleted. The deleted subterm may itself contain wave-fronts. This means that we can identify the innermost deleted subterms. A wave-front is often represented by an annotation which encloses t in a box and underlines the deleted subterm t'. (Note, however, that in the theory of the standard term rewriting, a "term" with a subterm deleted cannot be a term!) In this paper, we formally define the notion of term annotation and wave-fronts as follows.

Definition 3.1.2. Let us call the elements of $T(\Sigma, V)$ and $T(\Sigma \cup \{\Box\}, V)$ the *ordinary* terms and contexts, respectively. Let **box** and **ul** be the distinguished unary function symbols not contained in the signature Σ at hand. Then an *annotated term* is defined inductively as follows.

- If C is an ordinary context (which can be empty), $D_1, \ldots, D_n (n > 0)$ are nonempty ordinary contexts with a single hole, and $s_1, \ldots, s_n$ are either ordinary or annotated terms, then

$$C[\mathbf{box}(D_1[\mathbf{ul}(s_1)]), \ldots, \mathbf{box}(D_n[\mathbf{ul}(s_n)])],$$

  displayed with *annotations* as

$$C[\boxed{D_1[\underline{s_1}]}, \ldots, \boxed{D_n[\underline{s_n}]}],$$

is an annotated term.

Each context $D_i (i \in \{1...n\})$ is called a *wave-front*, and its hole is called a *wave-hole*.

Definition 3.1.3. A pair of annotated terms u and v, written $u \to v$, is an *annotated rule*.

Definition 3.1.4. Let w be an ordinary or annotated term. Then its *body*, $body(w)$, is defined as follows.

$$body(w) = \begin{cases} w, & \text{if} \quad w \in T(\Sigma, V), \\ C[D_1[body(s_1)], ..., D_n[body(s_n)]], & \text{if} \quad w \equiv C[\boxed{D_1[\underline{s_1}]}, ..., \boxed{D_n[\underline{s_n}]}]. \end{cases}$$

The *skeleton* of w, $skel(w)$, is defined as follows.

$$skel(w) = \begin{cases} w, & \text{if} \quad w \in T(\Sigma, V), \\ C[skel(s_1), ..., skel(s_n)], & \text{if} \quad w \equiv C[\boxed{D_1[\underline{s_1}]}, ..., \boxed{D_n[\underline{s_n}]}]. \end{cases}$$

Intuitively, $body(w)$ is an ordinary term obtained by canceling all annotations from w; $skel(w)$ is obtained by erasing all wave-fronts from the body. The wave-fronts are also regarded as the *difference* between the body and the skeleton. (In [7], the *body* function is called *erase* and its return value the body.)

For instance, consider an *annotated term*

$$w \equiv r(\boxed{\underline{r(xs)}@(x : nil)}).$$

Then its *body* and *skeleton* are as follows:

$$body(w) = r(r(xs)@(x : nil)),$$
$$skel(w) = r(r(xs)).$$

Note that in this example, $t \equiv skel(w)$ and $s \equiv body(w)$ are the left-hand side of (11) and (12), respectively. Intuitively, the symbols removed from s to get t constitute the *difference* between s and t. However, given two terms s and t, the difference between

them is not unique in general. Formally, the *difference match* function $dm(s,t)$ in Definition 3.1.5 adapted from [7] computes all the *differences* $\langle w, \delta \rangle$ such that

- $s \equiv body(w)$ (w is obtained by annotating s)
- $t \equiv skel(w)\delta$ (the annotation in w defines the difference between s and t)
- δ is a variable renaming substitution with domain $\mathcal{V}(skel(w))$.

Definition 3.1.5. Let $x, y \in V$, $s \equiv f(s_1, \ldots, s_n)$, $s' \equiv f(s'_1, \ldots, s'_n)$ and $t \equiv g(t_1, \ldots, t_m)$ where $f \neq g$.

$dm(x,y) = \{\langle x, \{x \mapsto y\}\rangle\}$

$dm(x,t) = \{\}$

$dm(s,s') = \{\langle f(w_1, \ldots, w_n), \bigcup_i \delta_i\rangle | \langle w_i, \delta_i\rangle \in dm(s_i, s'_i)$ for all $1 \leq i \leq n$
and $\delta_1 ... \delta_n$ are mutually compatible$\} \cup$
$\bigcup_i \{\langle \boxed{f(s_1, ..., s_{i-1}, \underline{w_i}, s_{i+1}, ..., s_n)}, \delta\rangle | \langle w_i, \delta\rangle \in dm(s_i, s')\}$

$dm(s,t) = \bigcup_i \{\langle \boxed{f(s_1, ..., s_{i-1}, \underline{w_i}, s_{i+1}, ..., s_n)}, \delta\rangle | \langle w_i, \delta\rangle \in dm(s_i, t)\}$

where two substitutions σ_1 and σ_2 are *compatible* if $\sigma_1(x) = \sigma_2(x)$ for every $x \in Dom(\sigma_1) \cap Dom(\sigma_2)$. For two compatible substitutions σ_1 and σ_2, the union $\sigma_1 \cup \sigma_2$ of them can be uniquely defined as the substitution σ satisfying $Dom(\sigma) = Dom(\sigma_1) \cup Dom(\sigma_2)$, $\sigma(x) = \sigma_1(x)$ for $x \in Dom(\sigma_1)$, and $\sigma(x) = \sigma_2(x)$ for $x \in Dom(\sigma_2)$.

In the sequel, each element $\langle w, \delta \rangle$ returned from the *dm* function will be simply displayed as an annotated term $w\delta$. For example, an element $\langle \boxed{x1 : \underline{xs1}}, \{xs1 \mapsto xs\}\rangle$ returned from $dm((x1 : xs1), xs)$ will be displayed as $\boxed{x1 : \underline{xs}}$.

As commented in [9], annotated terms can be considered as decorated trees where the skeleton is represented as a tree and each wave-front as a box decorating a node. The wave-fronts (boxes) often move up or down through the skeleton tree to make different annotations during difference matching. In such cases, we follow the heuristics described in [9] [41] etc. by only considering the *maximal* difference match in which wave-fronts are as high as possible in the skeleton tree.

Example 3.1.6. Let

$$s \equiv x2 : (y2 : ys2),$$

$$t \equiv x1 : xs1.$$

Then

$$dm(s,t) = \left\{ \begin{array}{l} \boxed{x2 : \underline{(x1 : xs1)}}, \\ x1 : \boxed{\underline{(xs1}} : ys2)\boxed{}, \\ x1 : \boxed{(y2 : \underline{xs1})} \end{array} \right\}.$$

Clearly, the first element is maximal, as the box is attached at the highest position (root) of the tree for the skeleton $x1 : xs1$.

The original definition of the difference matching algorithm in [7] is presented in a logic programming style where the predicate $dm(s,t,w,\delta)$ with inputs s and t supplies appropriate outputs w and δ satisfying the specified relationship among the four arguments, when it succeeds. Repeating this predicate call, one can collect all of such outputs. Our dm function is its functional version that returns those outputs as a set of $\langle w, \delta \rangle$ pairs. It was commented in [41] that using ground difference matching with renaming of variables seemed to be sufficient for identifying accumulating term structure. Therefore, we have slightly restricted the general definition of the original version according to its normal usage in the inductive theorem proving context (as is implicit in [41]). More precisely, our version restricts the substitution δ to a variable renaming substitution rather than an arbitrary substitution. Though there exists a fast polynomial algorithm for difference matching using dynamic programming [8], in this paper, we introduced the concise specification based on [7].

Note that this restricted version of the dm function is clearly related to the homeomorphic embedding [5] used in the theory of the simplification ordering, because the set $dm(s,t)$ contains an element $\langle w, \delta \rangle$ if and only if t is homeomorphically embedded in $s\delta$. The exact information on how to embed is encoded in w, an annotation to s, so that we can get t from $s\delta$ by removing all the symbol occurrences other than those in $skel(w\delta)$.

3.1.4 Automated Lemma Postulation

In [11], an effective tactic named *rippling* was proposed by Bundy et al. Walsh [41] combined this technique with difference matching [7] to overcome the difficulty faced when the induction theorem provers generate diverging sequences. Based on this technique, Shimazu, Aoto and Toyama [38] formalized an automated lemma postulation procedure in the framework of the rewriting induction as follows.

We first show a simplified version. Suppose that a sequence of hypotheses (which is seemingly diverging) contains the following rewrite rules.

$$C[s] \to t, \qquad (19)$$

$$C[D[s]] \to F[t]. \quad (20)$$

Note that the difference matching between them gives an annotation to (20) as follows.

$$C[\boxed{D[\underline{s}]}] \to \boxed{F[\underline{t}]}.$$

Then the technique for lemma postulation may be applied in the three steps as follows.

The first step applies the rewrite rule (19) to (20) in reverse (i.e., $t \to C[s]$) to get an equation

$$C[D[s]] \leftrightarrow F[C[s]].$$

Though in [38], this step is referred to as a rather general name, *modification*, in this paper, it will be called *joining*, as the resultant equation implies that $C[D[s]]$ and $F[C[s]]$ are joinable by (19) and (20) at $F[t]$.

The second step replaces occurrences of s (if it is a non-variable) with a fresh variable x to get a new conjecture:

$$C[D[x]] \leftrightarrow F[C[x]].$$

This step has been called simply *generalization* elsewhere [38]. Here we call it *non-var*

generalization to distinguish it from variable-renaming generalization introduced in the next section. Formally, an equation $s \leftrightarrow t$ is a *non-var generalization* of an equation $s\sigma = t\sigma$, if $x\sigma$ is a non-variable for all $x \in Dom(\sigma)$.

This lemma postulation method is unsound, because the generated conjectures are not necessarily inductive theorems of $\mathcal{R}$ even if the target equation is actually an inductive theorem. Hence the third step *filtering* is invoked to check if there is a real possibility that it is actually an inductive theorem. To test the equality, the system substitutes randomly-generated ground terms to the variables in the equation before normalizing its left- and right-hand sides to see their joinability. (Since $\mathcal{R}$ is convergent, this test is decidable.) The conjecture which has led to a counterexample for the equality is filtered out. On the other hand, the survived conjecture is added (by the POSTULATE inference rule) to the conjecture set $\mathcal{E}$ as a candidate for lemma to be proved later in the process.

Note that if the conjecture is directed from left to right, it works as a rewrite rule $C[D[x]] \to F[C[x]]$ for removing (or "rippling out") the difference (or "wave-front") D from the left-hand side of (20) to get a term $F[C[s]]$. Since $F[C[s]]$ can be further rewritten by (19) to $F[t]$, the hypothesis (20) may be reduced to a trivial equation $F[t] \leftrightarrow F[t]$.

This procedure is formally described as the following inference rule:

POSTULATE BY JOINING:

$$\langle \mathcal{E}, \mathcal{H} \rangle \vdash \langle \mathcal{E} \cup \{C[D[x]] \leftrightarrow F[C[x]]\}, \mathcal{H} \rangle,$$
$$\text{if } \{C[s] \to t, C[D[s]] \to F[t]\} \subseteq \mathcal{H}, \text{where } x \text{ is a fresh variable.}$$

Note that introducing a renaming substitution, the procedure in [38] is presented in a slightly more general form. Here, however, we adopted the presentation without explicit renaming, following the convention where variables in equations may be renamed whenever appropriate.

Example 3.1.7. Suppose we have the following two hypotheses annotated by difference-

matching (12) with (11):

$$r(r(xs)) \to xs,$$
$$r(\boxed{\underline{r(xs)}@(x1 : nil)}) \to \boxed{x1 : \underline{xs}}.$$

Clearly, we have $C = r(\Box)$, $D = \Box@(x1 : nil)$, $F = x1 : \Box$, $s = r(xs)$, $t = xs$, and obtain

$$r(r(xs)@(x1 : nil)) \leftrightarrow x1 : r(r(xs))$$

by joining. Then non-var generalization is applied to get a conjecture

$$r(ys@(x : nil)) \leftrightarrow x : r(ys).$$

It turns out that this conjecture is in fact a lemma that is sufficient to lead to a proof for the target theorem, $\mathcal{T}_2$, shown in Example 3.1.1.

3.2 Peripheral Sculpture

In this section, we introduce *zipped difference* to analyze the potential divergence patterns and then present a lemma postulation method called *peripheral sculpture* based on divergence detection. We introduce the basic definitions and inference rules.

3.2.1 Zipped Difference

Definition 3.2.1. Let $w \equiv C[\boxed{D_1[\underline{s_1}]}, \ldots, \boxed{D_n[\underline{s_n}]}]$ be an annotated term. Then its *peripheral part*, $peri(w)$, and *calm part*, $calm(w)$, are the unannotated contexts defined respectively as follows.

$$peri(w) = C,$$
$$calm(w) = C[D_1, ..., D_n].$$

The variables occurring in $peri(w)$ are $peripheral$.

Example 3.2.2. Consider an annotated term

$$w \equiv (y@\boxed{(x : \underline{y})})@\boxed{(\underline{x} : y)}.$$

Then y is a peripheral variable but x is not.

Definition 3.2.3. Given a (finite or infinite) sequence of terms $S = \{s_i \mid i = 0, 1, ...\}$, a *zipped difference* for S is a sequence $Z = \{w_i \mid i = 0, 1, ...\}$ of annotated terms such that

$$\forall i, \exists \delta_i : \langle w_i, \delta_i \rangle \in dm(s_{i+1}, s_i) \text{ and } \forall i, j : calm(w_i) \cong calm(w_j).$$

Recall that $s \cong t$ means s is a variant of t. An element $w_i \in Z$ consisting of the smallest number of occurrences of function symbols and variables is *minimal*. Definition 3.2.3 was inspired by Walsh [41].

Example 3.2.4. Let

$$S = \left\{ \begin{array}{l} s_0 \equiv xs, \\ s_1 \equiv x1 : xs1, \\ s_2 \equiv x2 : (y2 : ys2) \end{array} \right\}.$$

Then $dm(s_1, s_0)$ contains two differences

$$\{\boxed{x1 : \underline{xs}}, \boxed{\underline{xs} : xs1}\},$$

and $dm(s_2, s_1)$ contains the maximal difference

$$\boxed{x2 : \underline{(x1 : xs1)}}$$

and two non-maximal ones. The zipped difference is

$$Z = \{\boxed{x1 : \underline{xs1}}, \boxed{x2 : \underline{(y2 : ys2)}}\},$$

where the minimal element is $\boxed{x1 : \underline{xs1}}$. However, $\boxed{\underline{xs} : xs1}$ cannot be an element of a zipped difference, and hence only Z is the correct zipped difference.

Example 3.2.5. We apply Definition 3.2.3 to a sequence of hypotheses $\mathcal{H}$ as well, by regarding $\rightarrow$ as a binary function symbol with infix notation. Consider the following sequence $\mathcal{H}_3$ from Section 3.2 (with variable renaming):

$$\mathcal{H}_3 = \left\{ \begin{array}{l} h_0 \equiv r(r(xs)) \rightarrow xs, \\ h_1 \equiv r(r(xs1)@(x1 : nil)) \rightarrow x1 : xs1, \\ h_2 \equiv r((r(ys2)@(y2 : nil))@(x2 : nil)) \\ \quad \rightarrow x2 : (y2 : ys2) \end{array} \right\}.$$

Then $dm(h_1, h_0)$ contains two differences

$$\left\{ \begin{array}{l} r(\boxed{\underline{r(xs)}@(x1 : nil)}) \rightarrow \boxed{x1 : \underline{xs}}, \\ r(\boxed{\underline{r(xs)}@(x1 : nil)}) \rightarrow \boxed{\underline{xs} : xs1} \end{array} \right\},$$

and $dm(h_2, h_1)$ contains the maximal difference

$$r(\boxed{\underline{(r(xs1)@(x1 : nil))}@(x2 : nil)}) \rightarrow \boxed{x2 : \underline{(x1 : xs1)}}$$

and four non-maximal differences. The following is the the zipped difference:

$$Z' = \left\{ \begin{array}{l} r(\boxed{\underline{r(xs1)}@(x1 : nil)}) \rightarrow \boxed{x1 : \underline{xs1}}, \\ r(\boxed{\underline{(r(ys2)@(y2 : nil))}@(x2 : nil)}) \\ \quad \rightarrow \boxed{x2 : \underline{(y2 : ys2)}} \end{array} \right\}.$$

Intuitively, given a potentially diverging sequence S, its zipped difference postulates a divergence pattern of S, representing a common annotation pattern for the differences between successive terms. The calm parts of its elements represent a stable context and the remaining parts in the wave-holes are considered to represent a growing pattern of the divergence.

3.2.2 Lemma Postulation by Peripheral Sculpture

Given a sequence of hypotheses $\mathcal{H}$, the first two rules can be represented as follows when a zipped difference Z for $\mathcal{H}$ exists (where the variables may be renamed in the second rule.):

$$C[s_1, ..., s_n] \to C'[t_1, ..., t_m], \tag{22}$$

$$C[\boxed{D_1[\underline{s_1}]}, \ldots, \boxed{D_n[\underline{s_n}]}] \to C'[\boxed{D'_1[\underline{t_1}]}, \ldots, \boxed{D'_m[\underline{t_m}]}]. \tag{23}$$

In this subsection, we present a lemma postulation method applicable in this situation.

Definition 3.2.6. Let $\nu_1, \ldots, \nu_k$ be fresh variables, and consider a renaming substitution

$$\delta = \{x_1 \mapsto \nu_1, ..., x_k \mapsto \nu_k\},$$

with the domain

$$\{x_1, \ldots, x_k\} = \mathcal{V}(C) \cap \mathcal{V}(C') \cap [(\cup_i \mathcal{V}(D_i[s_i])) \cup (\cup_i \mathcal{V}(D'_i[t_i]))].$$

Then

$$C\delta[s_1, ..., s_n] \to C'\delta[t_1, ..., t_m]$$

is called a *peripheral sculpture* of Z.

Note that every variable in the domain should occur in every one of the three parts: C, C', and the remaining part. We rename the occurrences only in C and C' with the remaining part untouched.

Example 3.2.7. Suppose we have

$$w + (x + (x + (x + 0))) \to w + (x + (x + x)),$$
$$w + (x + (x + \boxed{s(\underline{x + 0})})) \to w + (x + (x + \boxed{s(\underline{x})})),$$

and

$$Z = \{w + (x + (x + \boxed{s(\underline{x + 0})})) \to w + (x + (x + \boxed{s(\underline{x})}))\}.$$

Then its peripheral sculpture is

$$w + (\nu + (\nu + (x + 0))) \to w + (\nu + (\nu + x)).$$

Given a sequence $\mathcal{H}$ of hypotheses, the following non-deterministic procedure tries to generate a conjecture to be proved as a lemma for the target theorem.

(i) Compute a zipped difference Z for $\mathcal{H}$.

(ii) If Z exists, compute a peripheral sculpture $\mathcal{S}$ of $\mathcal{H}$ with respect to Z.

(iii) If $\mathcal{S}$ exists, send it to the filtering process to see its possibility of being a theorem.

(iv) If $\mathcal{S}$ has survived, return $\mathcal{S}$ itself or its non-var generalization as a conjecture.

Step 2 is the key to this method. In practice, the two rules (22) and (23) are the first (minimal) and the second (second-minimal) elements of $\mathcal{H}$ and the remaining elements are just used for checking the existence of Z. If Z does not exist, the procedure is aborted. If there are more than one Z's, every Z is considered non-deterministically.

Note that the longer sequence we have for $\mathcal{H}$, the more reliable conjecture we get, because Z for a longer $\mathcal{H}$ shows us a longer diverging pattern. On the other hand, longer $\mathcal{H}$ may lead to less efficiency caused by the delay of useful lemma generation. Based on experience, the length 3 is recommended in [41].

The procedure can be formally described as an inference rule as follows:

POSTULATE BY PERIPHERAL SCULPTURE:

$$\langle \mathcal{E}, \mathcal{H} \rangle \vdash \langle \mathcal{E} \cup \{p \leftrightarrow q\}, \mathcal{H} \rangle,$$

if

$$\left\{ \begin{array}{l} C[s_1, ..., s_n] \to C'[t_1, ..., t_m], \\ C[\boxed{D_1[\underline{s_1}]} ... \boxed{D_n[\underline{s_n}]}] \to C'[\boxed{D'_1[\underline{t_1}]} ... \boxed{D'_m[\underline{t_m}]}] \end{array} \right\} \subseteq \mathcal{H},$$

- there exists a zipped difference Z for $\mathcal{H}$, and

- $p \to q$ is a peripheral sculpture of Z or its non-var generalization.

Note that the filtering process in Step 3 is not involved in the inference rule. This is because the filtering does not affect the exact form of the generated conjecture. We regard the filtering as a part of the control strategy of the theorem prover which uses it as a deciding factor for applying the postulation rule.

Example 3.2.8. Consider the following TRS $\mathcal{R}_3$.

$$\mathcal{R}_3 = \left\{ \begin{array}{c} 0 + y \to y, \\ s(x) + y \to s(x + y), \\ 0 * y \to 0, \\ s(x) * y \to y + (x * y) \end{array} \right\},$$

where $+$ and $*$ are recursively defined as the algebraic *addition* and *multiplication*. The target theorem is:

$$\mathcal{T}_3 : s(s(0)) * x \leftrightarrow x + x.$$

Starting with $\mathcal{T}_3$, the procedure diverges by constantly expanding new conjectures which cannot be simplified to a trivial equation. The difference matching procedure annotates the diverging sequence $\mathcal{H}$ as follows.

$$\begin{aligned} x + (x + 0) &\to x + x, \\ x + \boxed{s(\underline{x + 0})} &\to x + \boxed{s(\underline{x})}, \\ x + \boxed{s(\underline{s(x + 0)})} &\to x + \boxed{s(\underline{s(x)})}. \end{aligned}$$

In Step 1, we have a zipped difference

$$Z = \left\{ \begin{array}{c} x + \boxed{s(\underline{x + 0})} \to x + \boxed{s(\underline{x})}, \\ x + \boxed{s(\underline{s(x + 0)})} \to x + \boxed{s(\underline{s(x)})} \end{array} \right\}.$$

In Step 2, we have a peripheral sculpture $\mathcal{S}$:

$$\nu + (x + 0) \rightarrow \nu + x.$$

After Steps 3 and 4, we get a new conjecture:

$$\nu + (x + 0) \leftrightarrow \nu + x.$$

It will turn out that this conjecture is actually a lemma to resolve the divergence to complete the proof of $\mathcal{T}_3$.

The next example demonstrates that POSTULATE BY PERIPHERAL SCULPTURE may be applicable to more complex diverging patterns involving 'parallel' and 'nested' differences.

Example 3.2.9. With $\mathcal{R}_3$ in the previous example, our target theorem here is

$$\mathcal{T}_4 : s(s(s(s(0)))) * x \leftrightarrow s(s(0)) * (s(s(0)) * x).$$

The diverging sequence is annotated as follows.

$$\begin{aligned}
&(x + (x + 0)) + ((x + (x + 0)) + 0) \\
&\rightarrow x + (x + (x + (x + 0))), \\
&(x + \boxed{s(\underline{x + 0})}) + \boxed{s((x + \boxed{s(\underline{x + 0})}) + 0)} \\
&\rightarrow x + \boxed{s(x + \boxed{s(x + \boxed{s(\underline{x + 0})})})}, \\
&(x + \boxed{s(\underline{s(x + 0)})}) + \boxed{s(s((x + \boxed{s(\underline{s(x + 0)})}) + 0))} \\
&\rightarrow x + \boxed{s(s(x + \boxed{s(s(x + \boxed{s(\underline{s(x + 0)})}))}))}.
\end{aligned}$$

In Step 1, we have a zipped difference $Z =$

$$\left\{\begin{array}{l} (x + \boxed{s(\underline{x + 0})}) + \boxed{s((x + \boxed{s(\underline{x + 0})}) + 0)} \\ \quad \to x + \boxed{s(x + \boxed{s(x + \boxed{s(\underline{x + 0})})})}, \\ (x + \boxed{s(\underline{s(x + 0)})}) + \boxed{s(s((x + \boxed{s(\underline{s(x + 0)})}) + 0))} \\ \quad \to x + \boxed{s(s(x + \boxed{s(s(x + \boxed{s(\underline{s(x + 0)})}))}))} \end{array}\right\}.$$

In Step 2, we have a peripheral sculpture $\mathcal{S}$:

$$(\nu + (x + 0)) + ((x + (x + 0)) + 0) \to \nu + (x + (x + (x + 0))).$$

After Step 3, we have three subterms

$$\{x + (x + 0), x + 0, 0\}$$

for non-var generalization procedure. By generalizing them, a conjecture (corresponding to $x + (x + 0)$) and an equivalent of $\mathcal{S}$ passed the random testing, where the former one

$$(\nu + (x + 0)) + (y + 0) \leftrightarrow \nu + (x + y)$$

leads to a successful proof. Note that this conjecture is a consequence of the right identity and the associativity of $+$, but we are not given the corresponding axioms or lemmas.

Chapter 4

Multi-Context Reasoning System

There are several problems of KB or RI/RIt in practice. Firstly, the problem of KB is that it is difficult to set an appropriate reduction order before the procedure starts. Secondly, the problem of RI/RIt is that the nondeterministic choices may cause the procedures to end up with failing results (i.e., divergence or fail). Thirdly, appropriate lemmas are often required in practice for RI/RIt to achieve successful proofs.

The first problem can be partially solved by the multi-context reasoning system MKB which simulates the mutually related processes in a single process efficiently. As for the second and third problems, MRIt pursues all nondeterministic choices trying to lead the procedure to a successful result by exploiting the schema of MKB. In this chapter we expand our method into multi-context schema to develop a practically useful framework for inductive theorem provers.

We discuss MKB and MRIt in more detail in this chapter.

4.1 Multi-Context Completion System

A completion procedure for multiple reduction orderings called MKB developed in [28] accepts a finite set of reduction orderings $O = \{\succ_1, \ldots, \succ_n\}$ and a set of equations $\mathcal{E}_0$ as input. The proper output is a set of convergent rewrite rules $\mathcal{R}_c$. To achieve the multi-completion, MKB effectively simulates KB procedures in n parallel

processes $\{P_1, \ldots, P_n\}$ corresponding to O. Let $I = \{1, \ldots, n\}$ be the index set and $i \in I$ be an index. In this setting, P_i executes KB for the reduction order $\succ_i$ and the common input $\mathcal{E}_0$. The inference rules of MKB which simulate the related KB inferences all in a single operation are based on a special data structure called the $node$ defined below.

Definition 4.1.1. A *node* is a tuple $\langle s : t, R_0, R_1, E \rangle$, where $s : t$ is an ordered pair of terms s and t each called $datum$, and R_0, R_1, E are subsets of I called $labels$ such that:

- R_0, R_1 and E are mutually disjoint. (i.e., $R_0 \cap R_1 = R_0 \cap E = R_1 \cap E = \emptyset$)
- $i \in R_0$ implies $s \succ_i t$, and $i \in R_1$ implies $t \succ_i s$

Intuitively, the set $R_0(R_1)$ represents the indices of processes executing KB in which the set of rewrite rules $\mathcal{R}$ currently contains $s \to t$ ($t \to s$), and E represents those of processes in which $\mathcal{E}$ contains an equation $s \leftrightarrow t$ (or $t \leftrightarrow s$). The node $\langle s : t, R_0, R_1, E \rangle$ is considered to be identical with the node $\langle t : s, R_1, R_0, E \rangle$, so the inference rules of MKB working on a set N of nodes defined below implicitly specify the symmetric cases.

DELETE: $N \cup \{\langle s : s, \emptyset, \emptyset, E \rangle\} \vdash N$, if $E \neq \emptyset$.

ORIENT: $N \cup \{\langle s : t, R_0, R_1, E \cup E' \rangle\} \vdash$
$N \cup \{\langle s : t, R_0 \cup E', R_1, E \rangle\}$,
if $E' \neq \emptyset, E \cap E' = \emptyset$, and $s \succ_i t$ for all $i \in E'$.

REWRITE_1: $N \cup \{\langle s : t, R_0, R_1, E \rangle\} \vdash$
$$N \cup \left\{ \begin{array}{l} \langle s : t, R_0 \backslash R, R_1, E \backslash R \rangle \\ \\ \langle s : u, R_0 \cap R, \emptyset, E \cap R \rangle \end{array} \right\},$$
if $\langle l : r, R, \ldots, \ldots \rangle \in N,\ t \to_{\{l \to r\}} u$,
$t \cong l$, and $(R_0 \cup E) \cap R \neq \emptyset$.

REWRITE_2: $N \cup \{\langle s : t, R_0, R_1, E\rangle\} \vdash N \cup$
$$\left\{ \begin{array}{l} \langle s : t, R_0 \backslash R, R_1 \backslash R, E \backslash R\rangle \\ \\ \langle s : u, R_0 \cap R, \emptyset, (R_1 \cup E) \cap R\rangle \end{array} \right\},$$
if $\langle l : r, R, \ldots, \ldots\rangle \in N,\ t \rightarrow_{\{l \rightarrow r\}} u,$
$t \rhd l$, and $(R_0 \cup R_1 \cup E) \cap R \neq \emptyset$.

DEDUCE: $N \vdash N \cup \{\langle s : t, \emptyset, \emptyset, R \cap R'\rangle\}$,
if $\langle l : r, R, \ldots, \ldots\rangle \in N$,
$\langle l' : r', R', \ldots, \ldots\rangle \in N, R \cap R' \neq \emptyset$,
and $s \leftarrow_{\{l \rightarrow r\}} u \rightarrow_{\{l' \rightarrow r'\}} t$.

GC: $N \cup \{\langle s : t, \emptyset, \emptyset, \emptyset\rangle\} \vdash N$.

SUBSUME: $N \cup \left\{ \begin{array}{l} \langle s : t, R_0, R_1, E\rangle \\ \\ \langle s' : t', R'_0, R'_1, E'\rangle \end{array} \right\} \vdash$
$N \cup \{\langle s : t, R_0 \cup R'_0, R_1 \cup R'_1, E''\rangle\}$,
if $s : t$ and $s' : t'$ are variants and
$E'' = (E \backslash (R'_0 \cup R'_1)) \cup (E' \backslash (R_0 \cup R_1))$.

Given the current set N of nodes, $\langle E[N, i], R[N, i]\rangle$ defined in the following represents the current set of equations and rewrite rules in a process P_i.

Definition 4.1.2. Let $n = \langle s : t, R_0, R_1, E\rangle$ be a node and $i \in I$ be an index. The *$\mathcal{E}$-projection* $\mathcal{E}[n, i]$ of n onto i is a (singleton or empty) set of equations defined by

$$\mathcal{E}[n, i] = \left\{ \begin{array}{ll} \{s \leftrightarrow t\}, & \text{if } i \in E, \\ \\ \emptyset, & \text{otherwise.} \end{array} \right.$$

Similarly, the *$\mathcal{R}$-projection* $\mathcal{R}[n, i]$ of n onto i is a set of rules defined by

$$\mathcal{R}[n, i] = \left\{ \begin{array}{ll} \{s \rightarrow t\}, & \text{if } i \in R_0, \\ \\ \{t \rightarrow s\}, & \text{if } i \in R_1, \\ \\ \emptyset, & \text{otherwise.} \end{array} \right.$$

These notions can also be extended for a set N of nodes as follows:

$$\mathcal{E}[N,i] = \bigcup_{n \in N} \mathcal{E}[n,i], \quad \mathcal{R}[N,i] = \bigcup_{n \in N} \mathcal{R}[n,i]$$

MKB starts with the initial set N_0 of nodes:

$$N_0 = \{\langle s : t, \emptyset, \emptyset, I\rangle \mid s \leftrightarrow t \in \mathcal{E}_0\},$$

which means, given the initial set of equations $\mathcal{E}_0$, we have $\langle \mathcal{E}[N_0,i], \mathcal{R}[N_0,i]\rangle = \langle \mathcal{E}_0, \emptyset\rangle$ for all $i \in I$. The state sequence of MKB is generated as $N_0 \vdash N_1 \vdash \cdots \vdash N_c$. If $\mathcal{E}[N_c,i]$ is empty and all critical pairs of $\mathcal{R}[N_c,i]$ have been created, MKB returns $\mathcal{R}[N_c,i]$ as the result, which is a convergent TRS obtained by a successful KB sequence in the process P_i.

4.2 Multi-Context Rewriting Induction System

Based on the ideas of *multi-completion* (MKB) [28] [36] [37] and *rewriting induction with termination checkers* (RIt) [2], the *multi-context rewriting induction* (MRIt) [22] [35] efficiently simulates RIt processes, each corresponding to a non-deterministic computation which has made a particular series of commitments at the choice points they encountered for various decisions.

In particular, reduction orders (in which way to orient an equation; implicitly inducing induction patterns based on Noetherian induction) can be selected dynamically by calling external, modern automated termination checkers more powerful than the classical, simply parameterized reduction orders (such as recursive path orders and polynomial orders), based on the work of [42]. Other choices include induction strategies (which variable to select for induction) and rewriting strategies (which rule to apply and which subterm to be applied to).

To distinguish processes, each process is represented by a sequence of natural numbers $a_1 a_2 \ldots a_k$ (for some $k \geq 0$) called an *index*, when the i-th decision of this process was the choice No. a_i ($1 \leq i \leq k$). Thus the index can be interpreted as a position of a node at depth k in a search tree. In particular, the initial process (the root node) is represented by an empty sequence (denoted by ϵ). We do not distinguish between a process and its index.

In order to efficiently simulate a lot of closely-related inferences made in different processes, MRIt exploits the data structure called *nodes* and represent the state of the inference system by a set of nodes. The node is a tuple $\langle s : t, H_1, H_2, E \rangle$, where $s : t$ is an ordered pair of terms s and t, and H_1, H_2, E are subsets of process indices called $labels$. Intuitively, E represents all processes containing $s \leftrightarrow t$ as a conjecture, and H_1 (resp. H_2) represents all processes containing $s \rightarrow t$ (resp. $t \rightarrow s$) as an inductive hypothesis. The set of possible indices I is infinite in MRIt. The node $\langle s : t, H_1, H_2, E \rangle$ is considered to be identical with $\langle t : s, H_2, H_1, E \rangle$.

Given the current set N of nodes, $\mathcal{E}[N, p]$ and $\mathcal{H}[N, p]$ defined below represent the current sets of conjectures (equations) and hypotheses (rules), respectively, held in the process p.

$$\mathcal{E}[N, p] = \bigcup_{n \in N} \mathcal{E}[n, p], \quad \mathcal{H}[N, p] = \bigcup_{n \in N} \mathcal{H}[n, p].$$

$$\mathcal{E}[n, p] = \begin{cases} \{s \leftrightarrow t\}, & \text{if } p \in E, \\ \emptyset, & \text{otherwise.} \end{cases}$$

$$\mathcal{H}[n, p] = \begin{cases} \{s \rightarrow t\}, & \text{if } p \in H_1, \\ \{t \rightarrow s\}, & \text{if } p \in H_2, \\ \emptyset, & \text{otherwise.} \end{cases}$$

where $n = \langle s : t, H_1, H_2, E \rangle$. $\mathcal{E}[n, p]$ and $\mathcal{H}[n, p]$ are called *$\mathcal{E}$-projection* and *$\mathcal{H}$-projection* of n onto p, respectively.

Given the initial set of conjectures $\mathcal{E}_0$ and a ground-reducible and convergent

TRS $\mathcal{R}$, MRIt starts with the initial set N_0 of nodes:

$$N_0 = \{\langle s : t, \emptyset, \emptyset, \{\epsilon\}\rangle \mid s \leftrightarrow t \in \mathcal{E}_0\}.$$

Note that $\langle \mathcal{E}[N_0, \epsilon], \mathcal{H}[N_0, \epsilon]\rangle = \langle \mathcal{E}_0, \emptyset\rangle$. The inference rules of MRIt are listed in Appendix A. A series of applications of those rules to the sets of nodes generates a derivation $N_0 \vdash N_1 \vdash \cdots \vdash N_c$. If $\mathcal{E}[N_c, p]$ is empty for some process p, the system concludes that all the initial conjectures $\mathcal{E}_0$ are inductive theorems of $\mathcal{R}$.

We elaborate on inference rules DELETE, FORK and EXPAND of MRIt to show how the multi-context reasoning works.

The DELETE rule of MRIt simulates its counterpart of RIt. That is, if a trivial conjecture appears in any process, it is removed.

DELETE: $N \cup \{\langle s : s, H_1, H_2, E\rangle\} \vdash N$.

Assume that process p_1 holds a trivial conjecture $s \leftrightarrow s$ in the corresponding $\mathcal{E}_1$ and that process p_2 also holds $s \leftrightarrow s$ in the corresponding $\mathcal{E}_2$. To remove the trivial conjectures, the DELETE of RIt is invoked in each process, p_1 and p_2. In MRIt, such manipulations are effectively done by simply removing one node $\langle s : s, H_1, H_2, \{p_1, p_2\}\rangle$ from the node set. Note that H_1, H_2 can be empty in the sense that the deletion happens in $\mathcal{E}$ in RIt.

Suppose that the process with an index $p = a_1 a_2 \ldots a_k$ has n possible choices. Then it will be forked into n different processes $p\,1, p\,2, \ldots, p\,n$, each taking care of one of the choices. In [35] this operation is simulated in the node structure by replacing the index p in every label of every node with those new n indices, and formalized as the FORK inference rule:

FORK: $N \vdash \psi_P(N)$.

To formally understand this inference rule, we need the following definitions and notations introduced in [35]. The basic *fork function over* a given set P of processes, denoted by $\psi_P : I \to \mathbb{P}(I)$, is defined as follows:

Definition 4.2.1.

$$\psi_P(p) = \begin{cases} \{p1, p2, \ldots, p\psi(p)\}, & \text{if } p \in P, \\ \{p\}, & \text{otherwise.} \end{cases}$$

where I is the set of all indices (processes), $\mathbb{P}(I)$ the power set of I, and $\psi(p)$ the number of processes that p is to be forked into.

The notation $\psi_P(N)$ used in FORK represents the set of nodes created from N by replacing all the processes p in P with $p\,1, p\,2, \ldots, p\,\psi(p)$.

The EXPAND rule of MRIt is the counterpart of that of RIt, playing the leading role in rewriting induction.

EXPAND: $N \cup \{\langle s : t, H_1, H_2, E \cup E' \rangle\} \vdash$
$N \cup \{\langle s : t, H_1 \cup E', H_2, E \rangle\}$
$\cup \{\langle s' : t', \emptyset, \emptyset, E' \rangle \mid s' \leftrightarrow t' \in \text{Expd}_u(s, t)\}$
if $E' \neq \emptyset, u \in \mathcal{B}(s)$ and
$\mathcal{H}[N, p] \cup \mathcal{R} \cup \{s \to t\}$ terminates for all $p \in E'$.

Focusing on a node $n = \langle s : t, H_1, H_2, E \cup E' \rangle$ and a basic term $u \in \mathcal{B}(s)$, this inference rule applies the EXPAND rule of RIt to all processes (E') that can orient $s \leftrightarrow t$ from left to right. As a result, new nodes $\langle s' : t', \emptyset, \emptyset, E' \rangle$ are created for all conjectures $s' \leftrightarrow t'$ generated by $\text{Expd}_u(s, t)$. The labels of the original node n is modified so that the processes of E' moves from the third to the first label, meaning that in those processes the equation $s \leftrightarrow t$ was oriented from left to right. Note that the choice of the direction of orientation and the choice of the basic subterm are two kinds of non-deterministic choices. In practice, therefore, the theorem prover should combine EXPAND operation with two types of FORK operations: One is to fork a process into two processes depending on the two possible direction of orientation, and another is to fork each of the resultant processes with k basic subterm choices into k processes. The formal treatment of this combination is presented in [35].

Let $\vdash_{RIt}^{=}$ be the reflexive closure of $\vdash_{RIt}$ (meaning $\vdash_{RIt}^{=}$ denotes either $=$ or $\vdash_{RIt}$). The following two propositions shown in [35] state the soundness of FORK and other inference rule of MRIt other than FORK.

Proposition 4.2.2. Let $N' = \psi_P(N)$ be the set of nodes obtained by applying FORK to N. Then $\langle \mathcal{E}[N,p], \mathcal{H}[N,p] \rangle = \langle \mathcal{E}[N',q], \mathcal{H}[N',q] \rangle$ for all $p \in I$ and $q \in \psi_P(p)$.
In other words, FORK has no effect on processes, only generating copies of some processes.

Proposition 4.2.3. Let N' be the set of nodes obtained by applying to N an inference rule of MRIt other than FORK. Then $\langle \mathcal{E}[N,p], \mathcal{H}[N,p] \rangle \vdash_{RIt}^{=} \langle \mathcal{E}[N',p], \mathcal{H}[N',p] \rangle$ for all $p \in I$.
In other words, the inference rules of MRIt other than FORK simulate an RIt inference in some processes and have no effect on other processes.

4.3 Postulation in Multi-Context System

In this section, we will extend MRIt to develop an inductive theorem prover MRIt+ which combines 1) multi-context reasoning and 2) RIt with divergence-detection-based automated lemma postulation, where we replace the general POSTULATE rule of RIt with more specific inference rules for postulation discussed in Chapter 3: POSTULATE BY JOINING and POSTULATE BY PERIPHERAL SCULPTURE.

Suppose we have several processes $P = \{p_1, p_2, ...\}$ and a potentially diverging sequence $\{h_1 \equiv s_1 \to t_1, h_2 \equiv s_2 \to t_2, ...\}$ held by $p \in P$. Also suppose that there are k conjectures $\{l_i \leftrightarrow r_i\}$ that can be added to $\langle \mathcal{E}, \mathcal{H} \rangle$ in p by applying either POSTULATE BY JOINING or POSTULATE BY PERIPHERAL SCULPTURE. To deal with these conjectures, we have process p fork into $k+1$ processes where each process $p\,i (1 \leq i \leq k)$ holds one new conjecture $l_i \leftrightarrow r_i$, respectively, and the remaining process $p\,(k+1)$ continues without any of the newly generated conjectures. Such a process without postulation is necessary because divergence detection is "unsound",

meaning that the postulated conjecture may be incorrect, causing a search in vain for its proof.

Note that the same inference (postulation) discussed in the previous paragraph can be made in other processes as well containing $s_1 \to t_1$ and $s_2 \to t_2$ as hypotheses. Actually, this can be handled efficiently by the standard technique of multi-context equational reasoning as in the following inference rule.

MULTI-CONTEXT POSTULATE

$$
\begin{aligned}
&N \vdash \psi_P(N) \cup \{\langle l_i : r_i, \emptyset, \emptyset, P_i\rangle \mid 1 \le i \le k\} \\
&\text{if } \langle \mathcal{E}, \mathcal{H}\rangle \vdash_{RIt} \langle \mathcal{E} \cup \{l_i \leftrightarrow r_i\}, \mathcal{H}\rangle \\
&\qquad \text{for some process } p \in P (1 \le i \le k) \\
&\text{where } n_1 = \langle s_1 : t_1, H_1, ..., ...\rangle \in N, \\
&\qquad n_2 = \langle s_2 : t_2, H_2, ..., ...\rangle \in N, \\
&\qquad P = H_1 \cap H_2, \\
&\qquad P_i = \{p.i \mid p \in P\}, \\
&\qquad \mathcal{E} = \mathcal{E}[N, p], \\
&\qquad \mathcal{H} = \mathcal{H}[N, p], \\
&\qquad \psi(p) = k + 1 \text{ for all } p \in P, \\
&\text{and } \vdash_{RIt} \text{ denotes one-step derivation in RIt.}
\end{aligned}
$$

Since we have added MULTI-CONTEXT POSTULATE to MRIt, we need to augment these results with the following proposition. (The easy proof is omitted.)

Proposition 4.3.1. Let N' be the set of nodes obtained by applying MULTI-CONTEXT POSTULATE to N. Then $\langle \mathcal{E}[N, p], \mathcal{H}[N, p]\rangle \vdash^{=}_{RIt} \langle \mathcal{E}[N', q], \mathcal{H}[N', q]\rangle$ for all $p \in I$ and $q \in \psi_P(p)$.

In other words, MULTI-CONTEXT POSTULATE simulates an RIt inference (in fact, POSTULATE either BY JOINING or BY PERIPHERAL SCULPTURE) in some processes and

has no effect on other processes.

Chapter 5

Implementation

In this chapter, we discuss the implementation of multi-context algebraic reasoning systems: multi-context completion system (*lz-mkb*) [23] [24], multi-context inductive theorem prover (*lz-itp*) [22] and the multi-context inductive theorem prover with automated lemma generation (MRIt+) [21].

The implementations are made in *Scala*. Scala is a programming language which supports *functional programming* and *object-oriented programming*. The programs of *lz-mkb* and *lz-itp* were designed in an object-oriented way so that we could build and reuse the classes to organize the term structures, substitutions, nodes, inference rules, etc. At the same time, we followed the discipline of functional programming (e.g., "uniform return type" principle [30]) in coding so that it could be safer and easier to execute the program in a physically parallel computational environment. We also enabled *lazy evaluation* mechanism of Scala to improve the performance when the systems face large problems.

5.1 Implementation of Completion System

The node, a basic unit of MKB, is implemented as a class which contains an equation object as a datum and three $bitsets$ as labels. We chose bitset[(1)] to gain efficiency because there were numerous set operations during the computation. We

[(1)] a data structure defined in Scala's library

also created a class called *nodes* for the set N of nodes for which several inference rules of MKB are defined. We will discuss the implemented operations below in comparison with the original inference rules of MKB one by one.

The operation $N.subsume()$ combines two nodes into a single one when they contain the variant data (which are the same as each other up to renaming of variables). The duplicate indices in the third labels are removed to preserve the label conditions. We exploited a programming technique called *lazy evaluation* to gain efficiency in the implementation. To discuss the details, we consider the pseudocode of implementation presented as *Algorithm 5.1*, based on the presentation in [23]. The operation $N.subsume()$ is invoked by the operation $union(N, N')$ which is designed for combining nodes N and N'. We observe that in every iteration of the *while loop*, the $union(N, N')$ operation is called at least once (i.e., for every chosen n, *subsume()* would be called at *line 9* once; And for those that satisfied the proper conditions of *line11* and *line 13*, two more operations are required). This means *subsume()* would be invoked frequently during the whole procedure. It would make the program slower to simply check all of the nodes in N, when N was updated after rewrite operations. To gain efficiency, we created a $lazy$ hash map $[\mathfrak{I}_s, \mathfrak{N}]$, where $\mathfrak{N}$ is a $list$ of nodes and $\mathfrak{I}_s$ is a lazy value defined in the node class as the $size$ of the node (i.e., for a node $n = \langle s : t, r_0, r_1, e \rangle$, $n.\text{size} = s.\text{size} + t.\text{size}$), so that we need only check the nodes with the same size as the original nodes. This check can be done efficiently by using the hash map with the node size as its key. In other words, for every $n \in N$, n uses its size $\mathfrak{I}_n$ as the key to $[\mathfrak{I}_s, \mathfrak{N}]$, then the set $\mathfrak{N}_n$ containing all the nodes with same size $\mathfrak{I}_n$ is looked up for the nodes with variant data. In our Scala program, the hash map $[\mathfrak{I}_s, N]$ is declared to be *lazy*, because it is calculated only once and then be stored as a constant object ready to be returned for repeated calculation requests afterwards.

The operation *N.delete()* simply removes from N all nodes that contain a trivial equation, and returns the remaining nodes as N'. This operation is only applied to the nodes created by rules REWRITE and DEDUCE.

Algorithm 5.1 lz-mkb(E, O)

1: $N_o := \{\langle s : t, \emptyset, \emptyset, I\rangle \mid s \leftrightarrow t \in E\}$ where $I = \{1, \ldots, |O|\}$
2: $N_c := \emptyset$
3: **while** $success(N_o, N_c) = false$ **do**
4: **if** $N_o = \emptyset$ **then**
5: return(fail)
6: **else**
7: $n := N_o.choose()$
8: $\langle \mathcal{D}, \mathcal{N}, \mathcal{M}\rangle := rewrite(\{n\}, N_c)$
9: $N_o := union(N_o - \{n\}, \mathcal{N}.delete())$
10: $n := \mathcal{M}.head$
11: **if** $n \neq \langle \ldots, \emptyset, \emptyset, \emptyset\rangle$ **then**
12: $n := n.orient()$
13: **if** $n \neq \langle \ldots, \emptyset, \emptyset, \ldots\rangle$ **then**
14: $\langle \mathcal{D}, \mathcal{N}, \mathcal{M}\rangle := rewrite(N_c, \{n\})$
15: $N_o := union(N_o, \mathcal{N}.delete())$
16: $N_c := N_c + \mathcal{M} - \mathcal{D}$
17: $N_c := N_c.garbagecollect()$
18: $N_o := union(N_o, deduce(n, N_c).delete())$
19: **end if**
20: $N_c := union(N_c, \{n\})$
21: **end if**
22: **end if**
23: **end while**
24: return $\mathcal{R}[N_c, i]$ where $i = success(N_o, N_c)$

The operation *n.orient()* orients the equation from left to right or right to left by changing their labels from E to R_0 or E to R_1 according to the reduction order in each process. Notice that the application of the reduction order to an equation should be done twice (i.e., one with $s : t$ and one with $t : s$) in theory, but in practice we implemented it so that it was executed only once, noting that at most one of them should be true. The indices still remaining after this operation in E correspond to the reduction orders that failed to orient the equation.

The operation *rewrite(N, N')* is not included in the class of nodes but it takes nodes as arguments. In the original idea of MKB, REWRITE_1 and REWRITE_2, they simulate the COMPOSE, SIMPLIFY and COLLAPSE (if appropriate conditions are satisfied) in one single operation. More exactly, REWRITE_1 and REWRITE_2 are repeatedly applied to $N \cup N'$, rewriting the data of N by the rules of N' until no more rewriting is possible. It returns the set of nodes created in this process and the mutation operations are applied to N so that N is updated as

$$N := N - \{\text{original nodes}\} \cup \{\text{updated nodes}\}.$$

In our implementation, we follow the discipline of functional programming by never mutating the nodes. We just update them from outside. This means the method needs to return the intermediate results as fresh sets of nodes. The result is structured as a tuple $\langle \mathcal{D}, \mathcal{N}, \mathcal{M} \rangle$ where:

$\mathcal{D}$**:** the nodes rewritten by rewrite(N, N')(i.e., the original ones with the original datum $s : t$)

$\mathcal{N}$**:** the nodes "created" by rewrite(N, N')(i.e., the new nodes with the original datum $s : t$ and updated labels)

$\mathcal{M}$**:** the nodes "modified" during rewrite(N, N')(i.e., the new nodes with a new datum $s : u$ and updated labels)

Normally, after the *rewrite(N, N')* operation, N should be updated as $N := N + \mathcal{M} - \mathcal{D}$.

If N only has one node in it (i.e., $N = \{n\}$), the modified n would be returned by $\mathcal{M}.head$.

Notice that to the symmetric cases of nodes, we just use the *mirrors* which refer to the symmetric nodes of the original N and N' as input. In other words, in every one-step rewrite, we need to do this operation four times with different combinations from $\{(N, N'), (N.mir, N'.mir), (N.mir, N'), (N, N'.mir)\}$ one by one. Surely (N, N') is updated after every single rewrite_1 or rewrite_2. In this way, we obtain a tuple $\langle \mathcal{D}_\infty, \mathcal{N}_\infty, \mathcal{M}_\infty \rangle$ of three nodes in which every calculated node is included and no more rewrite can be applied. Finally, the tuple $\langle \mathcal{D}_\infty, \mathcal{N}_\infty - \mathcal{D}_\infty, \mathcal{M}_\infty - \mathcal{D}_\infty \rangle$ is returned as the result of the operation rewrite(N, N').

The operation $N.deduce(n)$ generates all the possible critical pairs between n and $\{n\} \cup N$. We consider all combinations of pair of nodes. For example, consider two nodes $n = \langle a : b, R_0, R_1, \dots \rangle$ and $n' = \langle c : d, R'_0, R'_1, \dots \rangle$. The operation $\{n\}.deduce(n')$ considers the critical pairs from $\{a \leftrightarrow b, c \leftrightarrow d\}$, which means the modification of labels should be considered for each of $\{R_0 \cap R'_0, R_0 \cap R'_1, R_1 \cap R'_0, R_1 \cap R'_1\}$.

The operation $N.garbagecollect()$ has no related inference rules in KB. In MKB, it can effectively reduce the size of the current node database by removing nodes with three empty labels, because no processes contain the corresponding rule or equation.

Notice that the procedure $success(N_o, N_c)$ checks if this completion process has succeeded. The process succeeds if there exists an index $i \in I$ such that i is not contained in any labels of N_o and any E labels of N_c nodes. Then $\mathcal{E}[N_o \cup N_c, i] = \emptyset$, and $\mathcal{R}[N_c, i]$ is a convergent set of rewrite rules contained in $\succ_i$. We also created lazy values in nodes to hold the occurrences of the index i in the labels, so that we do not need to calculate it in the unchanged N_c every time. This also makes the computation efficient as $N.choose()$ operation will always choose the minimal node in terms of its size.

5.2 Implementation of Inductive Theorem Prover

The node is also a basic unit of MRIt. It is implemented as a class which contains an equation object as a datum and three sets as labels. We also created a class called $nodes$ for the set N of nodes for which several inference rules of MRIt are defined. The $index$ of MRIt which corresponds to a process running RI procedure is implemented as a class containing a sequence of natural numbers holding a lazy hash code to gain efficiency during the numerous index comparisons.

The rule FORK is the key to cover all parallel processes running in different states. Note that due to the nondeterministic choices of contexts (e.g., in which direction to orient, which subterm to be expanded or which rewrite strategy should be applied), we cannot decide the number of processes and strategies statically. Therefore, we do not fix the number of processes in the new procedure, and allow it to dynamically change. For example, if a process with the index

$$p = [a_1 a_2 \dots a_k]$$

have n possible choices of contexts, we have it forked into n processes as:

$$[a_1 a_2 \dots a_k 1], [a_1 a_2 \dots a_k 2], \dots, [a_1 a_2 \dots a_k n].$$

Based on the label representation, we can simulate the fork operation by replacing the label p in the labels of all nodes with the set of n identifiers $p_1, \dots, p_n$. In practice, we embed this fork operation into other operations if necessary.

The operation $lemmaExplore(N, n)$ is a newly introduced operation of MRIt. In rewriting induction, it is well-known that it is effective for proving some problems by supplying appropriate lemmas. This is where we can embed our new method *peripheral sculpture* with any other lemma generation methods. We consider with the pseudocode of our implementation in *Algorithm 5.2*. The procedure is based on the open/closed (set-of-support/have-been-given) lists algorithm, which is well-

Algorithm 5.2 lz-itp($\mathcal{E}, \mathcal{R}$)

1: $N_o := \{\langle s : t, \emptyset, \emptyset, \{\epsilon\}\rangle \mid s \leftrightarrow t \in \mathcal{E}\}$
2: $N_c := \emptyset$
3: **while** $success(N_o, N_c) = false$ **do**
4: **if** $N_o = \emptyset$ **then**
5: return(fail)
6: **else**
7: $n := N_o.choose()$
8: $\langle \mathcal{F}_1, \mathcal{F}_2, \mathcal{M}, \mathcal{N}\rangle := simplify(N_o, N_c, n)$
9: $N_o := union(N_o - \{n\}, \mathcal{N}.delete())$
10: $N_o := \mathcal{F}_1$
11: $N_c := \mathcal{F}_2$
12: $n := \mathcal{M}.head$
13: **if** $n \neq \langle \ldots, \emptyset, \emptyset, \emptyset\rangle$ **then**
14: **if** $n \neq \langle \ldots, \emptyset, \emptyset, \ldots\rangle$ **then**
15: $\langle \mathcal{F}_1, \mathcal{F}_2, \mathcal{M}, \mathcal{N}\rangle := simplify(N_o, N_c, n.mir)$
16: $N_o := union(N_o, \mathcal{N}.delete())$
17: $N_o := \mathcal{F}_1$
18: $N_c := \mathcal{F}_2$
19: $n := \mathcal{M}.head$
20: $\langle \mathcal{F}_1, \mathcal{F}_2, \mathcal{M}, \mathcal{C}\rangle := expand(N_o, N_c, n)$
21: $N_o := union(N_o, \mathcal{C}.delete())$
22: $N_o := \mathcal{F}_1.subsume\_P()$
23: $N_c := \mathcal{F}_2.subsume\_P()$
24: $n := \mathcal{M}.head$
25: **end if**
26: $n := n.subsume\_P()$
27: $N_c := union(N_c, \{n\}).gc()$
28: $N_o := union(N_o, lemmaExplore(N_c, n))$
29: **end if**
30: **end if**
31: **end while**
32: return $\mathcal{H}[N_c, i]$ where $i = success(N_o, N_c)$

known in the literature of search and automated reasoning for artificial intelligence. When a node $n = \langle s : t, H_1, H_2, E \rangle$ [(2)] is simplified (both s and t are the normal forms in the corresponding processes) and expanded (line 24). We put it into N_c as the hypothesis and try to analyze all processes P held by n (line 28). It is not efficient to directly analyze all processes covered by n. Because although after operation $N.subsum\_P()$, the state $\langle \mathcal{E}[N_c, i], \mathcal{H}[N_c, i] \rangle$ becomes unique, there may still exist duplicate projections where $\mathcal{H}[N_c, i] = \mathcal{H}[N_c, j]$. We created a hash map $[\mathcal{S}_i, \mathcal{L}_i]$ in order to deal with the lemma generations in every process $i \in P$, where $\mathcal{S}_i$ indicates the hash code of $\mathcal{H}[N_c, i]$ (we created a class *projection* for projections, which contains the lazy hash code) and $\mathcal{L}_i$ denotes the result of possible lemmas. Since the key is unique in a hash map, we filter the duplicate keys easily by creating the map. The lemma generation function scans every set of hypotheses in different processes corresponding to the keys as the values of hash map $[\mathcal{S}_i, \mathcal{L}_i]$. Finally, the new nodes as possible lemmas in corresponding processes of n are constructed, then they are put into N_o (line 28).

The operation $expand(N, N', n)$ is the core of the whole procedure. Let $n = \langle s : t, H_1, H_2, E \cup E' \rangle$. The operation applies the EXPAND rule of RI in all processes of E' that can orient the equation $s \leftrightarrow t$ from left to right. The set E' is moved from the third label to the first in n since in each process in E' the conjecture $s \leftrightarrow t$ is removed and the new hypothesis $s \rightarrow t$ is added after the expansion. Moreover, for each new conjecture $s' \leftrightarrow t'$ in $\text{Expd}_u(s, t)$, a new node $\langle s' : t', \emptyset, \emptyset, E' \rangle$ is created in order to store the conjecture in the processes of E'. Note that (1) the direction of orientation and (2) the choice of the basic subterm to be expanded are two kinds of nondeterministic choices. Therefore there are two possible fork operations, where one is that to fork the original index $p \in N \cup N'$ into $p1$ and $p2$ by different choice of orienting directions (i.e., left to right or right to left), another is to fork the index p' into $p'1, \ldots, p'k$, if term s has k basic subterms to expand. In our implementation, we follow the discipline of functional programming by never mutating the nodes. We

[(2)] $n.mir$ represents the symmetric case of n

just update them from outside. This means the method needs to return the intermediate results as fresh sets of nodes. The result is structured as a tuple $\langle \mathcal{F}_1, \mathcal{F}_2, \mathcal{M}, \mathcal{C} \rangle$ where:

$\mathcal{F}_1$**:** the forked nodes from N (i.e., the labels of original nodes in N are forked into new labels depending on the nondeterministic choices)

$\mathcal{F}_2$**:** the forked nodes from N' (i.e., the labels of original nodes in N' are forked into new labels depending on the nondeterministic choices)

$\mathcal{M}$**:** the nodes "modified" during $expand(N, N', n)$ operation (i.e., a set only contains one node n' which holds the same datum $s : t$ but modified labels)

$\mathcal{C}$**:** the nodes newly created by $expand(N, N', n)$ operation (i.e., the nodes containing new conjectures)

The SIMPLIFY_R rule applies the SIMPLIFY rule of RI using a rewrite in the equational axiom $\mathcal{R}$, which is common to all processes. E is the set of all processes that have $s \leftrightarrow t$ as a conjecture. Since this equation is transformed to an equation $s' \leftrightarrow t$, the set E is removed from the original node, and a new node $\langle s' : t, \emptyset, \emptyset, E \rangle$ is created. The SIMPLIFY_H rule is almost the same as SIMPLIFY_R. The difference is that SIMPLIFY_R applies a rule of $\mathcal{R}$, while SIMPLIFY_H applies an inductive hypothesis of $\mathcal{H}$, which may exist only in some distinguished processes. This makes the third labels of the original node and the new node $E \backslash H$ and $E \cap H$, respectively.

The operation $simplify(N, N', n)$ applies the rule SIMPLIFY_R and SIMPLIFY_H to n as much as possible. Note that in the original MRIt, the two rules are defined separately. However, in our implementation we combine the two rules into one operation because we have to fork the other nodes N and N' at the same time. The rewrite strategy often plays an important role in simplification [16] [19] [35]. Therefore we fork the original index $p \in N \cup N'$ into $p1, \ldots, pk$, if there are k normal forms generated by different strategies (e.g., outermost and innermost strategy). Like the result tuple of EXPAND operation, the result of SIMPLIFY operation is also

structured as a tuple $\langle \mathcal{F}_1, \mathcal{F}_2, \mathcal{M}, \mathcal{N} \rangle$, where $\mathcal{N}$ represents the nodes newly created by $simplify(N, N', n)$ (i.e., the nodes containing rewritten term s' with modified labels).

The operation $N.delete()$ removes from N all nodes that contain a trivial equation, and returns the remaining nodes as N'. This operation is applied to the nodes created by rules SIMPLIFY_R, SIMPLIFY_H and EXPAND.

The operation $N.gc()$ implements the rule GC of MRIt, removes the nodes with three empty labels. It can effectively reduce the size of the current node database by removing nodes with three empty labels, because no processes contain the corresponding rule or equation.

The operation $N.subsume()$ combines two nodes into a single one when they contain the variant data (which are the same as each other up to renaming of variables). The duplicate indices in the third labels are removed to preserve the label conditions. The operation $N.subsume()$ is invoked by the operation $union(N, N')$ which is designed for combining nodes N and N'. We exploited the same *lazy* technique as [23] [24] to gain efficiency by creating a hash map $[\mathcal{I}_s, \mathcal{N}]$, where $\mathcal{N}$ is a *list* of nodes and $\mathcal{I}_s$ is a lazy value defined in the node class as the *size* of the node, so that we need only check the nodes with the same size as the original nodes. This check can be done effciently by using the hash map with the node size as its key. In other words, for every $n \in N$, n uses its size $\mathcal{I}_n$ as the key to $[\mathcal{I}_s, N]$, then the set $\mathcal{N}_n$ containing all the nodes with same size $\mathcal{I}_n$ is looked up for the nodes with variant data.

The operation $N.subsume\_P()$ stops redundant processes, which have the same state as other existing processes. The function $sub(N, L)$ is defined as $sub(N, L) = \{\langle s : t, H_1 \backslash L, H_2 \backslash L, E \backslash L \rangle \mid \langle s : t, H_1, H_2, E \rangle \in N\}$. It simply removes all indices in L from every node in N.

The procedure $success(N_o, N_c)$ checks if this induction procedure has succeeded. The process succeeds if there exists an index $i \in I$ such that i is not contained in any labels of N_o and any E labels of N_c nodes. Then $\mathcal{E}[N_o \cup N_c, i] = \emptyset$, and $\mathcal{R}[N_c, i]$ is a set of rewrite rules corresponding to the equations employed as the inductive

hypotheses in the proof. The proof details will also be captured by the program as an output.

Note that in lines 22 to 26 of *Algorithm 5.2*, we apply SUBSUME_P rule of MRIt to N_o, N_c and n by the same context. This means we should implicitly subsume the same duplicate indices L (depends on their states) with N_o, N_c and n (i.e., $N_o = sub(N_o, L), N_c = sub(N_c, L), n = sub(\{n\}, L).head$). For the same reason, we build $simplify(N_o, N_c, n)$ and $expand(N_o, N_c, n)$ to take three parameters in order to fork nodes from N_o and N_c at the same time.

The operation $N.choose()$ always chooses the minimal node in terms of its size to make the computation efficient. And there is another heuristic idea in our implementation different from the original MRIt. We try to simplify the conjectures at the very first before we expand them. Because we found that some inductive theorems were often reducible by the given TRS. We are not sure if this will make the proofs shorter (because in some cases it does while others not). However in many cases observed in our experiments, this will reduce the choices of nodes as well as the scale of the whole node database.

5.3 Lazy Evaluation

Lazy evaluation, also called *call-by-need* is an evaluating strategy which delays the evaluation of an expression until its value is needed and which also avoids repeated evaluations with the help of the 'memoization' mechanism to share the common computational results.

The advantage of lazy evaluation is often described as two parts:

(i) It only evaluates accessed parts of a value structure, so that it can define potentially infinite data structure in a natural way.

(ii) It stores the result once evaluation is done to avoid needless calculations, with the help of the 'memoization' mechanism.

In our implementation, we defined the following data structures: *Term, Equation, Rule, Node, Nodes*. Among them, *nodes* is a potentially infinite data structure because it is a set of *nodes* the number of which may keep increasing during the procedure. One may think that we can simply define the nodes as a lazy structure, so that when one of its nodes is accessed, other ones remain untouched. However, because we use the open/closed lists algorithm to search the database and pick from the open list the node with the smallest term pair, the system has to check every node of the open list to decide which one to pick first. That means every node would be accessed at least once, which would not benefit from the advantage (i) of lazy evaluation.

The same situation applies to the implementation of *term*. A *size()* function of term returns the number of its subterms. Even if we define the subterms as a lazy structure, since every subterm will be accessed at least once, it will turn out that there is no difference from the normal 'eager-evaluation' settings.

However, we can make use of the advantage (ii) of lazy evaluation by storing the result of evaluation. For example, we can still define subterms of a term as a lazy structure, because it is so frequently accessed during the procedures and the returning results do not change as long as the term does not change.

Example 5.3.1. There are two functions and one lazy value. Firstly, define a lazy value subts (line 2) for storing the result of getsubterms (line 3). Then define a normal function (line 1) to get this value. Notice that, getsubterms() is only invoked once no matter how many times subterms() is invoked.

```
1 def subterms()= subts
2 lazy val subts = this.getsubterms()
3 def getsubterms(): Seq[Term] = {
4     //calculate and return all subterms
5     ...
6 }
```

As for the *size()* function, we can also put its result into a lazy value just like the

tricky usage introduced in Example 5.3.1 to make it a "*lazy function*".

Another example is a lazy hash table to divide the set of nodes by their size. Because subsume() function is invoked frequently during the whole procedure both in MKB/MRIt, it would make the program slower to simply check all of the nodes in N, when N was updated after rewrite operations. To gain efficiency, we created a $lazy$ hash map $[\mathcal{I}_s, \mathcal{N}]$, where $\mathcal{N}$ is a $list$ of nodes and $\mathcal{I}_s$ is a lazy value defined in the node class as the $size$ of the node (i.e., for a node $n = \langle s : t, r_0, r_1, e \rangle$, n.size $=$ s.size $+$ t.size), so that we need only check the nodes with the same size as the original nodes. This check can be done efficiently by using the hash map with the node size as its key. In other words, for every $n \in N$, n uses its size $\mathcal{I}_n$ as the key to $[\mathcal{I}_s, \mathcal{N}]$, then the set $\mathcal{N}_n$ containing all the nodes with same size $\mathcal{I}_n$ is looked up for the nodes with variant data.

Example 5.3.2. Define the hash map $[\mathcal{I}_s, N]$ as a lazy value *lazy*, so that it is calculated only once and then be stored as a constant object ready to be returned for repeated calculation requests afterwards.

```
1 lazy val hashtable:HashMap[Int,List[Node]]={
2   //calculate and return hashtable
3 }
```

All of the lazy values used in multi-context reasoning schema are summarized in Table 5.1. The 'Class' column indicates the classes holding the lazy values, while the 'Lazy values' column represents the specific lazy values set in the corresponding classes.

Table 5.1 lazy values

Class	Lazy values	
Nodes	hash table	labels
Term	size	subterm
Node	label	size(hash index)

Chapter 6

Experiments and Discussion

6.1 Completion Problems

In this section, we will show how lz-mkb performed with the lazy evaluation when run on a PC with i5 CPU and 4GB main memory. All the problems solvable using the lexicographic path orderings for the termination check were selected as the sample problems from [39]. For example, the problem 1 is from the group theory. It contains three equations

$$\mathcal{E}_0 = \left\{ \begin{array}{l} f(x, f(y, z)) = f(f(x, y), z), \\ f(x, i(x)) = e, \\ f(x, e) = x \end{array} \right\}$$

where $\{f, i, e\}$ are function symbols (f is a binary operation, i represents the inverse and e is the identity element) and $\{x, y, z\}$ are variables. Given $\mathcal{E}_0$ and total lexicographic path orderings on $\{f, i, e\}$, the program returned a complete TRS $\mathcal{R}_c$ as

follows:

$$\mathcal{R}_c = \left\{ \begin{array}{l} f(x, i(x)) \rightarrow e \\ f(i(y), y) \rightarrow e \\ i(e) \rightarrow e \\ i(f(x, z)) \rightarrow f(i(z), i(x)) \\ i(i(x)) \rightarrow x \\ f(x, e) \rightarrow x \\ f(e, x) \rightarrow x \\ f(i(x), f(x, z)) \rightarrow z \\ f(x, f(i(x), z)) \rightarrow z \\ f(f(x, y), z) \rightarrow f(x, f(y, z)) \end{array} \right\}$$

The computation time for each examined problem is summarized in Table 6.1. The results obtained by the program using the lazy evaluation are labeled $lz\text{-}mkb$, and those obtained by the original one are labeled mkb. Clearly, *lz-mkb* is more efficient than *mkb* in all the problems examined.

Table 6.1 computation time of mkb and lz-mkb

problem	mkb(ms)	lz-mkb(ms)	reduced time	reduced(%)
1	15003	1959	13044	86.94
2	160	130	30	18.75
5	14997	2738	12259	81.74
8	275	205	70	25.45
11	90	60	30	33.33
14	480	351	129	26.88
17	85	65	20	23.53
19	730	471	259	35.48
30	140	95	45	32.14
avg.	-	-	-	40.47

We have summarized the lazy values used during the experiments in Table 5.1. The *label* in Node $n = \langle s : t, r_0, r_1, e \rangle$ calculates the union of $r0, r1$ and e. The variable *labels* in Nodes N collects all labels of the nodes in N. Associated with N is a *hash*

table which stores the nodes using their size as the hash key. It is used to gain efficiency during the optional operation N.subsume(). The $size$ and $subterm$ in Term are called frequently during the whole rewrite operation.

To see the different effects to the efficiency of the program with lazy Nodes(hash table,labels), lazy Term(size,subterm) or lazy Node(label,size), we ran them separately with the same problems as Table 6.1. The results are shown in Tables 6.2, 6.3, and 6.4.

Table 6.2 mkb and lz-mkb (lazy nodes only)

problem	mkb(ms)	lz-mkb(ms)	reduced time	reduced(%)
1	15003	12303	2700	18.00
2	160	140	20	12.5
5	14997	14468	529	3.53
8	275	230	45	16.36
11	90	75	15	16.67
14	480	400	80	16.67
17	85	80	5	5.88
19	730	570	160	21.92
30	140	130	10	7.14
avg.	-	-	-	13.18

Table 6.3 mkb and lz-mkb (lazy term only)

problem	mkb(ms)	lz-mkb(ms)	reduced time	reduced(%)
1	15003	2624	12379	82.51
2	160	145	15	9.38
5	14997	3890	11107	74.06
8	275	255	20	7.27
11	90	76	14	15.56
14	480	440	40	8.33
17	85	80	5	5.88
19	730	660	70	9.59
30	140	110	30	21.43
avg.	-	-	-	26.00

We can see the program with "lazy nodes only" (Table 6.2) works well with about 13 % reduced time on the average, because among all the callings of operation *union(N,N')* quite many of them return the original N, so the duplicate calculation

of the hash table is avoided. Also, the examination with "lazy term only" (Table 6.3) shows the best result with 26 % reduced time due to the frequency of rewriting callings during the whole procedure. However, the results with "lazy node only" (Table 6.4) are not very well with only 1.2 % reduced time (they have nearly the same computation time with the program without lazy values). The label and size in Node are always called at least once for every node by Nodes to create its hash table or check the end conditions, which could be the explanation for the results in Table 6.4.

Table 6.4 mkb and lz-mkb (lazy Node only)

problem	mkb(ms)	lz-mkb(ms)	reduced time	reduced(%)
1	15003	14880	123	0.82
2	160	155	5	3.13
5	14997	14921	76	0.51
8	275	270	5	1.82
11	90	90	0	0
14	480	475	5	1.04
17	85	85	0	0
19	730	725	5	0.68
30	140	136	4	2.86
avg.	-	-	-	1.21

6.2 Rewriting Induction Problems

In this section, we talk about some experimental results of *lz-itp*. In the implementation of *lz-itp*, we used a built-in termination checker (developed by ourselves) based on the dependency-pair method [4] [14] [15]. We also used the combination of polynomial interpretation and SAT solving as proposed in [13] in order to find reduction orders for ensuring termination. All experiments were performed on a PC with i5 CPU and 4GB main memory.

First we consider a propositional logic problem from [10]:

$$\mathcal{R} = \begin{cases} not(T) \to F \\ not(F) \to T \\ and(T, p) \to p \\ and(F, p) \to F \\ or(T, p) \to T \\ or(F, p) \to p \\ implies(p, q) \to or(not(p), q) \end{cases}$$

We prove the theorem

$$implies(and(p, q), or(p, q)) = T$$

by at least two EXPAND operations. It is obvious that the left-hand side of the theorem can be rewritten to

$$or(not(and(p, q)), or(p, q))$$

by the last rule $implies(p, q) \to or(not(p), q)$ first. Then it can be expanded to

$$or(not(and(T, p)), T) = T,$$

$$or(not(and(F, p)), p) = T,$$

where the first conjecture will be rewritten to $or(not(p), T) = T$ which needs the second expansion to finish the proof.

We can also expand the original target directly into

$$implies(p, or(T, p)) = T,$$

$$implies(F, or(F, p)) = T.$$

After the simplification of the first conjecture we will still get

$$or(not(p), T) = T$$

to be ready for the second expansion. As we can see, although the length of the proof did not change, in our program, the first method checked 7 nodes with one successful process over 2 processes and the second checked 11 nodes with 2 successful processes over 6 processes.

Some other problems from [1] [10] also showed the similar performance summarized in the Table 6.5 and Table 6.6: where "# of nod." shows the number of

Table 6.5 simplify first

problem	time (ms)	# of nod.	# of succ.	# of proc.
ex_1	17988	263	9	105
ex_2	6706	305	2	32
ex_3	1108	37	2	9

Table 6.6 expand first

problem	time (ms)	# of nod.	# of succ.	# of proc.
ex_1	47866	276	18	223
ex_2	7075	398	2	26
ex_3	1322	57	2	11

processed nodes when the procedure succeeded; "# of succ." shows the number of successful processes on average during the computation; "# of proc." shows the number of all processes when a process has succeeded. We can see that in these problems, the "simplify first" strategy could reduce the number of processed nodes so that the computation time of the whole procedure was also reduced.

In our implementation, the SIMPLIFY operation tries two rewrite strategies: the leftmost innermost strategy and the leftmost outermost strategy. Since MRIt also works on the set of nodes, we exploited the lazy evaluation scheme for the nodes manipulation proposed in [23] [24] to gain more efficiency. Moreover, we implemented the lemma exploration function with divergence detection [41] to deal with

the lemma-required problems. The problems selected from [1] which need appropriate lemmas were examined as shown in Table 6.7.

Table 6.7 experimental results of inductive theorem problems

problem	lem_1	lem_2	lem_3	lem_4	lem_5	lem_6
lz-itp	602	932	12379	17738	1050	1023
mrit+	615	969	12801	18090	1075	1049
mrit	-	-	12952	-	-	-

Note that the *mrit+* in Table 6.7 stands for an implementation of MRIt with lemma exploration, while the *mrit* stands for the original implementation of MRIt. The *mrit* failed in most of the cases with a time limit in 60000ms. We can see *lz-itp* which used the lazy evaluation was more efficient than *mrit+*.

6.3 Lemma-Required Problems

In this section, we present some results of the experiments to see how POSTULATE BY PERIPHERAL SCULPTURE and POSTULATE BY JOINING are effective in MULTI-CONTEXT POSTULATE.

6.3.1 Settings for Experiments

The experiments were performed with MRIt and MRIt+ on a PC with i5 CPU and 4GB memory. We used a total of 80 test problems, most of which were modified from Dream Corpus examples created by the Mathematical Reasoning Group, University of Edingurgh. The test problems borrowed from Dream Corpus are available at: http://kussharo.complex.eng.hokudai.ac.jp/∼haru/mrit/. All the problems mentioned in Table 6.8 are listed in Appendix B. In Dream Corpus, there were 69 unconditional equational problems suitable for the input to our system. In addition, we included 11 other problems which cannot be solved without lemma generation. For MRIt+, we had developed a built-in termination checker based on the dependency-pair method [4] [14] [15]. We had also implemented the combination of polynomial

interpretation and SAT solving as proposed in [13]. The time limit for each problem solving was set to 15 minutes.

We used the following strategy to apply the inference rules of MRIt+.

(i) Choose a node n with the smallest size (where the size of a node $\langle s : t, ... \rangle$ is the number of symbols constituting s and t), then apply EXPAND.

(ii) Normalize all nodes by applying SIMPLIFY-R and SIMPLIFY-H.

(iii) Apply DELETE, GC, SUBSUME and SUBSUME-P as much as possible, and if there exists a process p such that $\mathcal{E}[N, p] = \emptyset$, then stop with success.

(iv) Apply MULTI-CONTEXT POSTULATE, then activate the filtering process (where the maximal size of the randomly-generated ground terms is limited to 3), and go back to step (i).

To increase the capability of dealing with more potential proofs, we changed a condition in the EXPAND inference rule from $u \in \mathcal{B}(s)$ to $u \in \mathcal{QB}(s)$ based on [3] as follows. A term u is a *quasi-basic term* with respect to R, if (1) $root(u)$ is a defined symbol and (2) for all $l \to r \in R$ such that l is a basic term, l is unifiable with $cap(u)$, where $cap(f(u_1, ..., u_n))$ is a term $f(w_1, ..., w_n)$ with each w_i obtained from u_i by replacing maximal subterms with defined root symbol by fresh *constants*. The set of quasi-basic terms of s is denoted by $\mathcal{QB}(s)$.

For example, when $R = \{0 + x \to x, s(x) + y \to s(x + y)\}$, the term $z + (v + 0)$ is not basic but quasi-basic, as each left-hand is unifiable with $z + c$, where c is a fresh constant.

Note that in the original definition by Aoto [3], cap was defined with fresh *variables* (rather than constants) prohibited from being instantiated when unifying $cap(u)$ with l. This would require a slightly special unification algorithm. Implementable with the standard unification algorithm, our definition is slightly simpler than and clearly equivalent to Aoto's (as constants are never instantiated).

6.3.2 Results of Experiments

Among the test problems, 36 problems were solved by MRIt (without automated lemma generation). As for the rest of the problems (which, intuitively, need lemmas), MRIt+ (with MULTI-CONTEXT POSTULATE including both BY PERIPHERAL SCULPTURE and BY JOINING) solved 18 more problems that MRIt failed to solve. We also separately tested these problems according to the combinations of the two postulation methods and summarized the results in Table 6.8.

Table 6.8 experimental results of lemma-required problems

	SCULPTURE			JOINING			SCULPTURE&JOINING		
No.	time (ms)	SUG/ GENR	#of proc.	time (ms)	SUG/ GENR	#of proc.	time (ms)	SUG/ GENR	#of proc.
p_1	1688	19/113	13	∞	-	-	1662	19/113	13
p_2	1286	10/14	11	∞	-	-	1232	10/14	11
p_21	967	13/13	15	∞	-	-	992	13/13	15
p_22	28850	200/200	286	2362	7/7	22	2793	32/32	47
p_23	6875	29/29	26	5347	6/22	11	7090	35/51	28
p_24	7853	74/74	79	1348	2/2	9	1408	23/23	24
p_25	1612	25/25	40	∞	-	-	1732	25/25	40
p_26	10475	66/66	47	∞	-	-	29508	212/262	127
p_27	2225	14/14	19	2001	2/2	15	1904	16/16	21
d_25	∞	-	-	9433	133/240	36	9088	133/240	45
d_43	∞	-	-	775	18/32	4	737	18/32	4
d_47	∞	-	-	258	8/10	4	212	8/10	4
d_60	∞	-	-	1679	26/26	10	1683	26/26	10
d_111	∞	-	-	878	4/46	12	805	4/46	12
d_116	∞	-	-	729	4/40	12	891	4/40	12
d_232	460	6/6	5	∞	-	-	656	10/12	5
d_270	68603	864/2076	54	∞	-	-	48090	472/1170	222
d_1052	3108	16/16	28	2829	15/15	12	2902	19/19	15

In Table 6.8, the problem numbers with the prefix "d_" indicate some of the 69 problems selected from Dream Corpus, and those with "p_" indicate some of the 11 problems added by the authors. The first header indicates whether the problems were tested by SCULPTURE only, JOINING only or both SCULPTURE and JOINING. The "time(ms)" column shows the computation time of MRIt+ in milliseconds, where ∞ indicates that MRIt+ did not succeed within the time limit of 15 minutes (900000 ms). The "SUG/GENR" column shows the rate of generated conjectures that got

through the test by the conjecture filter, where GENR indicates the number of generated conjectures and SUG indicates the number of suggested lemmas that passed the conjecture filter. The "# of proc." column shows the number of the generated processes in MRIt+, where the data are given only for the successful trials, as other trials are considered to have generated an indefinite number of processes.

The results are very pleasing. The problems p_1,p_2,p_21,p_25, p_26, d_232 and d_270 could not be solved by JOINING but solved by SCULPTURE. In particular, p_1 was solved automatically, although the work in [41] had suggested some additional manual works to resolve the divergence. The problems d_25, d_270 and all problems with the prefix "p_" introduced complex differences as discussed in Example 4.9 (i.e., parallel and nested differences), but our method could treat such differences appropriately at a relatively small cost. Note that, by renaming $reverse$ to r, the problem d_43, $r(r(x : xs)) = x : xs$, is essentially equivalent to Example 3.1, $r(r(xs)) = xs$, because $r(r(nil)) = nil$, the base case for the latter, is trivially established by proving (6). In fact, both problems have been solved by MRIt+.

By observing the "SUG/GENR" columns, we see that the conjecture filter prevents incorrect conjectures from being suggested as lemmas for further processing. Clearly, the rate of correctly generated lemmas depends on the specific problems with specific methods. The average rate of correctly generated lemmas was 87% in SCULPTURE and 67% in JOINING.

6.3.3 Effectiveness of Peripheral Sculpture

The SCULPTURE method and the JOINING method can mutually strengthen the ability of generating effective lemmas to solve more problems, because one can solve some problems that the other could not. In addition, these two methods can also solve the same problems by postulating different conjectures. In our experiments, we have observed such different resolutions of different divergences in five problems: p_22, p_23, p_24, p_27 and d_1052. For example, in the experiment with p_24,

we observed a process generating the following diverging sequence:

$$y + (x + y) \rightarrow (y + y) + x,$$
$$y + \boxed{\underline{s(x + \boxed{s(\underline{y})})}} \rightarrow (y + \boxed{s(\underline{y})}) + \boxed{s(\underline{x})},$$
$$\ldots\ldots$$

SCULPTURE successfully resolved this divergence and completed the proof after postulating the following conjecture:

$$\nu + (x + y) = (\nu + y) + x.$$

Meanwhile, another process started to generate another diverging sequence as follows:

$$y + 0 \rightarrow y,$$
$$y + \boxed{s(\underline{0})} \rightarrow \boxed{s(\underline{y})},$$
$$\ldots\ldots$$

JOINING resolved this divergence and completed the proof after postulating the following conjecture:

$$y + s(x) = s(y + x).$$

Since these two processes are run concurrently, MRIt+ will complete the proof successfully as soon as one of them succeeds. In our actual experiment, JOINING reached the success earlier than SCULPTURE. Note that the overhead by running SCULPTURE together with JOINING for this case (p_24) was only $1408 - 1348 = 60$ ms, i.e., 4.3%.

Intuitively, JOINING focuses on the parts inside the wave-fronts, while SCULPTURE works on the opposite way by focusing on the parts outside the wave-fronts. Such different characteristics of them are why we can expect them to resolve different divergences in different processes of MRIt+.

6.3.4 Effectiveness of Multi-Context Postulation

An obvious advantage of multi-context reasoning systems commented in [35] is that they allow, in parallel, various strategies to be tried and various non-deterministic choices to be made in an efficient way. Thanks to this advantage, MULTI-CONTEXT POSTULATE can increase the possibility of success by combining different postulation methods in multi-context reasoning. In particular, there is no need to care about the unsoundness of the lemma postulation methods. By observing the "SCULPTURE&JOINING" column in Table 6.8, we see that SCULPTURE and JOINING worked very well together in MULTI-CONTEXT POSTULATE and solved the problems that might not have been solved otherwise. Since it is hard in practice to predict which process may face what kind of divergence, we use MULTI-CONTEXT POSTULATE to automatically give every diverging process a chance to adopt different postulation methods in different combinations.

In particular, multi-context reasoning can help different postulating methods cooperate with each other for leading to easier proofs. For example, in d_270, SCULPTURE suggested a conjecture

$$(\nu + (x + 0)) + (y + 0) = \nu + (x + y)$$

which could lead to a successful proof by picking up the subterm $x+0$ for expansion.

However, if JOINING had been enabled as well, it additionally suggested a smaller-sized conjecture

$$(x + y) + z = x + (y + z)$$

after $\nu + (x + 0)$ was picked up for expansion. Simple enough to be focused on by the heuristics of MRIt+, this conjecture turned out to be very useful for a shorter proof. This is why MULTI-CONTEXT POSTULATE with both PERIPHERAL SCULPTURE and JOINING spent less time than that with SCULPTURE only, as shown in the d_270 row of Table 6.8.

Chapter 7

Conclusion

In this dissertation, we proposed a new automated lemma generation method *peripheral sculpture* and multi-context schemes to incorporate this method for rewriting induction. We also presented: 1) new implementations of multi-context completion system *lz-mkb* based on MKB, 2) new implementations of multi-context algebraic inductive theorem prover *lz-itp* based on MRIt, both of which efficiently simulate the execution of parallel KB/RIt processes by dynamically dealing with the nondeterministic choices. Because these systems rely on the manipulation of the *node* database, we exploit the lazy evaluation schemas [23] [24] to gain more efficiency.

In rewriting induction (RI), the importance of lemma generation arises because the divergence prevents the procedure from getting a successful proof if no appropriate lemmas are postulated. Supplying appropriate lemmas is generally difficult since acquiring mathematical intuition and experience to postulate lemmas is not a trivial task to general users. Automated lemma generation, therefore, is desired. The classic unsound lemma generation method [41] gives higher ability of generating appropriate lemmas with a modest computational cost. We put into the Walsh's framework a new heuristic lemma generation method, *peripheral sculpture* by the following steps: 1) detect a potential divergence from the sequence of generated conjectures; 2) generate candidate lemmas by calculating the *peripheral sculptures* in the diverging sequence. Our method greatly improves the practical usefulness to make the theorem prover more powerful without introducing significant cost increase.

Since the method is unsound, there is no guarantee of the correctness of the generated candidate lemmas. If the system accepts a wrong lemma, it could cause an infinite sequence of wasteful inferences. Therefore, we need to separate into parallel contexts the choices of whether to accept the candidate lemma or not. From this point of view, the combination of multi-context induction and lemma generation can be a new research area of automated reasoning. As a means to study this area, we extended the efficient *multi-context rewriting induction system* of Sato and Kurihara [35] with our lemma generation method. The experimental results show that, with no much redundant costs, we have succeeded in solving several lemma-required benchmark problems which encountered complex differences (i.e., parallel and nested differences) and which the original systems [35] [41] could not solve.

There are several problems of KB or RI/RIt in practice. Firstly, the problem of KB is that it is difficult to set an appropriate reduction orders before the procedure starts. Secondly, the problem of RI/RIt is that the nondeterministic choices may cause the procedures to end up with failing results (i.e., divergence or fail). Thirdly, appropriate lemmas are often required in practice for RI/RIt to achieve successful proofs. The first problem can be partially solved by the multi-context reasoning system MKB which simulates the mutually related processes into virtually single processes. As for the second and third problems, MRIt pursues all nondeterministic choices trying to lead the procedure to a successful result by exploiting the schema of MKB. We expanded our method into multi-context schema to offer a practically useful framework for inductive theorem provers. In the implementation, we designed the systems in an object-oriented way so that we could build and reuse the classes to organize the term structures, substitutions, nodes, inference rules, etc. At the same time, we followed the discipline of functional programming in coding so that it could be safer and easier to execute the program in a physically parallel computational environment. We also enabled *lazy evaluation* mechanism of Scala to improve the performance when the systems face large problems. The results show that our algebraic reasoning systems *lz-mkb* and *lz-itp* are more efficient than the

original ones of MKB or MRIt.

Combination with other postulation methods (target-aimed or bottom-up) in multi-context reasoning systems to develop more powerful and efficient inductive theorem provers is clearly one of the future works.

References

[1] T. Aoto, "Dealing with non-orientable equations in rewriting induction,"Proc. 17th International Conf. on Rewriting Techniques and Applications, Lecture Notes in Computer Science, vol. 4098, pp.242-256, 2006.

[2] T. Aoto, "Rewriting induction using termination checker," JSSST 24th Annual Conf., 3C-3, 2007 (in Japanese).

[3] T. Aoto, "Designing a rewriting induction prover with an increased capability of nonorientable equations," Proc. Symbolic Computation in Software Science AustrianJapanese Workshop, RISC Technical Report, vol.08-08, pp.1-15, 2008.

[4] T. Arts and J. Giesl, "Termination of term rewriting using dependency pairs," Theoretical Computer Science, vol.236, pp.133-178, 2000.

[5] F. Baader and T. Nipkow, Term Rewriting and All That, Cambridge University Press, 1998.

[6] L. Bachmair, Canonical Equational Proofs, Birkhäuser, 1991.

[7] D. Basin and T. Walsh, "Difference matching," Proc. 11th International Conf. on Automated Deduction, Lecture Notes in Artificial Intelligence, vol.607, pp.295-309, 1992.

[8] D. Basin and T. Walsh, "Difference unification," Proc. 13th International Joint Conf. of Artificial Intelligence, pp.116-122, 1993.

[9] D. Basin and T. Walsh, "Termination orderings for rippling," Proc. 12th International Conf. on Automated Deduction, Lecture Notes in Artificial Intelligence, vol 814, pp.466-483, 1994.

[10] R. S. Boyer and J. S. Moore, A Computational Logic, Academic Press, New York, 1979.

[11] A. Bundy, A. Stevens, F. van Harmelen, A. Ireland, and A. Smaill, "Rippling: a heuristic for guiding inductive proofs," Artificial Intelligence, vol.62, pp.185-253, 1993.

[12] N. Dershowitz and J.-P. Jouannaud, "Rewrite systems," in Handbook of Theoretical Computer Science, ed. J. van Leeuwen, vol.B, pp.243-320, MIT Press, 1990.

[13] C. Fuhs, J. Giesl, A. Middeldorp, P. Schneider-Kamp, R. Thiemann, and H. Zankl, "SAT solving for termination analysis with polynomial interpretations," Proc. 10th International Conf. on Theory and Applications of Satisfiability Testing, Lecture Notes in Computer Science, vol. 4501, pp.340-354, 2007.

[14] J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke, "Mechanizing and improving dependency pairs," Journal of Automated Reasoning, vol.37, no.3, pp.155-203, 2006.

[15] N. Hirokawa and A. Middeldorp, "Tyrolean termination tool: techniques and features," Information and Computation, vol.205, no.4, pp.474-511, 2007.

[16] K. Hirotaka, T. Yoshihito, "Inductionless Induction and Rewriting Induction," Computer Software, vol.17, No.6, pp.1-12, 2000 (in Japanese).

[17] G. Huet and J. M. Hullot, "Proofs by induction in equational theories with constructors," Journal of Computer and System Science, vol.25, pp.239-266, 1982.

[18] D. Hutter, "Guiding induction proofs," Proc. 10th International Conf. on Automated Deduction, Lecture Notes in Artificial Intelligence, vol.449, pp.147-161, 1990.

[19] G. Huet and J.-M. Hullot, "Proofs by induction in equational theories with constructor," Journal of Computer and System Sciences, vol.25, no.2, pp.239—266, 1982.

[20] G. Huet and D. C. Oppen, "Equations and rewrite rules: A survey," in R. Book (ed.), Formal Language Theory: Perspectives and Open Problems, Academic Press, pp.349-405, 1980.

[21] CC. Ji, M. Kurihara, and H. Sato, "Multi-context automated lemma generation for term rewriting induction with divergence detection," IEICE Transactions on Information and Systems, vol.E102-D, No.2, pp.-, Feb. 2019.

[22] CC. Ji, H. Sato, and M. Kurihara, "A new implementation of multi-context algebraic inductive theorem prover," Lecture Notes in Engineering and Computer Science: Proc.The World Congress on Engineering and Computer Science 2015, pp.109-114, 2015.

[23] CC. Ji, H. Sato, and M. Kurihara, "Lazy evaluation schemes for efficient implementation of multi-context algebraic completion system," IAENG International Journal of Computer Science, vol. 42, no. 3, pp. 282—287, 2015.

[24] CC. Ji, H. Sato, and M. Kurihara, "An efficient implementation of multi-context algebraic reasoning systems with lazy evaluation," in Lecture Notes in Engineering and Computer Science: International MultiConference of Engineers and Computer Scientists 2015, pp. 201—205, 2015.

[25] M. Johansson, L. Dixon, and A. Bundy, "Conjecture synthesis for inductive theories," Journal of Automated Reasoning, vol. 47, no.3, pp.251-289, 2011.

[26] J. W. Klop, “Term rewriting systems,” in Handbook of Logic in Computer Science, ed. S. Abramsky et al., pp.1-116, Oxford University Press, 1992.

[27] D. E. Knuth and P. B. Bendix, “ Simple word problems in universal algebras, ” J. Leech(ed.), Computational Problems in Abstract Algebra, Pergamon Press, pp.263-297, 1970.

[28] M. Kurihara and H. Kondo, “Completion for multiple reduction orderings,” Journal of Automated Reasoning, vol.23, no.1, pp.25-42, 1999.

[29] D. R. Musser, “On proving induction properties of abstract data types,” Proc. 7th ACM Symposium on Principles of Programming Languages, pp.154-162, 1980.

[30] M. Odersky, Programming in Scala, 2nd ed., Artima Press, 2010.

[31] U. Pascal and K. Emmanuel, “Sound generalizations in mathematical induction,” Theoretical Computer Science, vol.323, pp.443-471, 2004.

[32] D. A. Plaisted, “Semantic confluence tests and completion methods,” Journal of Information and Control, vol.65, pp.182-215, 1985.

[33] D. A. Plaisted, “Equational reasoning and term rewriting systems,” in Handbook of Logic in Artificial Intelligence and Logic Programming, ed. D. M. Gabbay et al., vol. 1, pp.274-367, Oxford Univ. Press, 1993.

[34] U. Reddy, “Term rewriting induction,” Proc. 10th International Conf. on Automated Deduction, Lecture Notes in Computer Science, vol.449, pp.162-177, 1990.

[35] H. Sato and M. Kurihara, “Multi-context rewriting induction with termination checkers,” IEICE Transactions on Information and Systems, vol.E93.D, no.5, pp.942-952, 2010.

[36] H. Sato, S. Winkler, M. Kurihara, and A. Middeldorp, "Multi-completion with termination tools (system description)," Proc. 4th International Joint Conf. on Automated Reasoning, Lecture Notes in Artificial Intelligence, vol.5195, pp.306-312, 2008.

[37] H. Sato, M. Kurihara, S. Winkler, and A. Middeldorp, "Constraint-based multi-completion procedures for term rewriting systems," IEICE Transactions on Information and Systems, vol.E92-D, pp.220-234, 2009.

[38] S. Shimazu, T. Aoto, and Y. Toyama. "Automated lemma generation for rewriting induction with disproof," Computer Software, vol. 26, no. 2, pp.41-55, 2009 (in Japanese).

[39] J. Steinbach and U. Kühler, "Check your ordering - termination proofs and open problems," SEKI report SR-90-25 (SFB), Fachbereich Informatik, Universität Kaiserslautern, Germany, 1990.

[40] Terese, Term Rewriting Systems, Cambridge University Press, 2003.

[41] T. Walsh, "A divergence critic for inductive proof," Journal of Artificial Intelligence Research, vol. 4, pp.209-235, 1996.

[42] I. Wehrman, A. Stump, and E. Westbrook, "SLOTHROP: Knuth-Bendix completion with a modern termination checker," Proc. 17th International Conf. on Rewriting Techniques and Applications, Lecture Notes in Computer Science, vol.4098, pp.287-296, 2006.

[43] C.-P. Wirth, "History and future of implicit and inductionless induction: beware the old jade and the zombie!," in Mechanizing Mathematical Reasoning, Lecture Notes in Artificial Intelligence, vol.2605, pp.192-203, 2005.

Publications

Journal Papers

(i) CC. Ji, M. Kurihara, and H. Sato, "Multi-context automated lemma generation for term rewriting induction with divergence detection," *IEICE Transactions on Information and Systems*, vol.E102-D, No.2, pp.-, Feb. 2019 (to appear).

(ii) CC. Ji, H. Sato, and M. Kurihara, "Lazy evaluation schemes for efficient implementation of multi-context algebraic completion system," *IAENG International Journal of Computer Science*, vol. 42, no. 3, pp. 282—287, 2015.

International Conference Papers

(i) CC. Ji, H. Sato, and M. Kurihara, "An efficient implementation of multi-context algebraic reasoning systems with lazy evaluation," *Lecture Notes in Engineering and Computer Science: International MultiConference of Engineers and Computer Scientists 2015*, pp. 201—205, 2015.

(ii) CC. Ji, H. Sato, and M. Kurihara, "A new implementation of multi-context algebraic inductive theorem prover," *Lecture Notes in Engineering and Computer Science: Proc.The World Congress on Engineering and Computer Science 2015*, pp.109-114, 2015.

Lecture Talks

(i) CC. Ji and M. Kurihara, "Lazy evaluation schemes for efficient implementations of multi-context algebraic reasoning systems," *Computational Logic in the Alps workshop*, Innsbruck, Austria, September 5-9, 2016.

(ii) CC. Ji, H. Sato, and M. Kurihara, "Lazy evaluation schemes for efficient implementation of multi-context algebraic reasoning systems," *The 78th National Conference of IPSJ*, Yokohama, 2016.

Appendix A: Inference Rules of MRIt

DELETE: $N \cup \{\langle s : s, H_1, H_2, E\rangle\} \vdash N$

EXPAND: $N \cup \{\langle s : t, H_1, H_2, E \cup E'\rangle\} \vdash$
$N \cup \{\langle s : t, H_1 \cup E', H_2, E\rangle\}$
$\cup \{\langle s' : t', \emptyset, \emptyset, E'\rangle \mid s' \leftrightarrow t' \in \text{Expd}_u(s, t)\}$
if $E' \neq \emptyset, u \in \mathcal{B}(s)$ and
$\mathcal{H}[N, p] \cup \mathcal{R} \cup \{s \to t\}$ terminates for all $p \in E'$

SIMPLIFY-R: $N \cup \{\langle s : t, H_1, H_2, E\rangle\} \vdash$
$$N \cup \left\{ \begin{array}{l} \langle s : t, H_1, H_2, \emptyset\rangle, \\ \\ \langle s' : t, \emptyset, \emptyset, E\rangle \end{array} \right\}$$
if $E \neq \emptyset$ and $s \to_{\mathcal{R}} s'$

SIMPLIFY-H: $N \cup \{\langle s : t, H_1, H_2, E\rangle\} \vdash$
$$N \cup \left\{ \begin{array}{l} \langle s : t, H_1, H_2, E \backslash H\rangle, \\ \\ \langle s' : t, \emptyset, \emptyset, E \cap H\rangle \end{array} \right\}$$
if $E \cap H \neq \emptyset, \langle l : r, H, \ldots, \ldots\rangle \in N$,
and $s \to_{\{l \to r\}} s'$

FORK: $N \vdash \psi_P(N)$
for some fork function ψ and a set P of processes in N

GC: $N \cup \{\langle s : t, \emptyset, \emptyset, \emptyset\rangle\} \vdash N$

SUBSUME: $$N \cup \left\{ \begin{array}{l} \langle s : t, H_1, H_2, E\rangle, \\ \\ \langle s' : t', H_1', H_2', E'\rangle \end{array} \right\} \vdash$$
$N \cup \{\langle s : t, H_1 \cup H_1', H_2 \cup H_2', E''\rangle\}$
if $s : t$ and $s' : t'$ are variants and

$$E'' = (E \backslash (H_1' \cup H_2')) \cup (E' \backslash (H_1 \cup H_2))$$

SUBSUME-P: $N \vdash sub(N, L)$

if $\forall p \in L, \exists p' \in I(N) \backslash L$:

$\langle \mathcal{E}[N, p], \mathcal{H}[N, p] \rangle = \langle \mathcal{E}[N, p'], \mathcal{H}[N, p'] \rangle$

where $I(N)$ denotes the set of all processes that appear in a label of a node in N and $sub(N, L) = \{\langle s : t, H_1 \backslash L, H_2 \backslash L, E \backslash L \rangle | \ \langle s : t, H_1, H_2, E \rangle \in N\}$.

Appendix B: Problems and Generated Lemmas

Some problems used in our experiments and lemmas automatically generated by MRIt+ with the SCULPTURE&JOINING setting are listed in this appendix.
Each problem starts with its name, followed by rewrite rules (as axioms), followed by an equation (as an inductive theorem to be proved), and ends with the generated lemmas displayed in square brackets.

- p_1

  $nil@x \rightarrow x$

  $(x : ys)@zs \rightarrow x : (ys@zs)$

  $(xs@xs)@xs = xs@(xs@xs)$

  $[(xs@ys)@zs = xs@(ys@zs)]$

- p_2

  $0 + y \rightarrow y$

  $s(x) + y \rightarrow s(x + y)$

  $(x + x) + x = x + (x + x)$

  $[(x + y) + y = x + (y + y)]$

- p_21

  $0 + y \to y$

  $s(x) + y \to s(x + y)$

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

  $[(x + y) + z = x + (y + z)]$

- p_22

  $0 + y \to y$

  $s(x) + y \to s(x + y)$

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

  $[x + s(y) = s(x + y)]$

- p_23

  $0 + y \to y$

  $s(x) + y \to s(x + y)$

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

  $$\left[\begin{array}{l} (x + 0) + y = x + y \\ \\ (x + s(y)) + z = s((x + y) + z) \end{array}\right]$$

- p_24

  $0 + y \to y$

  $s(x) + y \to s(x + y)$

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

  $[x + s(y) = s(x + y)]$

- p_25

  $0 + y \to y$

  $s(x) + y \to s(x + y)$

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

$[(x + y) + z = x + (y + z)]$

- p_26

  $0 + y \rightarrow y$

  $s(x) + y \rightarrow s(x + y)$

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

  $$\left[\begin{array}{l} x + (y + 0) = x + y \\ \\ x + (y + s(z)) = s(x + (y + z)) \end{array}\right]$$

- p_27

  $0 + y \rightarrow y$

  $s(x) + y \rightarrow s(x + y)$

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

  $[x + s(y) = s(x + y)]$

- d_25

  $(x : xs)@ys \rightarrow x : (xs@ys)$

  $nil@ys \rightarrow ys$

  $reverse(x : xs) \rightarrow reverse(xs)@(x : nil)$

  $reverse(nil) \rightarrow nil$

  $reverse(xs@ys) = reverse(ys)@reverse(xs)$

  $[xs@(ys@zs) = (xs@ys)@zs]$

- d_43

  $(x : xs)@ys \rightarrow x : (xs@ys)$

  $nil@ys \rightarrow ys$

  $reverse(x : xs) \rightarrow reverse(xs)@(x : nil)$

$reverse(nil) \rightarrow nil$

$reverse(reverse(x : xs)) = x : xs$

$[reverse(xs@(x : nil)) = cons(x, reverse(xs))]$

- d_47

  $(x : xs)@ys \rightarrow x : (xs@ys)$

  $nil@ys \rightarrow ys$

  $reverse(x : xs) \rightarrow reverse(xs)@(x : nil)$

  $reverse(nil) \rightarrow nil$

  $length(x : xs) \rightarrow s(length(xs))$

  $length(nil) \rightarrow 0$

  $length(reverse(xs)) = length(xs)$

  $[length(xs@(x : nil)) = s(length(xs))]$

- d_60

  $0 + y \rightarrow y$

  $s(x) + y \rightarrow s(x + y)$

  $0 * y \rightarrow 0$

  $s(x) * y \rightarrow y + (x * y)$

  $double(0) \rightarrow 0$

  $double(s(x)) \rightarrow s(s(double(x)))$

  $double(x) = s(s(0)) * x$

  $[x + s(y) = s(x + y)]$

- d_111

  $0 + y \rightarrow y$

  $s(x) + y \rightarrow s(x + y)$

  $difference(0, j) \rightarrow 0$

$difference(s(i), 0) \rightarrow s(i)$

$difference(s(i), s(j)) \rightarrow difference(i, j)$

$difference((y + x), x) = y$

$[x + s(y) = s(x + y)]$

- d_116

  $0 + y \rightarrow y$

  $s(x) + y \rightarrow s(x + y)$

  $difference(0, j) \rightarrow 0$

  $difference(s(i), 0) \rightarrow s(i)$

  $difference(s(i), s(j)) \rightarrow difference(i, j)$

  $difference(s(y + x), x) = s(y)$

  $[x + s(y) = s(x + y)]$

- d_232

  $0 + y \rightarrow y$

  $s(x) + y \rightarrow s(x + y)$

  $0 * y \rightarrow 0$

  $s(x) * y \rightarrow y + (x * y)$

  $s(s(0)) * x = x + x$

  $[x + (y + 0) = x + y]$

- d_270

  $0 + y \rightarrow y$

  $s(x) + y \rightarrow s(x + y)$

  $0 * y \rightarrow 0$

  $s(x) * y \rightarrow y + (x * y)$

  $s(s(s(s(0)))) * x = s(s(0)) * (s(s(0)) * x)$

$$\left[\begin{array}{l}(x+(y+0))+(z+0)=x+(y+z)\\(x+y)+z=x+(y+z)\end{array}\right]$$

- d_1052

  $0+y \to y$

  $s(x)+y \to s(x+y)$

  $0*y \to 0$

  $s(x)*y \to y+(x*y)$

  $(x+y)*z=(x*z)+(y*z)$

  $[(x+y)+z=x+(y+z)]$