Schematic calculi for the analysis of decision procedures

Elena Tushkanova

SPIM

Thèse de Doctorat

école doctorale sciences pour l'ingénieur et microtechniques

UFC UNIVERSITÉ DE FRANCHE-COMTÉ

Schematic calculi for the analysis of decision procedures

Calculs schématiques pour l'analyse de procédures de décision

THESIS

presented and submitted on July 19, 2013

for the degree of

Doctor of philosophy at the University of Franche-Comté

(Computer Science)

by

Elena TUSHKANOVA

Composition of jury

President :	Claude Marché, Senior Researcher, INRIA, Saclay, France
Director :	Olga Kouchnarenko, Professor, University of Franche-Comté, Besançon, France Alain Giorgetti, Assistant professor, University of Franche-Comté, Besançon, France Christophe Ringeissen, Researcher, INRIA, Nancy, France
Referees :	Viorica Sofronie-Stokkermans, Professor, University Koblenz-Landau, Germany Sophie Tison, Professor, University of Lille, France
Examiner :	Nicolas Peltier, Researcher, CNRS, LIG, Grenoble, France

Department of Computer Science for Complex Systems femto-st SCIENCES & TECHNOLOGIES

Acknowledgments

There are many people who helped me to finish this thesis and whom I would like to thank.

First of all I would like to thank my former supervisor Senior Lecture Yury Belov, Computer Science Department of Yaroslavl State University, that has seen my potential in research work from the first year of my study at Yaroslavl State University. He has proposed my candidature to the Head of the Chair of theoretical informatics, Professor Valery Sokolov, when Professor Olga Kouchnarenko (UFR-ST, France) has offered an opportunity to participate in the work on probation in Inria (France) during the period from March 2009 till August 2009. I would like to thank Professor Valery Sokolov for choosing me for this opportunity, for his help in many administrative things in Russia, and of course for his support.

My internship has taken place at LIFC (Laboratoire d'Informatique de l'Université de Franche-Comté) under the supervision of Senior Scientist Claude Marché, Inria, Assistant Professor Alain Giorgetti, University of Franche-Comté, Computer Science Department, FEMTO-ST Institute, and Professor Olga Kouchnarenko. First I would like to thank Alain Giorgetti for his help and guidance in my research work. He has greatly enhanced my knowledge and comprehension of computer science. I would also like to thank Claude Marché. He has worked with me during my internship and has guided my research during that time. I am grateful to Olga Kouchnarenko that has proposed me a PhD post after our joint work during internship. Many thanks for her support and help during four years of working together.

I deeply appreciate the help of my co-director Researcher Christophe Ringeissen, Inria. I would like to express my deepest gratitude for his excellent guidance and patience.

The implementation part of this thesis would not have been possible without the help of the developer of narrowing in Maude, Santiago Escobar.

I would like to thank Professor Sophie Tison and Professor Viorica Sofronie-Stokkermans for accepting to be my referees. Also I would like to thank Claude Marché and Nicolas Peltier for accepting to be my examiners.

Additionally, I would like to thank my friends from the laboratory. They are Elizabeta Fourneret, Oscar Carrillo, Rami Kassab, Fouad Hanna, Aydée Sanchez-Santana, Kalou Cabrera, Adrien De Kermadec, Alois Dreyfus and Cédric Joffroy, for their continued support and help. Also I would like to thank my russian friends from Besancon. They are Maria Makarova, Elena Zinchenko, Valery Koroleva, Maxim Goryachev and Dmitriy Kuzikov.

I would never have been able to finish my dissertation without support from my family: my parents Alevtina and Alexandre Tushkanov, my brother Eugeniy, my granny Maria Sokolova, and Samuel Laurent. Thank you all for your support and patience when I wanted to give up.

Last but not the least, I would like to thank God, for answering my prayers, and for giving me the strength I needed to complete this work.

This thesis is dedicated to
my parents for their love, endless support and
encouragement, and to
my dear Samuel Laurent for his love and
giving me forces to make this work...

I love you!

Contents

List of Figures

Chapter 1

Introduction

Contents

1.1 Context

Nowadays computer programs are part of our everyday life. They are embedded in medical devices, transport, finance and banking, industry, aircraft control systems and many more sectors. Failures can bring not only bankruptcy, loss of time, but what is the most important loss of human lifes. Let us give few classical examples of software errors with extreme consequences.[1] In 1985-87, when a buggy software was controling the Therac-25 medical radiation therapy device, massive overdoses of radiation were administered to patients killing at least 3 of them. In 1993 an error in the flight-control software of a Swedish jetfighter generated massive disruption, and the plane finally crashed. A NASA subcontractor hired for building the Mars climate orbiter used English units instead of the metric system. As a result, the orbiter crashed almost immediately when it arrived at Mars in 1999, dashing all the hopes of a 327 million dollars project.

Some bugs may cause only trivial problems, but for many systems, failure is not an option. Therefore, it is of primary importance that such systems operate correctly and reliably. There are several methods to check the software correctness. Many bugs are discovered and eliminated through software testing. But a method like software testing cannot prove that the system, algorithm or program does not contain any errors and defects, and that it satisfies a certain property. Also the number of possible situations is so large, that only a tiny number of these situations can be tested. An alternative way of checking could be a formal verification that provides a mathematical proof that a

[1] From http://www.wired.com/software/coolapps/news/2005/11/69355?currentPage=all

program is correct with respect to a certain property. A formally verified program will work correctly for every given input.

An approach to formal verification is deductive verification. During the last decade, an important progress has been made in the field of deductive verification of programs. Now each popular programming language, like Java, C# and C, has its own formal specification language. For example, Java Modeling Language [LKP07] is designed for Java programs, Spec# [BDF$^+$05] is designed for C# and ACSL [BFM$^+$09] is designed for C. In this framework, functions and methods have pre- and post-conditions written in the specification language. If the precondition is met and the method terminates, then it should establish the postcondition. Why [FM07] is a platform for deductive verification of source code. From a source program annotated by specifications, it extracts verification conditions and transmits them to theorem provers like SMT solvers (Simplify, Z3, CVC3, Yices, Alt- Ergo, etc.) or proof assistants (Coq, Isabelle/HOL, PVS, etc.).

SMT (Satisfiability Modulo Theory) consists in deciding the satisfiability of a (ground) first-order formula with respect to a background theory. A satisfiability procedure is a decision procedure for the satisfiabilty problem, that is an algorithm that always terminates with a "yes" answer if the input formula is satisfiable, and with a "no" answer otherwise. In the context of this thesis, when we consider a decision procedure, we mean a satisfiability procedure. We focus on decision procedures specified as inference systems, which are sets of inference rules. An inference rule relates premises to a conclusion. For example, if $u = c$ and $u' = c$, then we conclude that $u = u'$. Designing and implementing these decision procedures remains a very hard task. To help the researcher with this time-consuming task, an important approach based on rewriting has been investigated in the last decade [ARR01, ARR03, ABRS09]. This approach allows building satisfiability procedures in a flexible way by using a general calculus for automated deduction, namely the Paramodulation Calculus ($\mathcal{PC}$) [NR01] (also called superposition calculus). In general, a fair and exhaustive application of the rules of $\mathcal{PC}$ leads to a semi-decision procedure that halts on unsatisfiable inputs (the empty clause is then generated) but may diverge on satisfiable ones. Fortunately, it may also terminate for some theories of interest in verification, and thus it becomes a decision procedure. In [ARR01, ARR03, ABRS09] it is shown how a paramodulation-based inference system can be used as a decision procedure for various theories including lists, arrays, records and combinations of them.

To ensure efficiency, it is very useful to have built-in axioms in the calculus, and so to design paramodulation calculi modulo theories. This is particularly important for arithmetic fragments due to the ubiquity of arithmetics in applications of formal methods. For instance, the standard paramodulation calculus has been extended in [NRR09c] in order to take into account the axioms of the theory of Integer Offsets. New combination methods *à la* Nelson-Oppen have been developed to consider unions of these theories sharing fragments of arithmetics. This paves the way of using non-disjoint combination methods within SMT solvers.

To reason on standard paramodulation calculus, a Schematic Paramodulation Calculus ($\mathcal{SPC}$) [LM02] has been studied to automatically prove termination. The advantage of $\mathcal{SPC}$ is that if it halts for one given abstract input, then $\mathcal{PC}$ halts for all the corresponding concrete inputs. More generally, $\mathcal{SPC}$ is an automated tool to check properties of $\mathcal{PC}$ like termination, stable infiniteness and deduction completeness. Improvements of the first

presentation of $\mathcal{SPC}$ have been proposed in [LT07, LRRT11]. The authors of these papers have focused not only on automatic decidability but also on automatic combinability.

Until now no schematization of the paramodulation calculus modulo the theory of Integer Offsets has yet been designed. Therefore, there is an obvious need for a method to automatically prove that an input theory admits a decision procedure based on paramodulation modulo Integer Offsets.

From the context the reader can understand that this thesis addresses problems related to the verification of software-based systems. We are mostly interested in the (safe) design of decision procedures for satisfiability problems that are at the core of SMT solvers. We focus on a rewriting-based approach for the design of decision procedures, which is based on the use of paramodulation calculi. In addition, we consider a modularity problem for a modeling language used in the Why verification platform, that extracts verification conditions from a source program annotated by specifications, and then transmits them to SMT solvers or proof assistants to check the program correctness. Thus, both problems considered in this thesis are related to SMT solvers.

1.2 Contributions

The contributions in this thesis address both practical and theoretical results:

(a) The main contribution of this thesis is a rule-based system implementing a complete many-sorted schematic paramodulation calculus for arbitrary theories. This tool is presented in Chapter 6. It allows us to automatically check whether the paramodulation calculus terminates for theories defined by arbitrary clauses and whether the related decision procedures are combinable. The tool can also be used to check the modular termination when a combination of either signature-disjoint theories, or theories with non-disjoint signatures is considered. Moreover, this implementation of schematic paramodulation provides a trace of each applied rule, which is very useful to validate or invalidate saturation proofs previously described in the literature. The correctness of this tool is validated in Chapter 7 by checking the decidability of classical theories such as the theory of lists (with and without extensionality), the theory of records without extensionality, the theory of possibly empty lists, the theory of arrays and the theory of recursive data structures for which paramodulation is known to terminate [ABRS09, NRR09c, LRRT11].

(b) The schematic paramodulation calculus has been improved thanks to our experimentations. In fact, the schematic paramodulation calculus proposed in [LRRT11] does not take into account the constants in the theory signature. For instance, the signature of the theory of possibly empty lists contains a constant nil. The schematic paramodulation calculus presented in this thesis takes these constants into account. This point is discussed in Chapter 4.

(c) A new schematic paramodulation calculus to describe saturations modulo Integer Offsets is presented in Chapter 5. The correctness of this calculus is validated thanks to our tool. Our approach requires a new form of schematization to cope with

arithmetic expressions. The interest of schematic paramodulation relies on a correspondence between a derivation using (concrete) paramodulation and a derivation using schematic paramodulation: Roughly speaking, the set of derivations obtained by schematic paramodulation over-approximates the set of derivations obtained by (concrete) paramodulation. We show under which conditions the termination of schematic paramodulation implies the termination of (concrete) paramodulation. Again, the fact of considering Integer Offsets requires some specific proof arguments. Thanks to this schematic calculus we can automatically check whether the paramodulation calculus modulo Integer Offsets is a decision procedure for the union of two non-disjoint theories sharing the theory of Integer Offsets.

(d) At the beginning of the PhD, we have participated to a study of modular specification of *generic* Java classes and methods under the supervision of Claude Marché, Alain Giorgetti and Olga Kouchnarenko. We have proposed extensions to the *Krakatoa Modeling Language*, a part of the Why platform for proving that a Java or C program is a correct implementation of some specification. This work is described in Chapter 8. The new constructs we propose for the specification of generic Java programs are presented through two significant examples: the specification of the generic method for sorting arrays which comes from the `java.util.Arrays` class in the Java API, and the specification of the `java.util.HashMap` class defining a generic hash map and its use for memoization. The key features are the introduction of parametricity both for types and for *theories* and an instantiation relation between theories. We discuss soundness conditions and their verification.

1.3 Plan

After this general introduction, we present in Chapter 2 everything needed for further reading this thesis. We define classical notions related to first-order logic and rewrite systems. We give some examples of theories axiomatizing datatypes such as lists, records and arrays. Then we discuss a well-known method for combining decision procedures for disjoint theories. And finally this chapter ends with a presentation of the useful notions of the rule-based language Maude used for the implementation of schematic paramodulation calculi.

Chapter 3 introduces paramodulation calculus which is a refutation-complete inference system at the core of all equational theorem provers. Then, we present a methodology based on saturation that provides satisfiability procedures. The satisfiability problem in the union of theories for which a satisfiability procedure of each theory is already built thanks to paramodulation calculus is also considered in this chapter. At the end of this chapter we present a paramodulation calculus developed for Integer Offsets, and we discuss the combination of theories sharing the theory of Integer Offsets.

Chapter 4 presents the schematic paramodulation calculus that can be used for proving termination of any fair paramodulation strategy and for checking whether paramodulation calculus decides the satisfiability problem for some unions of finitely presented theories. Since we propose some improvements of schematic paramodulation calculus, we prove

that a theorem stating that every clause in a saturation corresponds to a schematic clause in a schematic saturation still holds with our improvements.

In Chapter 5 we design a new schematic paramodulation calculus to describe saturations modulo Integer Offsets. We show under which conditions the termination of schematic paramodulation implies the termination of (concrete) paramodulation. We also study the question of automatic combinability of non-disjoint theories sharing the theory of Integer Offsets.

Our aim is to implement the schematic paramodulation calculi so that user could easily modify the code and to get a rule-based program which is as close as possible to the formal specification. The Maude language is very suitable for these purposes. It is a rule-based language that includes support for unification and narrowing, which are key operations of the calculus of interest. And the Maude meta-level provides a flexible way to control the application of rules and powerful search mechanisms. The implementation of the schematic paramodulation calculi is presented in Chapter 6. It has been implemented from scratch in Maude.

Chapter 7 reports on our experimentation results. We show that the schematic paramodulation calculus halts for some unit theories, such as the theory of lists (with and without extensionality) and the theory of records, for unit theories sharing Integer Offsets, such as the theory of lists with length, the theory of lists with integer elements and the theory of records with increment, and for some non-unit theories, such as the theory of possibly empty lists, the theory of arrays, and the theory of recursive data structures.

Chapter 8 presents an excerpt of KML, focusing on the part of that language which allows to specify algebraic-style data types [Mar07] and theories. It also proposes original specifications for a sorting algorithm and discusses their automatic proof. Moreover, it presents new specification constructs for specifying a generic Java method for sorting arrays and a generic Java class of hashmaps.

Chapter 9 concludes and gives some perspectives to this thesis.

1.4 Publications

The results presented in this thesis have been already published in the proceedings of workshops and international conference.

- The work presenting a rule-based system to execute a schematic paramodulation calculus for *unit unsorted* theories has been published as

  E. Tushkanova, A. Giorgetti, C. Ringeissen, and O. Kouchnarenko. A rule-based framework for building superposition-based decision procedures. In F. Durán, editor, *Proc. of the 9th Int. Workshop on Rewriting Logic and Its Applications (WRLA'12)*, volume 7571 of *Lecture Notes in Computer Science*, pages 221-239. Springer, 2012.

  In the paper submitted to the Journal of Science of Computer Programming (SCP) we go further and consider the general schematic paramodulation calculus for *arbitrary many-sorted* theories, not only its restriction to unsorted and unit ones.

- A presentation of the schematic paramodulation calculus for theories sharing the theory of Integer Offsets and its application to the termination of paramodulation will appear in the proceedings of the 24th International Conference on Rewriting Techniques and Applications (RTA'13).

  The full version of this paper has been published earlier as a research report:

  E. Tushkanova, C. Ringeissen, A. Giorgetti, and O. Kouchnarenko. Automatic Decidability for Theories Modulo Integer Offsets. Research Report RR-8139, INRIA, November 2012.

- The work addressing the question of modular specification of generic Java classes and methods has been published as

  A. Giorgetti, C. Marché, E. Tushkanova, and O. Kouchnarenko. Specifying generic Java programs: two case studies. In C. Brabrand and P.-E. Moreau, editors, *Proc. of the 10th Workshop on Language Descriptions, Tools and Applications (LDTA'10)*, pages 8:1-8:8, ACM, 2010.

  The full version of this paper has been published earlier as a research report:

  E.Tushkanova, A.Giorgetti, C. Marché, O. Kouchnarenko. Modular Specification of Java Programs. Research Report RR-7097, INRIA, 2009.

Chapter 2

Preliminary Notions

Contents

This chapter introduces the notions that are useful in this manuscript. We start by presenting the syntax of first-order logic, with the notions of signature, term, atom and formula. We continue by presenting the semantics of first-order logic, with the notions of

model, satisfiability and validity. Then we present some theories axiomatizing standard datatypes. After presenting some concepts related to rewrite systems and addressing the question of their termination, we present a well-known method for combination of theories proposed by Nelson and Oppen. Finally, a presentation of the Maude rewriting language, used for the implementation of schematic paramodulation calculi, with its important features such as reflection, unification and narrowing, ends this chapter.

2.1 Many-sorted first-order logic

2.1.1 Syntax of first-order logic

This section introduces the usual first-order syntactic notions of signature, term and formula. We find in this section, the definitions we use in the rest of this manuscript.

Definition 1 (Signature). *A first-order many-sorted signature Σ consists of*

- *a nonempty set of sorts S,*
- *a countable set of function symbols Σ^F whose arities are constructed using sorts that belong to S,*
- *a countable set of predicate symbols Σ^P whose arities are constructed using sorts that belong to S,*

A function declaration is of the form $f : s_1 \times \ldots \times s_n \to s$, where $f \in \Sigma^F$ is a function symbol, $n \geq 0$ is its arity, $s_1, \ldots, s_n$ and s are *sorts* from a finite set of sorts S. The sorts $s_1, \ldots, s_n$ are called the *argument sorts*, and s is called the *value sort* of f. Each sort is interpreted over a nonempty domain. For example the function car has the following declaration: $\mathsf{car} : \textsc{lists} \to \textsc{elem}$. The only predicates are equalities on sorts, denoted $=_s$ for each sort $s \in S$, whose type is $s \times s$. In what follows, we simply denote them $=$ when there is no risk of confusion. A functional symbol with arity 0 is called a *constant symbol*.

In all what follows, the signature Σ denotes Σ^F and the predicate $=_s$ $(s \in S)$ is denoted by $=$ when there is no risk of confusion.

Example 1. *In order to model lists, the following signature with the sorts* LISTS *and* ELEM *can be used:*

$\Sigma = \{\mathsf{cons} : \textsc{elem} \times \textsc{lists} \to \textsc{lists}, \mathsf{cdr} : \textsc{lists} \to \textsc{lists}, \mathsf{car} : \textsc{lists} \to \textsc{elem}\}$ □

Variables are also sorted and $x : s$ means that variable x has sort s. The set X_s denotes a set of variables of sort s generally supposed to be denumerable, and $X = \cup_{s \in S} X_s$ is the set of many-sorted variables. Many-sorted terms are built on many-sorted signatures and classified according to their sorts.

Definition 2 (Term). *A first-order term of sort $s \in S$*

- *is either a variable of sort s, or*
- *a constant of sort s, or*

- *has the form* $f(t_1, ..., t_n)$*, where* $f \in \Sigma^F$ *is a function symbol of arity* n *whose profile is* $s_1 \times \ldots \times s_n \to s$*, and* t_i *is a term of sort* s_i *for* $i = 1, .., n$*.*

A term is *ground* if it does not contain variables, i.e. the term $\mathsf{car}(x)$ is not ground if x is a variable, but $\mathsf{car}(b)$ is ground, if b is a constant of sort LISTS. If t is a term then $Var(t)$ denotes the set of variables occurring in t, e.g. $Var(\mathsf{cons}(a, x)) = \{x\}$, if x is a variable and a is a constant. This notation extends naturally to sets of terms. A subterm of a term t is a term that appears in t, e.g $\mathsf{car}(x)$ is a subterm of $\mathsf{cons}(\mathsf{car}(x), b)$. A *compound* term is an f-rooted term for a function symbol f of positive arity (> 0).

Definition 3 (Depth of a term)**.** *The depth of a term is defined inductively as follows:*

- $depth(t) = 0$ *if* t *is either a constant or a variable*
- $depth(f(t_1, \ldots, t_n)) = 1 + max\{depth(t_i) \mid 1 \leq i \leq n\}$

For instance, the depth of the term $\mathsf{cons}(a, b)$ equals to 1, and the depth of the term $\mathsf{cons}(\mathsf{car}(x), b)$ equals to 2. A term t may be viewed as a finite labeled tree, the leaves of which are labeled with variables or constants, and the internal nodes of which are labeled with symbols of positive arity.

A term is *flat* if its depth is 0 or 1. For example, if x is a variable of sort LISTS and a is a constant of sort ELEM then $\mathsf{car}(x)$ and a are two flat terms.

In order to access a subterm of a given term, it is convenient to introduce the concept of position in a term.

Definition 4 (Position and context of a term)**.** *A position (also called occurrence) within a term* t *is represented as a sequence* p *of positive integers describing the path from the root of* t *to the root of the subterm at that position, denoted by* $t_{|p}$*. The top position is denoted by* ε *and corresponds to the empty sequence. A term* u *has an occurrence in* t *if* $u = t_{|p}$ *for some position* p *in* t*. A context is a term with a distinguished position.*

The notation $t[s]_{|p}$ emphasizes that the term t contains s as subterm at position p. When the position p is clear from the context, we may simply write $t[s]$.

Example 2. *Let us consider the term* $\mathsf{cons}(\mathsf{car}(x), y)$*.*

- $\mathsf{cons}(\mathsf{car}(x), y)_{|\varepsilon} = \mathsf{cons}(\mathsf{car}(x), y)$
- $\mathsf{cons}(\mathsf{car}(x), y)_{|1} = \mathsf{car}(x)$
- $\mathsf{cons}(\mathsf{car}(x), y)_{|2} = y$

The term $\mathsf{car}(x)$ *is the subterm of the term* $\mathsf{cons}(\mathsf{car}(x), y)$ *at position* 1 *with context* $\mathsf{cons}([], y)$*.* □

Definition 5 (Substitution)**.** *A substitution is an application on* $T(\Sigma^F, X)$ *uniquely determined by its image of variables. It is thus written out as* $\{x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n\}$ *when there are only finitely many variables* $x_1, \ldots, x_n$ *not mapped to themselves, and* x_i *and* t_i *have the same sort for* $i = 1, \ldots, n$*. The application of a substitution* σ *to a term is recursively defined as follows:*

- *if t is a variable x_i for some $i = 1, \ldots, n$, then $\sigma(t) = t_i$,*
- *if t is a variable $x \neq x_i$ for all $i = 1, \ldots, n$, then $\sigma(t) = t$,*
- *if t is a term $f(u_1, \ldots, u_k)$ with $u_1, \ldots, u_k \in T(\Sigma^F, X)$ and $f \in \Sigma^F$, then $\sigma(t) = f(\sigma(u_1), \ldots, \sigma(u_k))$.*

Example 3. *Applying $\sigma = \{y \leftarrow \mathsf{car}(b)\}$ to the term $t = \mathsf{cons}(y, x)$, where y is a variable of sort* ELEM*, b is a constant of sort* LISTS *and x is a variable of sort* LISTS*, results in the term $\sigma(t) = \mathsf{cons}(\mathsf{car}(b), x)$.* □

If σ_1 and σ_2 are two substitutions, the composition $\sigma_1 \circ \sigma_2$ is defined by $\sigma_1 \circ \sigma_2(t) = \sigma_2(\sigma_1(t))$.

Definition 6 (Matching)**.** *A term t matches a term t' if there exists a substitution σ such that $\sigma(t) = t'$.*

Example 4. *The term $\mathsf{cons}(X, Y)$ matches the term $\mathsf{cons}(\mathsf{car}(a), \mathsf{cdr}(b))$ since*

$$\sigma(\mathsf{cons}(X, Y)) = \mathsf{cons}(\mathsf{car}(a), \mathsf{cdr}(b))$$

using the substitution $\sigma = \{X \leftarrow \mathsf{car}(a), Y \leftarrow \mathsf{cdr}(b)\}$. □

In matching the substitution is only applied to one of the terms. If the substitution is applied to both terms, then we get unification.

Definition 7. *Two terms t and t' are unifiable iff there exists a substitution σ such that $t\sigma = t'\sigma$. In this case, σ is a unifier of t and t'. A unifier σ is a most general unifier of t and t' if and only if for any unifier σ' of t and t' there exists a substitution δ such that $\sigma' = \sigma\delta$.*

Example 5. *The terms $\mathsf{cons}(\mathsf{car}(a), Y)$ and $\mathsf{cons}(X, \mathsf{cdr}(b))$ are unifiable with the unifier $\sigma = \{X \leftarrow \mathsf{car}(a), Y \leftarrow \mathsf{cdr}(b)\}$.* □

Definition 8 (Atom)**.** *A first-order atom is an equality $s = t$, where s and t are two terms of the same sort.*

A literal is an atom or the negation of an atom. A *positive literal* is an equality $s = t$ and a *negative literal* is a disequality $s \neq t$. In all that follows, $\bowtie$ stands for $=$ or $\neq$.

Definition 9 (Depth of a literal)**.** *The depth of a literal $s \bowtie t$ is defined as*

$$\mathit{depth}(s \bowtie t) = \mathit{depth}(s) + \mathit{depth}(t).$$

A positive literal is *flat* if its depth is 0 or 1. A negative literal is *flat* if its depth is 0.

Definition 10 (Clause)**.** *A clause is a finite disjunction of literals $l_1 \vee \ldots \vee l_n$, where l_i is a literal for each $1 \leq i \leq n$.*

An example of a clause can be the following disjunction of literals: $(c_1 \neq c_2 \vee \mathsf{car}(v) = e)$. A *unit clause* is a clause composed of exactly one literal. For example, $(c_1 = c_2)$ is a unit clause. A clause can be empty. In this case, it is an empty set of literals. The empty clause is denoted by $\bot$.

Definition 11 (Formula). *A first-order formula satisfies the following properties:*

- *each atom is a formula,*
- *if φ is a formula, then $\neg\varphi$ is a formula,*
- *if φ and ψ are two formulae and υ is a variable, then $\varphi \wedge \psi$, $\varphi \vee \psi$, $\exists \upsilon.\varphi$, $\forall \upsilon.\varphi$ are formulae.*

Definition 12 (Free variable). *A free variable is a variable that is not bound by universal ($\forall$) or existential ($\exists$) quantifiers. If φ is a formula, $Var(\varphi)$ denotes the set of free variables in φ.*

For example, $\forall x.P(x, y)$ has x bound by a universal quantifier and y is a free variable.

Definition 13 (Quantifier-free formula). *A quantifier-free formula is a formula in which no quantifier occurs.*

Definition 14 (Sentence). *A Σ-sentence is a Σ-formula with no free variables.*

Definition 15 (First-order theory). *A first-order theory (over a finite signature) is a set of sentences.*

When T is a finitely axiomatized theory, $Ax(T)$ denotes the set of axioms of T.

2.1.2 Semantics of first-order logic

In this section we present the semantics of first-order logic. We start by introducing the notion of structure, then we present the relation between structures and formulae, e.g. satisfiability and validity.

Definition 16 (Structure). *Let Σ be a signature. A Σ-structure $\mathcal{A}$ is a pair (A, I^Σ) such that $A = (A_s)_{s \in S}$ is the domain of $\mathcal{A}$ and I^Σ is a function that associates*

- *each sort $s \in S$ with a nonempty domain A_s,*
- *each function symbol $f \in \Sigma^F$ of arity $s_1 \times \ldots \times s_n \to s$ with a function $f^A : A_{s_1} \times \ldots \times A_{s_n} \to A_s$,*
- *each predicate symbol $p \in \Sigma^P$ of arity $s_1 \times \ldots \times s_n$ is mapped to a subset $p^A \subseteq A_{s_1} \times \ldots \times A_{s_n}$.*

Definition 17 (Valuation). *Let $\mathcal{A} = (A, I^\Sigma)$ be a Σ-structure and X be a set of variables. A valuation of X is an application from X to A such that any variable x of sort s is mapped to an element in A_s.*

Definition 18 (Evaluation)**.** *Let $\mathcal{A} = (A, I^{\Sigma})$ be a Σ-structure, X be a set of variables, φ be a Σ-formula such that $Var(\varphi) \subseteq X$, and $\alpha : X \to A$ be a valuation. The evaluation of φ is recursively defined as follows:*

- $\alpha(f(t_1, \ldots, t_n)) = f^A(\alpha(t_1), \ldots, \alpha(t_n))$,
- $\alpha(p(t_1, \ldots, t_n)) = \top$ *iff* $(\alpha(t_1), \ldots, \alpha(t_n)) \in p^A$,
- $\alpha(\varphi_1 \wedge \varphi_2) = \alpha(\varphi_1) \wedge \alpha(\varphi_2)$,
- $\alpha(\neg\varphi) = \neg\alpha(\varphi)$,
- $\alpha(\exists x : \varphi) = \top$ *iff there exists a valuation* α' *such that* $\alpha'(\varphi) = \top$, *where* $\alpha'(v) = \alpha(v)$ *for any* $v \in X \backslash \{x\}$,
- $\alpha(\top) = \top$.

Definition 19 (Satisfiability and validity)**.** *Let $\mathcal{A}$ be a Σ-structure and φ be a Σ-formula. We say that*

- *a valuation α of $Var(\varphi)$ in A satisfies φ in $\mathcal{A}$, in other words, φ is satisfiable in $\mathcal{A}$, written $(\mathcal{A}, \alpha) \models \varphi$, if $(\mathcal{A}, \alpha)$ evaluates φ to $\top$, and*
- *$\mathcal{A}$ is a model of φ, in other words, φ is valid in $\mathcal{A}$, written $\mathcal{A} \models \varphi$, if $(\mathcal{A}, \alpha) \models \varphi$ for all the valuations α of $Var(\varphi)$ in A.*

Definition 20. *A Σ-structure $\mathcal{A}$ is a model of some Σ-theory T if A is a model for all the sentences of T.*

Definition 21 (Satisfiability modulo)**.** *Let T be a Σ-theory. A Σ-formula φ is T-satisfiable if φ is satisfiable in some model of T.*

Let S be a set of ground literals, then we say that S is T-satisfiable (T-unsatisfiable) if and only if $T \cup S$ is satisfiable (resp. unsatisfiable). The satisfiability problem for a theory T is the problem of determining whether any given finite set of ground literals is T-satisfiable or not.

Definition 22 (Satisfiability procedure)**.** *A satisfiability procedure for a theory T is any algorithm that solves the satisfiability problem for T.*

Definition 23 (Validity modulo)**.** *Let T be a Σ-theory. A Σ-formula φ is T-valid if φ is valid in all the models of T.*

2.2 Examples of theories of classical datatypes

Satisfiability procedures for theories of standard datatypes are at the core of most state-of-the-art verification tools. They are required for a wide range of verification tasks and are fundamental for efficiency. As mentioned above, satisfiability problems have the form $T \cup S$, where S is a set of ground flat literals, T is a background theory, and the goal is to

prove that $T \cup S$ is unsatisfiable. A satisfiability procedure for a theory T is an algorithm capable of checking whether $T \cup S$ is satisfiable or not, for any finite set S of ground flat literals. We present in this section the theories that are widely used in verification. Those are theories of classical datatypes such as lists, arrays and records.

2.2.1 Theory of lists

The many-sorted signature Σ_L of the theory of lists is the set of function symbols $\{\mathsf{car} : \text{LISTS} \to \text{ELEM}, \mathsf{cdr} : \text{LISTS} \to \text{LISTS}, \mathsf{cons} : \text{ELEM} \times \text{LISTS} \to \text{LISTS}\}$.

This theory is axiomatized by the following set of axioms $Ax(L)$:

$$\begin{array}{lcl} \mathsf{car}(\mathsf{cons}(X,Y)) & = & X \\ \mathsf{cdr}(\mathsf{cons}(X,Y)) & = & Y \\ \mathsf{cons}(\mathsf{car}(Y),\mathsf{cdr}(Y)) & = & Y (\mathit{extensionality}) \end{array}$$

where X is a universally quantified variable of sort ELEM and Y is a universally quantified variable of sort LISTS.

2.2.2 Theory of lists with length

The many-sorted signature Σ_{LLI} of the theory of lists with length is the set of function symbols $\{\mathsf{car} : \text{LISTS} \to \text{ELEM}, \mathsf{cdr} : \text{LISTS} \to \text{LISTS}, \mathsf{cons} : \text{ELEM} \times \text{LISTS} \to \text{LISTS}, \mathsf{len} : \text{LISTS} \to \text{INT}, \mathsf{nil} :\to \text{LISTS}, \mathsf{0} :\to \text{INT}, \mathsf{s} : \text{INT} \to \text{INT}\}$.

This theory is axiomatized by the following set of axioms $Ax(LLI)$:

1. Axioms for lists
   a) $\mathsf{car}(\mathsf{cons}(X,Y)) = X$
   b) $\mathsf{cdr}(\mathsf{cons}(X,Y)) = Y$
   c) $\mathsf{cons}(X,Y) \neq \mathsf{nil}$
2. Axiom for the length
   a) $\mathsf{len}(\mathsf{cons}(X,Y)) = \mathsf{s}(\mathsf{len}(Y))$
   b) $\mathsf{len}(\mathsf{nil}) = \mathsf{0}$

where X is a universally quantified variable of sort ELEM and Y is a universally quantified variable of sort LISTS.

2.2.3 Theory of records

A record is an array with a fixed enumerated set of elements. Contrary to the theory of arrays, the theory of records can be specified by unit clauses. We consider here the theory of records of length 3 without extensionality. It is defined by the many-sorted signature $\Sigma_{Rec} = \bigcup_{i=1}^{3} \{\mathsf{rstore}_i : \text{REC} \times \text{T}_i \to \text{REC}, \mathsf{rselect}_i : \text{REC} \to \text{T}_i\}$, and axiomatized by the following set of axioms $Ax(Rec)$

$$\begin{array}{lcll} \mathsf{rselect}_i(\mathsf{rstore}_i(X,Y)) & = & Y & \text{for } i \in \{1,2,3\} \\ \mathsf{rselect}_i(\mathsf{rstore}_j(X,Y)) & = & \mathsf{rselect}_i(X,Y) & \text{for } i,j \in \{1,2,3\} \text{ with } i \neq j \end{array}$$

where X is a universally quantified variable of sort REC, and Y is a universally quantified variable of sort INT.

2.2.4 Theory of records with increment

The many-sorted signature Σ_{RII} of the theory of records with increment is the set of function symbols $\bigcup_{i=1}^{3}\{\mathsf{rstore}_i : \text{REC} \times \text{INT} \to \text{REC}, \mathsf{rselect}_i : \text{REC} \to \text{INT}, \mathsf{incr} : \text{REC} \to \text{REC}, \mathsf{s} : \text{INT} \to \text{INT}\}$.

This theory is axiomatized by the following set $Ax(RII)$ of axioms:

$$\begin{array}{rcl} \mathsf{rselect}_i(\mathsf{rstore}_i(X,Y)) & = & Y \text{ for } i \in \{1,2,3\} \\ \mathsf{rselect}_i(\mathsf{rstore}_j(X,Y)) & = & \mathsf{rselect}_i(X) \text{ for } i,j \in \{1,2,3\} \text{ with } i \neq j \\ \mathsf{rselect}_i(\mathsf{incr}(X)) & = & \mathsf{s}(\mathsf{rselect}_i(X)) \text{ for } i \in \{1,2,3\} \end{array}$$

where X is a universally quantified variable of sort REC, and Y is a universally quantified variable of sort INT.

2.2.5 Theory of arrays

The many-sorted signature Σ_A of the theory of arrays is the set of function symbols $\{\mathsf{select} : \text{ARRAY} \times \text{INDEX} \to \text{ELEM}, \mathsf{store} : \text{ARRAY} \times \text{INDEX} \times \text{ELEM} \to \text{ARRAY}\}$.

This theory is axiomatized by the following set of axioms:

$$\begin{array}{rcl} \mathsf{select}(\mathsf{store}(V,I,E),I) & = & E \\ \mathsf{select}(\mathsf{store}(V,I,E),J) & = & \mathsf{select}(V,J) \vee I = J \end{array}$$

where V is a universally quantified variable of sort ARRAY, I, J are universally quantified variables of sort INDEX, and E is a universally quantified variable of sort ELEM.

2.3 Rewriting

Since our results are based on rewriting, we introduce the classical notions of rewrite systems. Also we present a method for proving termination of term rewriting systems based on simplification ordering. More details on the notions described in this section could be found in [DJ90, BN98, KK06].

2.3.1 Rewrite system

The main idea of rewriting is to impose directionality in the use of equalities.

Definition 24 (Rewrite rule)**.** *A rewrite rule is an ordered pair of terms denoted $l \to r$ such that $Var(r) \subseteq Var(l)$. The terms l and r are respectively called the left-hand side and the right-hand side of the rule.*

A *rewrite system* is a (finite or infinite) set of rewrite rules. A rule is applied by replacing an instance of the left-hand side by the same instance of its right-hand side, but never the converse, contrary to equalities. Note that two rules are considered to be the same if they only differ by a renaming of their variables. A rewrite system R induces a binary relation on terms called the *rewriting relation.*

Definition 25. *Given a rewrite system R, a term t rewrites to a term t', which is denoted by $t \to_R t'$, if there exist*

- *a rule $l \to r$ of R,*
- *a position p in t,*
- *a substitution σ, satisfying $t[s]_{|p} = \sigma(l)$ and called a* match *from l to $t[s]_{|p}$, where s is a subterm of t at position p,*

such that $t' = t[s \leftarrow \sigma(r)]_{|p}$.

When t rewrites to t' with a rule $l \to r$ and a substitution σ, it will be always assumed that the variables of l and t are disjoint. We say that there is a rewriting step $t \to_R t'$ where t rewrites to t' by a rule of R. A term is in *normal form* if it cannot be further rewritten.

Definition 26 (Rewriting derivation)**.** *A rewriting derivation is any sequence of rewriting steps*

$$t_1 \to_R t_2 \to_R \dots$$

The rewriting relation $\to_R^*$ is defined on terms by $t \to_R^* t'$ if there exists a rewriting derivation from t to t'. If the derivation contains at least one step, it is denoted by $\to_R^+$.

Definition 27 (Termination of binary relation)**.** *A binary relation $\to$ is terminating if there is no infinite derivation $t_1 \to t_2 \to t_3 \to \dots$.*

It is *terminating* for a set T of elements if there is no infinite derivation with $t_1 \in T$.

Definition 28 (Confluent relation)**.** *A relation $\to$ is* confluent *if there is an element v such that $s \to^* v$ and $t \to^* v$ whenever $u \to^* s$ and $u \to^* t$ for some elements s,t and u.*

Definition 29 (Convergent rewrite system)**.** *A rewrite system R is* convergent *if it is terminating and confluent.*

In convergent rewrite systems all derivations lead to a unique normal form. Such systems are used to decide equational theories. The clause obtained from a clause C by replacing the terms occurring in C with their normal forms w.r.t. a convergent rewrite system R is denoted $C\downarrow_R$.

2.3.2 Ordering for termination of rewrite systems

One of the main problems with term rewrite systems is to detect their termination, i.e. to determine whether there exists an infinite rewriting derivation or not. A well-known method for proving termination has been proposed by Dershowitz [Der82]. This method is based on recursive path orderings which are known to be simplification ones. One of the families of recursive path orderings is based on sequences, and is named lexicographic path orderings. This ordering is used in the side conditions of the inference rules of paramodulation calculus.

After introducing some preliminary notions, we present the definition of lexicographic path ordering and its lexicographic and multiset extensions.

Definition 30 (Binary relation)**.** *A binary relation R on a set T is:*

- *reflexive if $\forall x \in T$, xRx,*
- *antisymmetric if $\forall x, y \in T$, xRy and $yRx \Rightarrow x = y$,*
- *transitive if $\forall x, y, z \in T$, xRy and $yRz \Rightarrow xRz$,*
- *a* quasi-ordering *if it is reflexive and transitive and in this case, (T, R) is called a* quasi-ordered *set,*
- *an* ordering *if it is reflexive, antisymmetric and transitive, such an ordering is also called a partial ordering and (T, R) is called a* poset*,*
- *a* total (quasi-) ordering *if it is a (quasi-) ordering and $\forall x, y \in T$, xRy or yRx.*

A quasi-ordering is sometimes called a *pre-ordering*. The equivalence relation associated with a quasi-ordering $\geq$ on a set T is denoted $\approx_{\geq}$ and defined as the intersection of $\geq$ and of its symmetric relation $\leq$, i.e.

$$\forall x, y \in T, x \approx_{\geq} y \Leftrightarrow x \geq y \text{ and } y \geq x.$$

The associated *strict* ordering $>$ is defined by:

$$t > t' \text{ if } t \geq t' \text{ and } t \not\approx_{\geq} t'.$$

Definition 31 (Well-founded ordering)**.** *A quasi-ordered set $(T, \geq)$ is well-founded if there exists no infinite decreasing sequence $t_1 > t_2 > \ldots$ of elements of T.*

Definition 32 (Reduction ordering)**.** *An ordering $>$ is a* reduction ordering *if it is a well-founded ordering closed under context and substitution, that is such that for any context $C[\_]$ and any substitution σ, if $t > s$ then $C[t] > C[s]$ and $\sigma(t) > \sigma(s)$.*

Definition 33 (Simplification ordering)**.** *An ordering $>$ on terms is a* simplification ordering *if it is*

- *stable (by instantiation), i.e. $l > r$ implies $\sigma(l) > \sigma(r)$ for every substitution σ,*
- *monotonic, i.e. $l > r$ implies $t[l]_p > t[r]_p$ for every term t and position p, and*
- *has the subterm property, i.e. it contains the subterm ordering: if r is a strict subterm of l, then $l > r$.*

Simplification orderings are well-founded for finite signatures and could be built from a well-founded ordering on the set Σ^F of function symbols, called a *precedence* and denoted by $>_F$.

The lexicographic path ordering (LPO) has been defined by an inference system in [DJ90].

Definition 34 (Lexicographic path ordering)**.** *Given a precedence $>_F$ on function symbols, the* ***lexicographic path ordering*** *$>_{lpo}$ is defined as follows:*

$$LPO1 \qquad \frac{(s_1, \ldots, s_n) >_{lpo}^{lex} (t_1, \ldots, t_m) \qquad f(s_1, \ldots, s_n) >_{lpo} t_1, \ldots, t_m}{f(s_1, \ldots, s_n) >_{lpo} f(t_1, \ldots, t_m)}$$

$$LPO2 \qquad \frac{f \;>_F\; g \qquad f(s_1, \ldots, s_n) >_{lpo} t_1, \ldots, t_m}{f(s_1, \ldots, s_n) >_{lpo} g(t_1, \ldots, t_m)}$$

$$LPO3 \qquad \frac{u_k \;>_{lpo}\; t}{f(u_1, \ldots, u_k, \ldots, u_p) >_{lpo} t}$$

$$LPO4 \qquad \frac{}{f(u_1, \ldots, u_k, \ldots, u_p) >_{lpo} u_k}$$

where f and g are two functional symbols, $n \geq 0$ and $m \geq 0$ are two non-negative integers, $p \geq 1$ is a positive integer, and $s_1, \ldots, s_n, t_1, \ldots, t_m, u_1, \ldots, u_p, t$ are terms. We write $s >_{lpo} t_1, \ldots, t_m$ when $s >_{lpo} t_k$ for $k = 1, \ldots, m$.

The ordering $>_{lpo}^{lex}$ denotes the lexicographic extension of $>_{lpo}$ which is defined by the following inference system:

$$(1) \qquad \frac{}{(s_1, \ldots, s_n) >_{lpo}^{lex} ()}$$

$$(2) \qquad \frac{s_1 \;>_{lpo}\; t_1}{(s_1, \ldots, s_n) >_{lpo}^{lex} (t_1, \ldots, t_m)}$$

$$(3) \qquad \frac{s_1 \;=\; t_1 \qquad (s_2, \ldots, s_n) >_{lpo}^{lex} (t_2, \ldots, t_m)}{(s_1, \ldots, s_n) >_{lpo}^{lex} (t_1, \ldots, t_m)}$$

where $n \geq 1$ and $m \geq 1$ are two positive integers, and $s_1, \ldots, s_n, t_1, \ldots, t_m$ are terms.

An ordering on terms is extended to literals by using its multiset extension on literals viewed as multisets of terms. Actually, a multiset (a bag) is a generalization of the notion of set in which elements are allowed to appear more than once without order.

Example 6. *In the multiset $[a, b, b]$ the element b appears twice, while the set would be $\{a, b\}$.* □

Any positive literal $l = r$ (resp. negative literal $l \neq r$) is viewed as a bag $[l, r]$ (resp. $[l, l, r, r]$).

Definition 35 (Multiset extension)**.** *The multiset extension $>_{bag}$ is defined by the following inference system:*

$$(1) \qquad \frac{}{[\,] =_{bag} [\,]}$$

$$(2) \qquad \frac{[s_1, \ldots, s_n] \;=_{bag}\; [t_1, \ldots, t_n] \qquad s = t}{[s_1, \ldots, s_n, s] \;=_{bag}\; [t_1, \ldots, t_n, t]}$$

$$(3) \qquad \frac{[s_1, \ldots, s_m] \geq_{bag} [t_1, \ldots, t_n] \qquad s \;>\; u_1, \ldots, u_k}{[s_1, \ldots, s_m, s] \;>_{bag}\; [t_1, \ldots, t_n, u_1, \ldots, u_k]}$$

where $=$ *stands for syntactic equality,* $m \geq 0$, $n \geq 0$, $s > u_1, \ldots, u_k$ *holds either if* $k = 0$ *or if* $s > u_i$ *for all* $1 \leq i \leq k$, *and* $\geq_{bag}$ *is the union of the equivalence relation* $=_{bag}$ *and the partial ordering* $>_{bag}$.

If $>$ is well-founded than its multiset extension $>_{bag}$ is well-founded on finite multiset.

Let us first consider the equality relation $=_{bag}$. Two bags are equal when they contain the same elements with the same multiplicity. The former definition decomposes this condition in two cases:

- Either both bags are empty,
- or some element s from the first bag is equal to an element t in the second one, and their subbags $[s_1, \ldots, s_n]$ and $[t_1, \ldots, t_n]$ obtained by removal of s $(= t)$ are also equal.

Example 7. *The bag* $[2, 3, 3, 4]$ *and the bag* $[4, 3, 1, 3]$ *are the same.* □

One bag is bigger than another one (by $>_{bag}$) if there is an element s in the first bag that is greater than each element in some subbag $[u_1, \ldots, u_k]$ from the second bag, and the remaining subbag $[s_1, \ldots, s_n]$ from the first bag is bigger or equal to the remaining subbag $[t_1, \ldots, t_n]$ from the second bag.

Example 8. *Let us consider the following examples:*

- $[2, 3, 4] >_{bag} []$, *since the second bag is empty.*
- $[2, 2, 3, 4] >_{bag} [2, 4]$, *since* s *represented as* 3 *(resp.* 2*) is greater than* $u_1, \ldots, u_k$ *with* $k = 0$ *and the remaining subbag* $[2, 4]$ *and* $[2, 4]$ *are equal.*
- $[3, 5] >_{bag} [3, 4, 4, 1]$, *since* 5 *is greater than each element in the subbag* $[4, 4, 1]$, *and the remaining subbags* $[3]$ *and* $[3]$ *are equal.*
- $[3, 3, 4, 5] >_{bag} [3, 2, 2, 2, 4, 5]$, *since* 3 *is greater than each element in the subbag* $[2, 2, 2]$, *and the remaining subbags* $[3, 4, 5]$ *and* $[3, 4, 5]$ *are equal.*

□

Since the ordering on terms is performed by the lexicographic path ordering, then the ordering on literals is performed by the multiset extension of LPO.

LPO is a reduction ordering for systems having fixed-arity function symbols if the condition that non-constant symbols are greater than constants is satisfied. In 1980 it has been shown [KL80] that LPO is a simplification ordering. LPO is total on ground terms.

2.4 Combinability

An important question in automated reasoning is whether two decision procedures for two theories T_1 and T_2 can be combined into a single decision procedure for T_1 and T_2.

The most popular method for combining decision procedures for disjoint theories has been proposed by Nelson and Oppen in 1979 [NO79]. This combination method allows deciding the satisfiability of a conjunction ϕ of ground literals in the union of two signature-disjoint theories T_1 and T_2 such that a T_i-satisfiability procedure is available, for $i = 1, 2$.

The following notion is used to show the soundness of the Nelson-Oppen method.

Definition 36 (Stably infinite/finite theory)**.** *A Σ-theory T is stably infinite (resp. stably finite) if every quantifier-free formula φ is T-satisfiable if and only if it is satisfied by a T-interpretation $\mathcal{A}$ whose domain A is infinite (resp. finite).*

The Nelson-Oppen method is correct whenever the following restrictions are satisfied by the theories T_1 and T_2:

- T_1 and T_2 are two signature-disjoint theories, and
- T_1 and T_2 are both stably infinite theories.

The theory of lists (Section 2.2.1) and the theory of arrays (Section 2.2.5) are stably infinite theories.

In fact there are two versions of the Nelson-Oppen method: a nondeterministic one and a deterministic one. Let us present the nondeterministic version on the example, where T_1 is the theory of lists (Section 2.2.1) and T_2 is the theory of arrays (Section 2.2.5), and the deterministic one on the example, where T_1 is the theory of lists and T_2 is the theory of records (Section 2.2.3). Here, all the theories are unsorted, i.e. their sorts are ignored.

2.4.1 Nondeterministic version of the N.-O. combination method

The nondeterministic version of the Nelson-Oppen combination method is consists of two steps:

1. *Purification.* This step consists in term flattening and converting ϕ into a conjunction $\phi_1 \cup \phi_2$, where ϕ_i contains only Σ_i-literals. The flattening is done by replacing each subterm t of a term by a fresh variable X and adding the equality $X = t$ to ϕ. When no term can be flatten, ϕ is converted into a conjunction $\phi_1 \cup \phi_2$.

   We consider the following conjunction ϕ of literals:

   $$\mathsf{car}(x_1) = x_3 \wedge \mathsf{car}(x_2) = x_4 \wedge x_1 = x_2 \wedge \mathsf{select}(x_5, \mathsf{car}(x_1)) \neq \mathsf{select}(x_5, x_4)$$

   The function symbol car (resp. select) comes from the theory of lists (resp. of arrays).

Let us now apply the purification to the conjunction of literals ϕ. Firstly, we flatten the term $\mathsf{select}(x_5, \mathsf{car}(x_1))$ by replacing the subterm $\mathsf{car}(x_1)$ by a fresh variable y_1. We obtain the new conjunction

$$\left\{ \begin{array}{l} \mathsf{car}(x_1) = x_3, \\ \mathsf{car}(x_2) = x_4, \\ x_1 = x_2, \\ \mathsf{select}(x_5, y_1) \neq \mathsf{select}(x_5, x_4), \\ y_1 = \mathsf{car}(x_1) \end{array} \right\}$$

Then, we flatten the disequality by replacing the right-hand side (resp. left-hand side) term of the disequality by a fresh variable y_2 (resp. y_3), and obtain

$$\left\{ \begin{array}{l} \mathsf{car}(x_1) = x_3, \\ \mathsf{car}(x_2) = x_4, \\ x_1 = x_2, \\ y_2 \neq y_3, \\ y_1 = \mathsf{car}(x_1), \\ y_2 = \mathsf{select}(x_5, y_1), \\ y_3 = \mathsf{select}(x_5, x_4) \end{array} \right\}$$

Since no more terms can be flattened, we obtain the conjunction of literals for each theory:

$$\phi_1 = \left\{ \begin{array}{l} \mathsf{car}(x_1) = x_3, \\ \mathsf{car}(x_2) = x_4, \\ x_1 = x_2, \\ y_1 = \mathsf{car}(x_1) \end{array} \right\} \qquad (2.1) \qquad \phi_2 = \left\{ \begin{array}{l} y_2 \neq y_3, \\ y_2 = \mathsf{select}(x_5, y_1), \\ y_3 = \mathsf{select}(x_5, x_4) \end{array} \right\} \qquad (2.2)$$

2. *Check.* In this step we use all the ways of identifying/differentiating the shared variables. For that purpose, we consider the equivalence relations on shared variables.

   Our example contains two shared variables $\{x_4, y_1\}$, and therefore there are only two possible equivalence relations:

   - if x_4 and y_1 are not equivalent, then we consider the shared disequality $x_4 \neq y_1$, and check whether $\phi_1 \wedge x_4 \neq y_1$ is T_1-satisfiable. From three first equalities of φ_1 we get $x_3 = x_4$. From the first equality and the last one of φ_1 we get $y_1 = x_3$. From $x_3 = x_4$ and $y_1 = x_3$ we can conclude that $y_1 = x_4$. Thus we obtain a contradiction. Therefore, $\phi_1 \wedge x_4 \neq y_1$ is T_1-unsatisfiable.
   - if x_4 and y_1 are equivalent, then we consider the shared equality $x_4 = y_1$, and check whether $\phi_2 \wedge x_4 = y_1$ is T_2-satisfiable. If $y_1 = y_4$, then $y_2 = y_3$, that contradicts the equality $y_2 \neq y_3$. Therefore, $\phi_2 \wedge x_4 = y_1$ is T_2-unsatisfiable.

   Thus, in all cases we cannot have simultaneously the T_1-satisfiability and the T_2-satisfiability. Therefore, we conclude that ϕ is $(T_1 \cup T_2)$-unsatisfiable.

Due to the fact that a number of equivalence relations of a set grows exponentially in the number of elements of the set, it is not effective to use the nondeterministic version of this method in the implementation. In the following section, we describe the deterministic version of the Nelson-Oppen method.

2.4.2 Deterministic version of the N.-O. combination method

In this version the considered theories should be not only stable-infinite (see Definition 36), but also convex.

Definition 37 (Convex theory, [MZ02]). *A Σ-theory T is convex if for every conjunction ϕ of Σ-literals and for every disjunction $\bigvee_{i=1}^{n} x_i = y_i$,*

$$T \cup \phi \vDash \bigvee_{i=1}^{n} x_i = y_i \text{ iff } T \cup \phi \vDash x_j = y_j \text{ from some } j \in \{1, .., n\}.$$

The theory of lists (Section 2.2.1) and the theory of records (Section 2.2.3) are convex theories, while the theory of arrays (Section 2.2.5) is a non-convex one [MZ02], since in this theory, the conjunction $\{\mathsf{select}(\mathsf{store}(a, i, e), j) = x, \mathsf{select}(a, j) = y\}$ entails a disjunction $x = e \vee x = y$ but does not entail neither $x = e$ nor $x = y$.

In this method the enumeration of all possible equivalence relations among shared variables is replaced with *propagation*. The propagation lasts until the unsatisfiability is returned or no more equalities can be propagated.

Let us consider the following conjunction ϕ of literals:

$$\mathsf{car}(x_1) = x_3 \wedge \mathsf{car}(x_2) = x_4 \wedge x_1 = x_2 \wedge \mathsf{rselect}_1(\mathsf{car}(x_1)) \neq \mathsf{rselect}_1(x_4)$$

The function symbol car (resp. $\mathsf{rselect}_1$) comes from the theory of lists (resp. of records).

After purification ϕ_1 and ϕ_2 are as follows:

$$\phi_1 = \left\{ \begin{array}{l} \mathsf{car}(x_1) = x_3, \\ \mathsf{car}(x_2) = x_4, \\ x_1 = x_2, \\ y_1 = \mathsf{car}(x_1) \end{array} \right\} \quad (2.3) \qquad \phi_2 = \left\{ \begin{array}{l} y_2 \neq y_3, \\ y_2 = \mathsf{rselect}_1(y_1), \\ y_3 = \mathsf{rselect}_1(x_4) \end{array} \right\} \quad (2.4)$$

The next step is propagation. Thus, ϕ_1 propagates $y_1 = x_4$, and since this literal does not exist in ϕ_2, we add it to ϕ_2:

$$\phi_1 = \left\{ \begin{array}{l} \mathsf{car}(x_1) = x_3, \\ \mathsf{car}(x_2) = x_4, \\ x_1 = x_2, \\ y_1 = \mathsf{car}(x_1) \end{array} \right\} \quad (2.5) \qquad \phi_2 = \left\{ \begin{array}{l} y_2 \neq y_3, \\ y_2 = \mathsf{rselect}_1(y_1), \\ y_3 = \mathsf{rselect}_1(x_4) \\ y_1 = x_4 \end{array} \right\} \quad (2.6)$$

Then, $\phi_2 \cup \{y_1 = x_4\}$ is T_2-unsatisfiable, since there is a contradiction in ϕ_2. Therefore, we can conclude that ϕ is $(T_1 \cup T_2)$-unsatisfiable.

The Nelson-Oppen combination method requires a way to compute the entailed elementary equalities between shared variables. By default, we can proceed by refutation: an elementary equality $x = y$ is entailed by φ modulo T if $\varphi \wedge x \neq y$ is T-unsatisfiable. This is not efficient since we have to guess all possible elementary equalities and to call the T-satisfiability procedure for each of them. Another solution is to consider a satisfiability procedure which is able to deduce automatically all the elementary equalities entailed by a T-satisfiable formula φ. The paramodulation calculus (presented in Chapter 3) is an interesting candidate to build *deduction-complete* satisfiability procedures, by collecting all equalities between shared variables occurring in the saturation.

Definition 38 ([TRRK10])**.** *Let T be a convex theory and ϕ be a T-satisfiable set of literals. A set of elementary equalities (equalities between variables) E is deduction complete (for ϕ modulo T) if*

$$\forall x, y \in Var(\phi),\ T \vDash \phi \Rightarrow x = y\ iff\ E \vDash x = y$$

A deduction complete T-satisfiability procedure is a T-satisfiable procedure, such that if ϕ is T-satisfiable, then it returns $false$, otherwise it returns $true\{E\}$ where E is deduction complete for ϕ modulo T.

2.5 Maude language

Maude [CDE+03] is a rule-based language. Maude's basic programming statements are equations and rules. Its semantics is based on rewriting logic where terms are reduced by applying rewrite rules. Maude has many important features such as reflection, pattern-matching, unification and narrowing. Reflection is a very desirable property of a computational system, because a reflective system can access its own meta-level and this way can be much more powerful, flexible and adaptable than a non-reflective one. Maude's language design and implementation make systematic use of the fact that rewriting logic is reflective [CM96b, CM96a]. Narrowing [CDE+09] is a generalization of term rewriting that allows free variables in terms (as in logic programming) and replaces pattern-matching by unification in order to (non-deterministically) instantiate and reduce a term. The narrowing feature is provided in an extension of Maude named Full Maude. All the notions described below are extracted from [CDE+03].

2.5.1 Maude specifications

In Maude the basic units of specification and programming are called *modules*. A module consists of syntax declarations, providing an appropriate language to describe the system under study, and of statements asserting properties of this system. There are two kinds of modules: functional module and system module.

Functional module A functional module is an equational theory with initial algebra semantics that specifies one or more data types and operations. It has the following syntax:

```
fmod MODULE-NAME is
  declarations
endfm
```

Declarations include the importation of other functional modules discussed below, and sort, subsort, and operator declarations. Types are declared with the `sort` keyword followed by an identifier (the sort name).

```
sort SortName .
```

A period at the end of the sort declaration, as for the other types of declarations, is crucial. Multiple sorts may be declared using the `sorts` keyword:

```
sorts SortName1 ... SortNameN .
```

Sorts can be partially ordered via a *subsort* relation. Subsort inclusions are declared using the keyword `subsort`. The declaration

```
subsort SortName1 < SortName2 .
```

states that the first sort `SortName1` is a subsort of the second one `SortName2`.

Functions are declared by `op` declarations

```
op OpName : SortName1 ... SortNameN -> SortName [OperatorAttributes] .
```

where `SortName1`, ..., `SortNameN` and `SortName` are sorts. If the argument list is empty, then the operator is called a *constant* of sort `SortName`. A function can be declared in prefix or mixfix form. If the operator is in mixfix form, then n underscores ('`_`') indicate the place where arguments of the n sorts must be placed in expressions formed with this operator. For example, the binary operator

```
op _ plus _ : Nat Nat -> Nat .
```

is in mixfix form.

Function declarations may include attributes that provide additional information about the operator: semantic, syntactic, pragmatic, etc. For example, Maude supports the following equational attributes: `assoc` (associativity), `comm` (commutativity), `idem` (idempotency), and `memo`. The last one allows to memoize the results of equational simplification (the canonical forms) in the memoization table for this operator. Whenever the Maude interpreter encounters a subterm whose top operator has the `memo` attribute, it looks to see if its canonical form is already stored. If so, that result is used; otherwise, equational simplification proceeds according to the operator's strategy.

A functional module can contain equations and variable declarations. Variables can be declared on-the-fly in Maude with syntax consisting of an identifier (the variable name), a colon, and another identifier (its sort). For example, `N:SortName`. It can also be declared in a module using the keyword `var`, for example, `var N : SortName .`

Equations may be either unconditional or conditional:

```
eq Term1 = Term2 .
ceq Term1 = Term2 if cond .
```

A condition (`cond`) can be either a single ordinary equation $t = t'$ or a single matching equation $t := t'$, or a conjunction of these equations by using the binary conjunction connective `/\` which is assumed to be associative. An example of a matching equation as a condition of conditional equation can be the following operator

```
sorts List Elem .
vars L L' : List .
var E : Elem .
op car : List -> Elem
ceq car(L) = E if cons(E,L') := L .
```

that defines the function `car` that extracts the first element of a list. The matching equation `cons(E,L') := L` expresses that `L`, used in the left-hand side of the equation, has to be of the form `cons(E,L')`, thus connecting the first element of `L` to `E`, which is used in the right-hand side of the equation.

Equational specifications are required to be terminating and confluent. It is terminating if its set of equations does not lead to infinite computation during reduction of terms. It is confluent if the reduction of a term always yields to the same result, no matter in which order, and where in the term, the equations are applied.

System module A system module is a rewrite theory. A rewrite theory has sorts, operators, and can have also types of statements such as equations and rules, all of which can be conditional. Therefore, any rewrite theory has an underlying equational theory, containing the equations plus the rules. In system modules, each step of rewriting is a step of replacement of equals by equals, until we find a fully evaluated value.

A system module is declared with the keywords `mod ... endm`. It can contain not only any elements of a functional module but also unconditional and conditional rewrite rules of the form:

```
rl [label] : Term1 => Term2 .
crl [label] : Term1 => Term2 if cond .
```

As in conditional equations, the condition (`cond`) can consist of a single statement, or can be a conjunction formed with the associative connective `/\`.

Unlike equational specifications, rewrite specifications can be nonterminating and nonconfluent, however, the equational part of a rewrite specification is still required to be terminating and confluent.

Module importation In Maude we can import other predefined modules with three existing modes: `protecting`, `extending` or `including` (the shorter versions are `pr`, `ex` and `inc`).

Importing a module M into M' in protecting mode means that no junk and no confusion are added to M when we include it in M'. The idea of extending mode is to allow junk, but to rule out confusion. Finally, in including mode there can be junk and/or confusion.

Parameterized programming Parameterized modules, theories and views are the basic building blocks of parameterized programming [BG80, DGS93]. A parameterized module is a module with one or more parameters, each of which is expressed by means of one theory, that is, modules can be parameterized by one or more theories. A *theory* defines the interface of a parameterized module, that is, the structure and properties required of an actual parameter. The instantiation of the formal parameters of a parameterized module with actual parameter modules or theories requires a *view*, mapping entities from the formal interface theory to the corresponding entities in the actual parameter module.

Theories declare the syntactic and semantic properties to be satisfied by the actual parameter modules used in an instantiation. As for modules, Maude supports two different types of theories: functional theories and system theories. Functional theories are declared with the keywords `fth ... endfth`, and system theories with the keywords `th ... endth`. Both of them can have sorts, subsort relationships, operators, variables, membership axioms, and equations, and can import other theories or modules. System theories can also have rules.

For instance, `TRIV` is the functional theory that consists of a single sort `Elt`.

```
fth TRIV is
  sort Elt .
endfth
```

This theory is very often used in the definition of data structures, such as lists, sets, trees, etc. It is common to define a module, say `LIST`, `SET`, `TREE`, etc., parameterized by the `TRIV` theory. The theory `TRIV` is predefined in Maude, together with several useful views from `TRIV` to other predefined modules and theories.

A module can be parameterized by one or more theories. For example, a simple parameterized module

```
fmod SET{X :: TRIV} is ... endfm
```

forms sets of models of the trivial parameter theory `TRIV`.

Views are used in Maude to specify how a particular target module or theory is claimed to satisfy a source theory. In the definition of a view we have to indicate its name, the source theory, the target module or theory, and the mapping of each sort and operator in the source theory.

```
view ViewName from Source to Target is
  Mappings
endv
```

For example, the mapping of a sort in the source theory to a sort in the target module or theory is expressed with the following syntax

```
sort Identifier to Identifier .
```

Some basic parameterized data types are predefined in the file `prelude.maude`, e.g. `SET{X :: TRIV}`, `LIST{X :: TRIV}`, etc. The file `prelude.maude` contains many views out of `TRIV`. For example, the predefined view `Int` in Maude

```
view Int from TRIV to INT is
  sort Elt to Int .
endv
```

shows how the module `INT`, where the sort `Int` for integers and arithmetic operators are defined, satisfies the theory `TRIV`.

A parameterized module is instantiated with views. For example, we can define a module providing a set of integers as follows

```
fmod INT-SET is
  pr SET{Int} .
  ...
endfm
```

where `Int` is a name of the predefined view.

2.5.2 Reflection, Metalevel

Reflection is a very important and powerful feature of rewriting logic. Thanks to this feature Maude programs can be meta-represented as a data that can be manipulated by appropriate functions [CMP07].

The key functionality of a metalevel theory with several descent functions has been implemented in Maude in a functional module named `META-LEVEL`, by using Maude's own interpreter. This module includes the modules `META-TERM`, where Maude terms are meta-represented as elements of a data type `Term` of terms, `META-MODULE`, where Maude modules are meta-represented as terms in a data type `Module` of modules, and `META-VIEW`, where Maude views are meta-represented as terms in a data type `View` of views. In the `META-LEVEL`, the operations `upModule`, `upTerm` and `downTerm` allow moving from the concrete to the abstract (meta) level. The processes of rewriting a term in a system module using Maude's `rewrite` and `frewrite` commands are meta-represented by built-in functions `metaRewrite` and `metaFrewrite`. The process of matching two terms at the top is metare-presented by the built-in function `metaMatch`. The process of matching a pattern to any subterm of a term is reified by the built-in function `metaXmatch`. The process of searching for a term satisfying some conditions starting at an initial term is meta-represented by built-in function `metaSearch`.

2.5.3 Unification

Unification is a fundamental deductive mechanism used in many automated deduction tasks. In the context of Maude, unification can be very useful to reason not only about equational theories, but also about rewrite theories. In Maude unification is reflected in the `META-LEVEL` module by the descent function `metaUnify`.

2.5.4 Narrowing

Narrowing generalizes term rewriting by allowing free variables in terms, and by performing unification instead of matching in order to (non-deterministically) reduce a term. At each rewriting step one must choose which subterm of the subject term and which rule of the specification are going to be considered.

In a rewriting step and a narrowing step we use a rewrite rule $l \to r$ to rewrite t at a position p in t. The difference between both it that narrowing unifies the left-hand side l and the chosen subject term $t\ |_p$ before actually performing the rewriting step. Also, narrowing is usually restricted to non-variable positions of t, whereas rewriting does not require such a restriction.

Full Maude provides a narrowing-based `search` command that is meta-represented by the built-in function `metaSearch`.

More details about Maude, Maude features and Maude tools could be found at `http://maude.cs.uiuc.edu/`.

2.6 Summary

This subsequent chapters build upon the basic notions presented in the present chapter. For the notion of first-order theory, central in this thesis, we have introduced the preliminary notions of first-order logic, signature, term, clause, structure, satisfiability and validity. We have shown some examples of theories of classical datatypes, such as lists, arrays and records, for which paramodulation is known to terminate. Since we consider the rewriting-based approach to build satisfiability procedures, we have introduced in this chapter the classical notion of term rewrite system. Moreover, we have introduced a method for proving termination of rewrite system based on simplification ordering. We have also studied the method of combinability of two satisfiability procedures proposed by Nelson and Oppen. Finally, the Maude language has been introduced to help understanding our implementation of schematic paramodulation calculi presented in Chapter 6.

Chapter 3

Paramodulation Calculi

Contents

Until 1969 the resolution method [Rob65] (which can prove that a first-order formula is not satisfiable) has been used, when reasoning on equalities. In this case we can use the resolution calculus by specifying the axioms of the theory of equality (reflexivity, symmetry, transivity, congruence) as clauses, but this is completely inefficient. In fact, the resolution generates too many unnecessary new clauses. To overcome this problem, Robinson and Wos [RW69] explored another possibility: they designed a new dedicated calculus, called *Paramodulation*. The aim of paramodulation is to integrate in the calculus the theory of equality. Therefore, the paramodulation calculus is a dedicated calculus for first order logic with equality. Since that time many improvements of the original version of paramodulation have been proposed [Bra75, Pet83, HR91]. These improvements were crucial in order to have an efficient paramodulation-based theorem-prover. Another important refinement was the use of orderings [Pet83, BG90, Rus91]. Orderings have been used to control the selection of the literals and subterms in them to be unified. In 1995 the authors of [BGLS95] have defined a framework for paramodulation (basic paramodulation) which depends on a reduction ordering and a selection function. Independently Nieuwenhuis and Rubio [NR92] have developed an inference system for completion and for refutational theorem proving based on basic superposition and have proved completeness in the context of deletion rules such as subsumption and simplification. In 2001 Nieuwenhuis and Rubio [NR01] have presented a consolidation of all the previous works in the handbook of automated reasoning.

Then, Armando, Ranise and Rusinowich [ARR01, ARR03] have worked on a general and flexible approach to derive satisfiability procedures by paramodulation. In fact, satisfiability procedures modulo background theories of classical data structures such as lists, arrays and records are at the core of many state-of-the-art verification tools. The design, the proof of correction and the implementation of such satisfiability procedures is a very complex task. One of the main difficulties consists in proving their correctness. Moreover, the implementation of each procedure is done in an *ad hoc* way, with no intention to reuse the code. To overcome this problem, the authors of [ARR01] proposed a uniform approach based on saturation to build satisfiability procedures and to prove their correctness. The proof of correctness is reduced to the proof of the termination of a fair and exhaustive application of the rules of *Paramodulation Calculus* $\mathcal{PC}$ [NR01] (also called superposition calculus). This calculus is presented in Section 3.1. An approach for building decision procedures by saturation is discussed in Section 3.2. The termination of paramodulation on several theories, such as the theory of records, the theory of lists, etc. has been proved in [ABRS09].

However, most of the verification problems involve different theories for which the approach based on saturation may not apply. Therefore, there is an obvious need to build a satisfiability procedure for the union of theories by reusing and combining existing satisfiability procedures for component theories.

The authors of [ARR03] have shown how to combine the satisfiability procedures obtained thanks to $\mathcal{PC}$ for the theory of arrays and the theory of lists *à la* Shostak. A general modularity theorem has been proposed in [ABRS05]. This theorem states sufficient conditions for $\mathcal{PC}$ to terminate on satisfiability problems of a union of theories, if it terminates on the satisfiability problems of each theory taken separately. This theorem is discussed in Section 3.3.

The authors of [NRR09c] have presented a new technique to combine satisfiability procedures for theories that model data structures (the theory of lists and the theory of records) and that share the theory of Integer Offsets. Contrary to the Nelson-Oppen approach (Section 2.4), this approach considers non-disjoint theories. In [NRR09c] it is shown how to apply a paramodulation calculus to build satisfiability procedures that can be plugged into the non-disjoint combination framework. This technique is presented in Section 3.4. The authors of [RS11] have discussed a question of modular termination of theories sharing Integer Offsets. The main theorem is presented in Section 3.4.3.

3.1 Paramodulation calculus

Let us consider a theory T axiomatized by a set $Ax(T)$ of finitely many axioms and a set S composed of ground flat literals. To decide whether S is satisfiable in T, the rules of paramodulation calculus are applied in an exhaustive way to the set $Ax(T) \cup S$. If the empty clause is derived, then S is unsatisfiable in T. Otherwise, it is satisfiable in T. The correctness of these procedures is guaranteed by the correctness of the rules of the paramodulation calculus.

The paramodulation calculus is a refutation-complete inference system for general first-order equational logic [NR01]. It means that any fair application of the rules to

an unsatisfiable set of clauses will derive the empty clause. Fairness means that if some inference is possible it will be performed at some step unless one of the parent clauses gets simplified, subsumed, or deleted.

In general this calculus provides a semi-decision procedure that halts on unsatisfiable input, but may not terminate on satisfiable ones. However, it also terminates on satisfiable inputs for some theories axiomatizing standard datatypes such as lists, arrays, etc. Therefore, the paramodulation calculus provides a decision procedure for the theory of interest if one can show that it terminates on every input made of the finitely many axioms and any set of ground literals.

A fundamental feature of $\mathcal{PC}$ is the use of a simplification ordering $<$ to control the application of some rules by orienting equalities. This ordering is total on ground terms. We use a lexicographic path ordering presented in Section 2.3.2.

The inference system $\mathcal{PC}$ consists of the rules in Figures 3.1 and 3.2. Clauses are presented here under the form of disjunctions as in [LM02]. They could also be presented under the form of implications as in [ARR03, LRRT11].

Superposition $\dfrac{C \vee l[u'] \bowtie r \qquad D \vee u = t}{\sigma(C \vee D \vee l[t] \bowtie r)}$

if $\sigma(u) \not\leq \sigma(t)$, $\sigma(l[u']) \not\leq \sigma(r)$, $l[u'] \bowtie r$ and $u = t$ are selected in their clauses.

Reflection $\dfrac{C \vee u \neq u'}{\sigma(C)}$

if $u \neq u'$ is selected in its clause.

Eq. Factoring $\dfrac{C \vee u = t \vee u' = t'}{\sigma(C \vee t \neq t' \vee u = t)}$

if $\sigma(u) \not\leq \sigma(t)$, $u = t$ is selected in its clause, $\sigma(t) \not\leq \sigma(t')$ and $\sigma(u') \not\leq \sigma(t')$.

Above, σ is the most general unifier of u and u' and C and D are clauses.
In the *Superposition* rule, u' is not a variable.

Figure 3.1: Expansion inference rules of $\mathcal{PC}$

The expansion rules (Figure 3.1) aim at generating new (deduced) clauses. These rules use a selection function *sel* such that for each clause C, $sel(C)$ contains a negative literal in C or all maximal literals in C with respect to $<$. For brevity left and right paramodulation rules are grouped into a single rule, called *Superposition*, that uses an equality to perform a replacement of equal by equal into a literal. The *Superposition* rule is applied between two selected literals by using terms that are maximal in their literals with respect to $<$. The *Reflection* and *Eq. Factoring* rules generate a new clause from the instantiation of an existing one. *Reflection* removes a selected disequality in a clause

Subsumption	$\dfrac{S \cup \{C, C'\}}{S \cup \{C\}}$
	if for some substitution σ, $\sigma(C) \subseteq C'$.
Simplification	$\dfrac{S \cup \{C[l'], l = r\}}{S \cup \{C[\sigma(r)], l = r\}}$
	if $l' = \sigma(l)$, $\sigma(l) > \sigma(r)$, and $C[l'] > (\sigma(l) = \sigma(r))$
Tautology	$\dfrac{S \cup \{C \vee t = t\}}{S}$
	Above, C and C' are clauses and S is a set of clauses.

Figure 3.2: Contraction inference rules of $\mathcal{PC}$

when its two sides can be unified. When the clause is unit, it generates the empty clause. *Eq. Factoring* factorizes two equalities when their left-hand sides can be unified.

The contraction rules (Figure 3.2) aim at simplifying the set of clauses. Using *Subsumption*, a clause is removed when it is an instance of another one. *Simplification* rewrites a literal into a simpler one by using an equality that can be considered as a rewrite rule. Trivial equalities are removed by *Tautology*.

Definition 39 (Redundancy). *Let I be an inference system. We say that*

- *a clause C is* redundant *with respect to I and with respect to a set S of clauses if either $C \in S$, or S can be obtained from $S \cup \{C\}$ by a sequence of applications of the contraction rules from I.*
- *an inference is* redundant *with respect to a set of clauses S if its conclusion is redundant with respect to I and with respect to S.*

Definition 40 (Saturation). *A set of clauses S is* saturated *with respect to an inference system I if every inference from I with a premise in S is redundant with respect to S.*

A saturation of a set of clauses S by $\mathcal{PC}$ is the final set of clauses generated by a fair derivation from S using rules in $\mathcal{PC}$ with higher priority given to the contraction rules. If the saturation terminates for the union of $Ax(T)$ and S, for any S, then it is a decision procedure for T: if the final set of clauses contains the empty clause then the input set of literals is unsatisfiable, otherwise, it is satisfiable.

The notion of fair derivation is made precise by the following definitions:

Definition 41 (Derivation). *A* derivation *is a sequence $S_0, S_1, \ldots, S_i, \ldots$ of sets of clauses where each S_{i+1} is obtained from S_i by applying an inference to add a clause (by expansion rules) or to delete a clause (by contraction rules).*

A derivation is characterized by its limit, *defined as the set of persistent clauses* $S_\infty = \bigcup_{j\geq 0}\bigcap_{i>j} S_i$, *that is, the union for each* $j \geq 0$ *of the set of clauses occurring in all future steps starting from* S_j.

For the *Simplification* rule, one can remark that its application corresponds to two steps in the derivation: the first step adds a new literal, whilst the second one deletes a literal.

Definition 42 (Fair derivation)**.** *A derivation* $S_0, S_1, ..., S_i, ...$ *is* fair *if for every inference with premises in the limit, there is some* $j \geq 0$ *such that the inference is redundant with respect to* S_j.

The set of persistent literals obtained by a fair derivation is called the *saturation* of the derivation.

Theorem 1 ([NR01])**.** *If* $S_0, S_1, \ldots$ *is a fair derivation with respect to* $\mathcal{PC}$, *then its limit* S_∞ *is saturated with respect to* $\mathcal{PC}$, *and* S_0 *is unsatisfiable if and only if* S_j *is the empty clause for some* j. *Moreover, if* $S_0, S_1, \ldots, S_n$ *is a fair derivation then* S_n *is saturated and logically equivalent to* S_0.

3.2 Saturation-based satisfiability procedures

Let T be a theory axiomatized by a set $Ax(T)$ of finitely many axioms. The methodology based on saturation proposed in [ARR01] consists of two steps:

1. *flattening* all the input literals (by introducing "fresh" constants),
2. *choosing an order* and *proving the termination.* Termination means that one have to ensure that the saturation of axioms with an arbitrary set of ground flat literals generates only finitely many clauses. Termination may depend on simple properties of the ordering $>$: an ordering that satisfies them is called T-good.

Example 9 (Flattening)**.** *Let us consider two examples:*

- *positive literal* $f(f(f(a))) = b$. *To flatten this literal, we introduce the new constant* c_1 *that represent* $f(a)$. *Thus, we get a conjunction* $\{f(f(c_1)) = b, c_1 = f(a)\}$. *Then, we introduce another constant* c_2 *that represents* $f(c_1)$. *Thus the conjunction becomes* $\{f(c_2) = b, c_1 = f(a), c_2 = f(c_1)\}$. *Thus the literal* $f(f(f(a))) = b$ *can be represented by the conjunction of literals* $\{f(c_2) = b, c_2 = f(c_1), c_1 = f(a)\}$.
- *negative literal* $f(g(a)) \neq f(b)$. *Similarly to the previous example, we replace the subterm* $g(a)$ *of the term* $f(g(a))$ *by a new constant* c_1. *Thus we get the conjunction* $\{f(c_1) \neq f(b), c_1 = g(a)\}$. *Then we flatten both sides of the disequality by introducing two new constants* c_2 *and* c_3. *Thus the literal* $f(g(a)) \neq f(b)$ *can be represented by the conjunction of literals* $\{c_2 \neq c_3, c_2 = f(c_1), c_3 = f(b), c_1 = g(a)\}$.

□

The flattening augments the size of the input set of literals S to $O(n)$, where n is the number of subterms in S.

Example 10 (T-good ordering for L). *An ordering $>$ for the theory of lists is T-good if*

- *$t > c$ for all ground compound terms t and all constants c*
- *$l > e$ for all constants l of sort* LISTS *and constants e of sort* ELEM.

□

Therefore, if T is a theory for which this methodology applies, then a T-satisfiability procedure can be constructed by flattening the input literals and then applying the rules of paramodulation calculus with a suitable order. If the final set contains the empty clause, then the T-satisfiability procedure returns unsatisfiable, otherwise it returns satisfiable.

In all that follows, we suppose that the set of literals for which we want to decide satisfiability is only composed of ground flat literals.

For some theories the paramodulation is known to terminate, and therefore, to be a satisfiability procedure. For example:

1. the theory of lists *à la* Shostak (Section 2.2.1)

   Theorem 2 ([ARR03]). *PC is a polynomial satisfiability procedure for L.*

2. the theory of arrays (Section 2.2.5)

   Theorem 3 ([ARR03]). *PC is an exponential satisfiability procedure for A.*

3. the theory of records (Section 2.2.3)

   Theorem 4 ([ABRS09]). *PC is a polynomial satisfiability procedure for R.*

4. the theory of possibly empty lists (Section 4.4)

   Theorem 5 ([ABRS09]). *PC is an exponential satisfiability procedure for PEL.*

3.3 Combination of theories

The combination of the theory of lists and the theory of arrays is considered in [ARR03]. It is shown that $\mathcal{PC}$ terminates when considering the union of axioms $Ax(L) \cup Ax(A)$. More generally, the question is whether the rewrite-based approach is well-suited to design satisfiability procedures for a combination of theories. When satisfiability procedures are built for theories $T_1, \ldots, T_n$ thanks to $\mathcal{PC}$, the problem is to show that $\mathcal{PC}$ also decides the satisfiability problem in the union of theories $T = \bigcup_{i=1}^{n} T_i$. In [ABRS05, ABRS09] three sufficient conditions are established in order to show that $\mathcal{PC}$ terminates on T-satisfiability problems if it terminates on T_i-satisfiability problems for all i, $1 \leq i \leq n$:

1. The ordering is T-good, where $T = \bigcup_{i=1}^{n} T_i$, which means that for all i, $1 \leq i \leq n$, the ordering is T_i-good.

2. There is no across-theories inferences, since a variable can superpose with any non-variable subterm. To identify the clauses generating these undesirable inferences, the concept of variable-active clause has been introduced.

   Definition 43 (Variable-active clause)**.** *A clause C is variable-active with respect to an ordering $>$ if C contains a maximal (with respect to $>$) literal of the form $X = t$, where X is a variable not occurring in $Var(t)$.*

3. The signatures of theories do not share functional symbols. Sharing of constant symbols is allowed. Therefore, the only inference across theories are paramodulations from constants to constants, that are finitely many, since there are finitely many constants.

Theorem 6 (Theorem 5, [ABRS05])**.** *Let $T_1, \ldots, T_n$ be signature-disjoint theories, and let $T = \bigcup_{i=1}^{n} T_i$. Assume that for all i, $1 \leq i \leq n$, S_i is a set of ground flat T_i-literals. If for all i, $1 \leq i \leq n$, $\mathcal{PC}$ terminates on $Ax(T_i) \cup S_i$ with a saturation containing no variable-active clause, then $\mathcal{PC}$ terminates on $Ax(T) \cup S_1 \cup \ldots \cup S_n$, and therefore is a satisfiability procedure for T.*

To ensure efficiency, it is very useful to have built-in axioms in the calculus, and so to design paramodulation calculi modulo theories. This is particularly important for arithmetic fragments due to the ubiquity of arithmetics in applications of formal methods. For instance, paramodulation calculus has been developed for Integer Offsets [NRR09c]. This point is discussed in the next section.

3.4 Paramodulation calculus for Integer Offsets

The paramodulation calculus for theories sharing Integer Offsets has been presented in [NRR09c]. The authors of this paper considered only unitary clauses, i.e. clauses composed of at most one literal. Therefore, they considered the unitary version of paramodulation calculus $\mathcal{PC}$. They adapt this calculus to the theory of Integer Offsets, so that it can serve as a basis for the design of decision procedures for Integer Offsets extensions.

3.4.1 Theory of Integer Offsets

The theory of Integer Offsets T_I is axiomatized by the following set $Ax(I)$ of axioms:

$$\begin{array}{lll} (s0) & \forall X. & \mathsf{s}(X) \neq \mathsf{0} \\ (inj) & \forall X, Y. & \mathsf{s}(X) = \mathsf{s}(Y) \Rightarrow X = Y \\ (acy) & \forall X. & X \neq \mathsf{s}^n(X) \quad \text{for all} \quad n \geq 1 \end{array}$$

over the signature $\Sigma_I := \{\mathsf{0} : \text{INT}, \mathsf{s} : \text{INT} \rightarrow \text{INT}\}$. The second axiom specifies that the successor function s is injective. The third axiom is in fact an axiom scheme, which specifies that this function is acyclic.

3.4.2 Extending the paramodulation calculus to Integer Offsets

The unitary paramodulation calculus for theories sharing Integer Offsets $\mathcal{UPC}_I$ consists of the unitary version of the rules of paramodulation calculus $\mathcal{PC}$: the expansion rules (Figure 3.1), the contraction rules (Figure 3.2), plus the additional reduction rules corresponding to the axioms of the theory of Integer Offsets (Figure 3.3).

R1	$\dfrac{S \cup \{\mathsf{s}(u) = \mathsf{s}(v)\}}{S \cup \{u = v\}}$ **if** u and v are ground terms.
R2	$\dfrac{S \cup \{\mathsf{s}(u) = t, \mathsf{s}(v) = t\}}{S \cup \{\mathsf{s}(v) = t, u = v\}}$ **if** u, v and t are ground terms, $\mathsf{s}(u) > t$, $\mathsf{s}(v) > t$ and $u > v$.
C1	$\dfrac{S \cup \{\mathsf{s}(t) = \mathsf{0}\}}{S \cup \{\mathsf{s}(t) = \mathsf{0}, \bot\}}$ **if** t is a ground term.
C2	$\dfrac{S \cup \{\mathsf{s}^n(t) = t\}}{S \cup \{\mathsf{s}^n(t) = t, \bot\}}$ **if** t is a ground term and $n \geq 1$.

Figure 3.3: Ground reduction rules for Integer Offsets

The authors of [NRR09c] adapt the notion of derivation to the paramodulation calculus for theories sharing Integer Offsets as follows:

Definition 44 (Derivation)**.** *A derivation (δ) with respect to $\mathcal{UPC}_I$ is a (finite or infinite) sequence of sets of literals $S_1, S_2, S_3, \ldots, S_i, \ldots$ such that, for every i, it happens that:*

- *S_{i+1} is obtained from S_i adding a literal obtained by the application of one of the rules in Figures 3.1, 3.2 and 3.3 to some literals in S_i;*
- *S_{i+1} is obtained from S_i removing a literal according to one of the rules in Figure 3.2 or to the rule $R1$ or $R2$.*

Note that the application of *Simplification*, $R1$ and $R2$ involve two steps in the derivation: in the first step a new literal is added, and in the second one a literal is deleted.

The refutation completeness of $\mathcal{UPC}_I$ is also studied in [NRR09c]. It is assumed that the ordering considered when performing any application of $\mathcal{UPC}_I$ is $T_I - good$.

Definition 45 ($T_I - good$ ordering)**.** *An ordering $<$ is* $T_I -$ good *if $<$ is a simplification ordering which is total on ground terms, such that*

1. $\mathsf{0}$ *is minimal, and*

2. *for any non* s*-rooted terms* t_1 *and* t_2, $\mathsf{s}^{n_1}(t_1) > \mathsf{s}^{n_2}(t_2)$ *iff either* $t_1 = t_2$ *and* n_1 *is bigger than* n_2, *or* $t_1 > t_2$.

Theorem 7 ([NRR09c]). *Let* T *be a* Σ*-theory presented as a finite set of unit clauses such that* $\Sigma \supseteq \Sigma_I$, *and assume to put an ordering over terms that is* $T_I -$ good. $\mathcal{UPC}_I$ *induces a decision procedure for the satisfiability problem with respect to* $T \cup T_I$ *if, for any set* G *of ground literals:*

- *the saturation of* $Ax(T) \cup G$ *w.r.t.* $\mathcal{UPC}_I$ *is finite,*
- *the saturation of* $Ax(T) \cup G$ *w.r.t.* $\mathcal{UPC}_I$ *does not contain non-ground equalities whose maximal term is* s*-rooted.*

The authors of [NRR09c] have shown that $\mathcal{UPC}_I$ terminates for some theories sharing Integer Offsets, such that the theory of lists with length, the theory of lists over integer elements and the theory of records with increment.

3.4.3 Combination of theories

The combination of theories extending T_I is addressed in [RS11]. As for the disjoint case (see Theorem 6), it is important to forbid variable-active clauses in the saturation.

Theorem 8 ([RS11]). *Let* $T_1, \ldots, T_n$ *be theories sharing only the function symbols of* T_I. *and let* $T = \bigcup_{i=1}^{n} T_i$. *Assume that for all* i, $1 \leq i \leq n$, S_i *is a set of ground flat* T_i*-literals. Assume that for all* i, $1 \leq i \leq n$, $\mathcal{UPC}_I$ *terminates on* $Ax(T_i) \cup S_i$ *with a saturation containing (1) no non-ground equality with a maximal* s*-rooted term, and (2) no variable-active clause. Then,* $\mathcal{UPC}_I$ *terminates on* $Ax(T) \cup S_1 \cup \ldots \cup S_n$, *and therefore is a satisfiability procedure for* $T \cup T_I$.

In Theorem 8, the conditions on the form of saturations correspond to the definition of *safe* saturations introduced in [RS11].

3.5 Summary

In this chapter we have presented the paramodulation calculus. We have introduced the expansion and contraction rules of $\mathcal{PC}$ and have described each of its rule. We have also introduced such notions as redundancy, saturation, derivation and fairness, which are the key notions of paramodulation calculus. We have presented the approach for deriving decision procedures based on saturation [ARR01].

The question of having built-in axioms in the calculus has been addressed as well. We have shown how the authors of [NRR09c] designed the paramodulation calculus developed for Integer Offsets. This is particularly important since arithmetics are used ubiquitously in applications of formal methods.

Moreover, we have discussed the combination problem of both signature-disjoint theories and theories with non-disjoint signature. As one could notice this problem is easily solved thanks to the paramodulation calculus.

Chapter 4

Schematic Paramodulation Calculus

Contents

A classical termination proof of some theory by paramodulation calculus consists in considering the finitely many cases of inputs made of the (finitely many) axioms and any set of ground flat literals. This proof can be done by hand, by analysing the finitely many forms of clauses generated by saturation, but the process is tedious and error-prone.

To simplify this process, a *Schematic Paramodulation Calculus* has been developed in [LM02] to build the schematic form of the saturations. It can be seen as an abstraction of the paramodulation calculus: If it halts on one given abstract input, then the paramodulation calculus halts for all the corresponding concrete inputs. The input of schematic paramodulation calculus consists of axioms of the background theory T and the schematic representation of an arbitrary set of ground flat literals in T. Thus, thanks to the schematic paramodulation calculus we can check the decidability of the satisfiability problem for T. Also, as explained in [LM02], this calculus gives us an upper bound on the number of clauses generated by the paramodulation calculus.

The schematic paramodulation does not only allow checking the decidability of the satisfiability problem for a theory, but it also determines the modular termination. If paramodulation calculus decides the satisfiability problem of two theories separately, and if saturations of considered theories satisfy the condition presented in Section 4.6, then it also decides the satisfiability problem of their union [LRRT11]. Therefore, schematic paramodulation is a fundamental tool to check important properties related to decidability and combinability [LRRT11].

The schematic paramodulation calculus is almost identical to the concrete paramodulation calculus, except that clauses are constrained. Such clauses contain constrained variables: these variables can be replaced only by constants. In this chapter, before presenting schematic paramodulation calculus, we introduce useful notions such as constraint, constrained clause, constrained variable, constrained instance, etc. Then, we present a lexicographic path ordering used in the side-conditions of the rules of this calculus. This ordering is different from the one introduced in Section 2.3.2 since schematic paramodulation calculus uses constrained variables. After that, we present the expansion and the contraction rules of the calculus. We show that any concrete saturation computed by concrete paramodulation can be viewed as an instance of an abstract saturation by schematic paramodulation. Finally, we consider an approach for building satisfiability procedures for unions of theories.

4.1 Constrained clauses

This section introduces the notions of constraint, constrained clause, constrained variable and elementary instance.

Definition 46 (Atomic constraint, Constraint)**.** *An* atomic constraint *is of the form* $const(t)$*, where* t *is a term. A* constraint *is a conjunction of atomic constraints.*

Definition 47 (Constrained clause)**.** *A* constrained clause *is of the form* $C\|\varphi$*, where* C *is a clause and* φ *is a constraint.*

Definition 48 (Constrained variable)**.** *A variable* x *is* constrained *in a constrained clause* $C\|\varphi$ *if* φ *contains* $const(x)$*; otherwise it is* unconstrained.

In fact, a constrained variable is a schematization of constants. It can only be instantiated by a constant. An unconstrained variable is a universally quantified variable that can be instantiated by any term. For sake of brevity, $const(x_1, \ldots, x_n)$ denotes the conjunction $const(x_1) \wedge \cdots \wedge const(x_n)$.

Definition 49 (Constraint satisfaction)**.** *The atomic constraint* $const(t)$ *is* true *iff* t *is a constant. A constraint is* true *if all its atomic constraints are true. A substitution* σ satisfies *a constraint* φ *if* $\sigma(\varphi)$ *is true. A constraint is* satisfiable *if there exists a substitution* σ *satisfying it.*

Consequently, the atomic constraint $const(t)$ is unsatisfiable if t is a term of depth 1 or more, i.e., when t contains a non-constant function symbol. When a constraint contains a true atomic constraint, we assume that this true atomic constraint is automatically removed from the constraint.

Definition 50 (Constraint instance)**.** *A* constraint instance *of the constrained clause* $C\|\varphi$ *is any clause of the form* $\sigma(C)$ *where* σ *is a substitution satisfying* φ*.*

For example, if a is a constant then the clause $f(a) = X$ is a constraint instance of the constrained clause $f(x) = X\|const(x)$, where x is a constrained variable and X is

an unconstrained variable. A constrained clause is used to schematize the set of all its constraint instances.

The notion of constraint instance is extended to constrained clauses by the following definition.

Definition 51 (Elementary instance)**.** *Let $C\|\varphi$ and $C'\|\varphi'$ be two constrained clauses. We say that $C'\|\varphi'$ is an* elementary instance *of $C\|\varphi$ if there exists a substitution σ replacing some constrained variables of $C\|\varphi$ with constrained variables or constants of $C'\|\varphi'$ such that $C' = \sigma(C)$ and $\varphi' = \sigma(\varphi)$.*

For instance, the clause $f(x) = \mathsf{nil} \vee y = g(z)\|const(x, y, z)$ is an elementary instance of the clause $f(a) = b \vee c = g(d)\|const(a, b, c, d)$, because the substitution $\sigma = \{a \leftarrow x, b \leftarrow \mathsf{nil}, c \leftarrow y, d \leftarrow z\}$ satisfies all the conditions in Definition 51.

Notice that our notion of constraint is less general and thus more precise than the one in [LRRT11], where an atomic constraint is some $t \leq t'$. Our atomic constraint $const(t)$ corresponds to $t \leq c^T$ in [LRRT11], where c^T represents the biggest constant with respect to $\leq$. Since the ordering $\leq$ is extended to open terms, there may exist unground substitutions σ such that $\sigma(t \leq t')$ is true. By contrast, our constraints are satisfiable iff they are satisfiable by a ground substitution that replaces all their variables with constants.

4.2 Ordering

In the classical definition of LPO (Definition 34), variables are not considered because they are not comparable. Here, constrained clauses contain constrained variables that can be replaced with constants of the same sort. Two constrained variables of different sorts are comparable. In fact, each constrained variable is instantiated with a constant of the appropriate sort. Therefore, when comparing two constrained variables, two corresponding constants are compared by using the precedence order $>_F$ that is composed of functional symbols and at least one constant per sort.

Example 11. *The precedence order for the theory of lists with length presented in Section 2.2.2 is the following:* $\mathsf{cons} < \mathsf{cdr} < \mathsf{car} < \mathsf{nil} < e < \mathsf{len} < \mathsf{0} < \mathsf{s}$*, where* nil *is a constant of sort* LISTS*, e is a constant of sort* ELEM *and* 0 *is a constant of sort* INT*. As mentioned above, to compare two constrained variables, for example, one of them is of sort* LISTS*, and another one is of sort* ELEM*, we compare their corresponding constants. In our example the corresponding constant of the constrained variable of sort* LISTS *is* nil*, and the corresponding constant of the constrained variable of sort* ELEM *is e. From the precedence it is known that the constants of sort* LISTS *are greater than the constants of sort* ELEM*. Therefore, the constrained variable of sort* LISTS *is greater than the constrained variable of sort* ELEM*.* □

We adapt the classical definition of LPO in order to take into account constrained variables. We use the following definition when considering schematic paramodulation calculus. The standard definition of lexicographic path ordering (Definition 34) is used when considering (concrete) paramodulation calculus.

Definition 52 (Lexicographic path ordering with constrained variables)**.** *Given a precedence $>_F$ on function symbols, the* ***lexicographic path ordering*** *$>_{lpo}$ is defined as follows:*

$$(1)\quad \frac{(s_1,\ldots,s_n) >_{lpo}^{lex} (t_1,\ldots,t_m) \qquad f(s_1,\ldots,s_n) >_{lpo} t_1,\ldots,t_m}{f(s_1,\ldots,s_n) >_{lpo} f(t_1,\ldots,t_m)}$$

$$(2)\quad \frac{f \;>_F\; g \qquad f(s_1,\ldots,s_n) >_{lpo} t_1,\ldots,t_m}{f(s_1,\ldots,s_n) >_{lpo} g(t_1,\ldots,t_m)}$$

$$(3)\quad \frac{f >_F c_{sort(v)}}{f(s_1,\ldots,s_n) >_{lpo} v}$$

$$(4)\quad \frac{c_{sort(v)} \;>_F\; g \qquad v >_{lpo} t_1,\ldots,t_m}{v >_{lpo} g(t_1,\ldots,t_m)}$$

$$(5)\quad \frac{sort(vc_1) \neq sort(vc_2) \qquad c_{sort(vc_1)} \;>_F\; c_{sort(vc_2)}}{vc_1 >_{lpo} vc_2}$$

$$(6)\quad \frac{u_k \;>_{lpo}\; t}{f(u_1,\ldots,u_k,\ldots,u_p) >_{lpo} t}$$

$$(7)\quad \frac{}{f(u_1,\ldots,u_k,\ldots,u_p) >_{lpo} u_k}$$

where f and g are two functional symbols, $n \geq 0$ and $m \geq 0$ are two non-negative integers, $p \geq 1$ is a positive integer, and $s_1, \ldots, s_n$, $t_1, \ldots, t_m$, $u_1, \ldots, u_p$, t are terms, vc_1 and vc_2 are two terms of depth 0 (i.e. constrained variables or constants). We write $s >_{lpo} t_1,\ldots,t_m$ when $s >_{lpo} t_k$ for $k = 1,\ldots,m$.

The expression $sort(t)$, where t is a constrained variable or constant, denotes the sort of the given term. We denote by $c_{sort(t)}$ the constant itself if t is a constant or the constant that represents the given constrained variable t: the sort of this constant is the same as the sort of the given constrained variable. Let us consider again the precedence given in Example 11. In this precedence the constant nil represents constrained variables of sort LISTS, the constant e represents constrained variables of sort ELEM, and the constant 0 represents constrained variables of sort INT.

The rules (1), (6), and (7) respectively correspond to the rules LPO1, LPO3 and LPO4 in Definition 34. All the possible cases of the rule LPO2 are considered in (2)–(5).

The rule (2) compares two compound terms with different root symbols. The term $f(s_1,\ldots,s_n)$ is greater than the term $g(t_1,\ldots,t_m)$ if the root symbol f of the first term precedes the root symbol g of the second one, and the first term $f(s_1,\ldots,s_n)$ is greater than each subterm of the second one.

The rule (3) sets that a compound term $f(s_1,\ldots,s_n)$ is greater than a constrained variable v if its root symbol f precedes a constant $c_{sort(v)}$ corresponding to the constrained variable v.

The rule (4) compares a constrained variable and a compound term. We say that $v >_{lpo} g(t_1,\ldots,t_m)$ if a constant representing v precedes the root symbol g, and if v is greater than each subtems of the given compound term.

The rule (5) establishes that one constrained variable is greater than another one if they are of different sorts, and if the constant representing the first constrained variable precedes the constant representing the second one.

When the signature of a given theory is unsorted, the constrained variables are not comparable.

4.3 Schematic calculus

$\mathcal{SPC}$ consists of the rules in Figs. 4.1 and 4.2. With respect to [LRRT11], we have slightly adapted the subsumption rule so that the instantiation is not only a renaming but can also be a substitution instantiating constrained variables by constrained variables or constants. For example, the constrained clause $x \neq \mathsf{nil} \parallel const(x)$ where nil is a constant is subsumed by the constrained clause $a \neq b \parallel const(a, b)$. This allows us to have a more compact form of saturations for all the considered theories in Chapter 7.

Superposition

$$\frac{C \vee l[u'] \bowtie r \| \varphi \qquad D \vee u = t \| \psi}{\sigma(C \vee D \vee l[t] \bowtie r \| \varphi \wedge \psi)}$$

if $\sigma(u) \not\leq \sigma(t)$, $\sigma(l[u']) \not\leq \sigma(r)$, $l[u'] \bowtie r$ and $u = t$ are selected in their clauses.

Reflection

$$\frac{C \vee u \neq u' \| \varphi}{\sigma(C \| \varphi)}$$

if $u \neq u'$ is selected in its clause

Eq. Factoring

$$\frac{C \vee u = t \vee u' = t' \| \varphi}{\sigma(C \vee t \neq t' \vee u = t \| \varphi)}$$

if $\sigma(u) \not\leq \sigma(t)$, $u = t$ is selected in its clause, $\sigma(t) \not\leq \sigma(t')$ and $\sigma(u') \not\leq \sigma(t')$.

Above, σ is the most general unifier of u and u' and C and D are clauses. In the *Superposition* rule, u' is not a unconstrained variable.

Figure 4.1: Expansion inference rules of $\mathcal{SPC}$

For a given theory T with signature Σ, let

$$\begin{aligned} G_0 = \; & \{\bot, x = y \parallel const(x, y), x \neq y \parallel const(x, y)\} \\ & \cup \bigcup_{f \in \Sigma} \{f(x_1, \ldots, x_n) = x_0 \parallel const(x_0, x_1, \ldots, x_n)\} \end{aligned}$$

Subsumption $\dfrac{S \cup \{C\|\varphi, C'\|\varphi'\}}{S \cup \{C\|\varphi\}}$

if a) $C \in Ax(T)$, φ is empty and for some substitution σ, $C' = \sigma(C)$; or b) $C' = \sigma(C)$ and $\varphi' = \sigma(\varphi)$, where σ is a renaming or a mapping from constrained variables to constrained variables or constants

Simplification $\dfrac{S \cup \{C[l']\|\varphi, l = r\}}{S \cup \{C[\sigma(r)]\|\varphi, l = r\}}$

if i) $l = r \in Ax(T)$, ii) $l' = \sigma(l)$, iii) $\sigma(l) > \sigma(r)$, and $C[l'] > (\sigma(l) = \sigma(r))$

Tautology $\dfrac{S \cup \{C \vee t = t\|\varphi\}}{S}$

Deletion $\dfrac{S \cup \{C\|\varphi\}}{S}$ **if** φ is unsatisfiable

Above, $C\|\varphi$ and $C'\|\varphi'$ are constrained clauses and S is a set of constrained clauses.

Figure 4.2: Contraction inference rules of $\mathcal{SPC}$

where $n \geq 1$. This set schematizes any set of ground flat equalities and disequalities built over Σ, along with the empty clause. The procedure for checking termination of any fair paramodulation strategy is based on *Schematic Saturation*, which consists in executing $\mathcal{SPC}$ on $Ax(T) \cup G_0$. If $\mathcal{SPC}$ halts on $Ax(T) \cup G_0$, then $\mathcal{PC}$ halts on $Ax(T) \cup S$, for any arbitrary set S of ground flat literals. This property will be proved in Sect. 4.5. A key ingredient is the Schematic Deletion rule defined in the next section.

4.4 Schematic Deletion rule

The schematic saturation may generate longer and longer clauses (containing new constrained variables) from clauses containing unconstrained variables and therefore diverge. To illustrate this fact let us consider two examples, namely the theory of arrays and the theory of possibly empty lists.

The theory of arrays is axiomatized by the following set $Ax(A)$ of axioms:

$$\mathsf{select}(\mathsf{store}(A, I, E), I) = E$$
$$\mathsf{select}(\mathsf{store}(A, I, E), J) = \mathsf{select}(A, J) \vee I = J$$

For every set S of ground flat literals, any saturation of $Ax(A) \cup S$ is finite [ABRS09], while schematic saturation diverges [LRRT11]. In fact, it generates the clause

$$\mathsf{select}(x, I) = \mathsf{select}(y, I) \vee z = I \parallel const(x, y, z)$$

whose superposition with a renamed copy of itself generates a clause of a new form

$$\mathsf{select}(x, I) = \mathsf{select}(y, I) \vee z = I \vee w = I \parallel const(x, y, z, w)$$

This process continues to generate longer and longer clauses so that the schematic saturation does not terminate.

The theory of possibly empty lists (*PEL* for short) is axiomatized by the following set $Ax(PEL)$ of axioms:

$$\mathsf{car}(\mathsf{cons}(X, Y)) = X \qquad \mathsf{cons}(\mathsf{car}(Y), \mathsf{cdr}(Y)) = Y \vee Y = \mathsf{nil}$$
$$\mathsf{cdr}(\mathsf{cons}(X, Y)) = Y \qquad \mathsf{car}(\mathsf{nil}) = \mathsf{nil}$$
$$\mathsf{cons}(X, Y) \neq \mathsf{nil} \qquad \mathsf{cdr}(\mathsf{nil}) = \mathsf{nil}$$

The schematic saturation generates the clause

$$\mathsf{cons}(x, \mathsf{cdr}(y)) = z \vee z = \mathsf{nil} \parallel const(x, y, z)$$

whose superposition with a renamed copy of itself generates a clause

$$z = z' \vee z = \mathsf{nil} \vee z' = \mathsf{nil} \parallel const(z, z')$$

This process goes on to generate longer and longer clauses so that the schematic saturation diverges as well.

A *Schematic Deletion* rule has been designed [LRRT11] to cope with this problem. We adapt this rule to take into account the constants in the theory signature, such as the constant nil for the theory of possibly empty lists. The new version of the *Schematic Deletion* rule is composed of the two rules in Figure 4.3. The idea behind these rules is to delete:

1. disjunctions of two or more equalities and disequalities between two constrained variables or between a constrained variable and a constant (e.g. $z = z' \vee z = \mathsf{nil} \vee z' = \mathsf{nil} \parallel const(z, z')$), and
2. constrained clauses composed of an elementary instance $D \vee l \| \varphi$ of some other constrained clause and literals l_i which are not maximal in $D \vee l$ and are elementary instances of $l \| \varphi$ (e.g. $\mathsf{select}(x, I) = \mathsf{select}(y, I) \vee z = I \vee w = I \parallel const(x, y, z, w)$).

In the first case the clause is deleted because its (dis)equalities may superpose with themselves to generate infinitely many disjunctions of (dis)equalities between constrained variables or between a constrained variable and a constant. In the second case the clause is deleted because superposition between this clause and itself may generate infinitely many new clauses of the same kind.

Sch. Deletion1 $\dfrac{S \cup \{C\|\varphi\}}{S}$

if $C\|\varphi$ is a non-unit clause containing only equalities or disequalities between constrained variables or between a constrained variable and a constant.

Sch. Deletion2 $\dfrac{S \cup \{D'\|\varphi'\} \cup \{D \vee l \vee l_1 \vee \ldots \vee l_n\|\varphi\}}{S \cup \{D'\|\varphi'\}}$

if $n \geq 0$, $D \vee l\|\varphi$ is an elementary instance of the clause $D'\|\varphi'$, and $l_i\|\varphi$ is an elementary instance of $l\|\varphi$, where l is a non-maximal literal in $D \vee l$, for $i = 1, \ldots, n$

Figure 4.3: Schematic Deletion rules of $\mathcal{SPC}$

4.5 Adequation result

Let us now present the result stating that every clause in a saturation corresponds to a schematic clause in a schematic saturation. This result was initially proved for the schematic calculus considered in [LRRT11]. The same result holds for the schematic calculus $\mathcal{SPC}$ considered here.

Theorem 9 (Correspondence between $\mathcal{PC}$ and $\mathcal{SPC}$)**.** *Let T be a theory axiomatized by a finite set $Ax(T)$ of clauses, which is saturated with respect to $\mathcal{PC}$. Let G^T_∞ be the set of all clauses in a saturation of $Ax(T) \cup G_0$ by $\mathcal{SPC}$. Then for every set S of ground flat Σ_T-literals, every clause in a saturation $Ax(T) \cup S$ by $\mathcal{PC}$ is a clause of the form*

$$C \vee l_1 \vee \ldots \vee l_n \qquad (*)$$

where

- *$n \geq 0$, and*
- *C is a constraint instance of some clause C' in G^T_∞, and*
- *l_i is*
  - *either a constraint instance of some non-maximal literal in C', or else*
  - *a constraint instance of some maximal (dis)equality between constrained variables in C', or else*
  - *a non-maximal (dis)equality between constants.*

Proof. The proof in [LRRT11] can be replayed in the same way. The proof is by induction on the length of derivations of $\mathcal{PC}$. The base case is obvious. For the inductive case, we need to show all the three facts:

1. Each clause added in the process of saturation of $Ax(T) \cup S$ by $\mathcal{PC}$ is of the form (*). This is true because we use the same expansion rules as in [LRRT11].

2. If a constrained clause $C\|\varphi$ is deleted by *Subsumption* or by *Tautology Deletion* from (or simplified by *Simplification* in) the saturation of $Ax(T) \cup G_0$ by $\mathcal{SPC}$, then all clauses containing a constraint instance of $C\|\varphi$ will also be deleted from (or simplified in) the saturation of $Ax(T) \cup S$ by $\mathcal{PC}$. Compared to [LRRT11], only *Subsumption* has slightly changed, and this new contraction rule still satisfies this fact.

3. If a constrained clause $C\|\varphi$ is deleted by *Schematic Deletion* from the saturation of $Ax(T) \cup G_0$ by $\mathcal{SPC}$, then all constraint instances of this constrained clause $C\|\varphi$ are of the form (*). Let us consider the two cases of the new *Schematic Deletion*. In the first case, the fact that $l_1 \vee \ldots \vee l_n\|\varphi$ is a non-unit clause containing only equalities or disequalities between terms of depth 0 means that it is a schematization of disjunctions of (dis)equalities between constants. It is easy to see that any disjunction of (dis)equalities between constants is of the form (*). In the second case, the fact that $D \vee l\|\varphi$ is an elementary instance of some clause $D'\|\varphi'$, and $l_i\|\varphi$ is an elementary instance of some non-maximal literal $l\|\varphi$ in $D \vee l$ means that any constraint instance of $D \vee l \vee l_1 \vee \ldots \vee l_n\|\varphi$ is of the form (*).

□

4.6 Automatic combinability

In Section 3.1 we have seen that Paramodulation Calculus provides a satisfiability procedure for some theories of classical datatypes such as lists, arrays and records. The proof of correctness is reduced to the proof of the termination of the saturation of an arbitrary set of ground flat literals and axioms of the considered theory. We have also seen that the satisfiability procedures for stably infinite theories are combinable by using the Nelson-Oppen method presented in Section 2.4.

In Section 3.3 a rewriting-based approach is used to combine signature-disjoint theories. This approach addresses an interesting modularity problem: if $\mathcal{PC}$ halts for both theory T_1 and theory T_2, can one conclude that $\mathcal{PC}$ halts for $T_1 \cup T_2$? In this case, the only problem that can prevent the termination of $\mathcal{PC}$ for the union of the two theories comes from inferences across theories, since a variable can superpose with any non-variable subterm. To circumvent this problem and ensure modular termination, it is sufficient to exclude inferences on variables across theories. To identify the clauses generating these undesirable inferences, the concept of variable-active clause has been introduced in [ABRS05].

Definition 53 (Combinable Theory)**.** *The theory T axiomatized by a finite set $Ax(T)$ of axioms is said* combinable with $\mathcal{PC}$ *if for every set S of ground flat literals, any saturation of $Ax(T) \cup S$ by $\mathcal{PC}$ is finite and does not contain any variable-active clauses.*

In this section we show that $\mathcal{SPC}$ can check whether $\mathcal{PC}$ decides some unions of finitely presented theories.

In [LRRT11] the notion of variable-active clause has been extended to a constrained clause.

Definition 54 (Variable-active Clause)**.** *A constrained clause is variable-active with respect to $>$ if one of its constraint instances is variable-active with respect to $>$.*

The correspondence between $\mathcal{SPC}$ and $\mathcal{PC}$ provides us a way to automatically check that a theory T is combinable with $\mathcal{PC}$.

Lemma 1 ([LRRT11])**.** *A theory T is combinable with $\mathcal{PC}$ if any saturation of $Ax(T) \cup G_0$ by $\mathcal{SPC}$ is finite and does not contain any variable-active clauses.*

The checking whether a clause is variable-active or not can be done syntactically. As described in Section 6.7, we detect whether the saturation contains a maximal literal $X = t$ where the variable X does not occur in t or not.

Theorem 10 ([LRRT11])**.** *Let T_1 and T_2 be two signature-disjoint theories, which are already saturated with respect to $\mathcal{PC}$. If T_1 and T_2 are combinable with $\mathcal{PC}$, then $\mathcal{PC}$ is a satisfiable procedure for $T_1 \cup T_2$.*

The notion of variable-active clause also allows us to determine properties of theories such as stable infiniteness (Definition 36) and deduction completeness (Definition 38) that we do not address in this thesis. If we want to show that a theory T is stable infinite, we have to show that a set S of ground flat literals is satisfiable in an infinite model of T. So, if we are able to show that the saturation of some set of literals does not contain an equality of the form $X = t$, where X is a variable that does not appear in t, then we have a syntactic criterion for determination of stably infiniteness.

Theorem 11 (Automatic stable infiniteness, [LRRT11])**.** *Let T be a theory axiomatized by a finite set $Ax(T)$ of clauses which is saturated with respect to $\mathcal{PC}$. Let G_∞ be the set of all clauses generated in a finite saturation of $Ax(T) \cup G_0$ by $\mathcal{SPC}$. If G_∞ does not contain variable-active clauses, then T is stable infinite.*

For the deduction completeness we have to show that the theory T axiomatized by a finite set $Ax(T)$ of Horn clauses is saturated with respect to $\mathcal{PC}$, and that the saturated set obtained by $\mathcal{SPC}$ does not contain variable-active clauses.

Theorem 12 (Automatic deduction completeness, [LRRT11])**.** *Let T be a theory axiomatized by a finite set $Ax(T)$ of Horn clauses which is saturated with respect to $\mathcal{PC}$. Let G_∞ be the set of all clauses generated in a finite saturation of $Ax(T) \cup G_0^T$ by $\mathcal{SPC}$. If G_∞ does not contain variable-active clauses, then $\mathcal{PC}$ is a deduction complete T-satisfiability procedure with respect to elementary equalities.*

For non-Horn theories the situation is more complicated since some inferences on non-unit ground elementary clauses may be blocked because of the ordering used in $\mathcal{PC}$ [LRRT11]. In order to obtain deduction completeness, Tran [Tra07] proposes to

use a splitting rule presented in [RV01]. This rule allows to activate every possible inference among elementary clauses and therefore to derive sufficiently many disjunctions of ground elementary equalities. The idea of this rule is to split any clause of the form $A \vee B$ into two clauses $A \vee p$ and $B \vee \neg p$, where p is a new propositional variable and A, B do not share any variables. Thanks to this rule, any ground elementary clause can be split into clauses containing exactly one (dis)equality and possibly new propositional variables. In this case an ordering such that p is the smallest one is considered. It allows to activate every inference on ground elementary (dis)equalities. Therefore, as soon as the set of clauses is saturated, we can compute a complete set of ground elementary clauses by eliminating all new propositional variables introduced by splitting.

Example 12 ([LRRT11]). *Let S be the following set of clauses*

$$\begin{array}{rcl} i & \neq & i' \\ \mathsf{select}(a', i') & = & e' \\ \mathsf{store}(a, i, e) & = & a' \\ \mathsf{select}(a, i') & = & e' \end{array}$$

A saturation S' of $Ax(A) \cup S$ yields $Ax(A) \cup S$ and the following clauses

$$\begin{array}{rcl} e' & = & e \vee i = i' \\ \mathsf{select}(a, i') & = & e' \\ \mathsf{select}(a, i') & = & e \vee i = i' \\ \mathsf{select}(a, J) & = & \mathsf{select}(a', J) \vee J = i \end{array}$$

It is easy to see that $\{i \neq i', e' = e \vee i = i'\}$ implies $e' = e$, and hence we have that S' entails $e' = e$. But the set $\{e' = e \vee i = i'\}$ of ground elementary clauses of S' does not entail $e' = e$ if we consider an ordering $>$ such that $e' > e > i > i'$. But by using the splitting rule, the clause $e' = e \vee i = i'$ would be split into $e' = e \vee p$ and $i = i' \vee \neg p$. Superposition between $i = i' \vee \neg p$ and $i \neq i'$ generates $i' \neq i' \vee \neg p$ to which Reflection applies and generates $\neg p$. Then the proposition variable p can be eliminated by resolving $e' = e \vee p$ and $\neg p$ to derive $e' = e$. □

More details about deduction completeness for non-Horn theories can be found in [Tra07].

4.7 Summary

In this chapter we have presented the schematic paramodulation calculus developed by Lynch and Morawska [LM02] and extended in [LRRT11]. We have slightly adapted two rules:

- the *Subsumption* rule so that the instantiation is not only a renaming but can also be a substitution instantiating constrained variables by constrained variables or constants,
- the *Schematic Deletion* rule in order to take into account the constants in the theory signature.

Due to these changes we have shown that every clause in a saturation corresponds to a schematic clause in a schematic saturation.

We have also described an approach for automatic combinability based on variable-active clauses. The notion of variable-active clause is very useful not only for modular combinability, but also for proving properties such as stably infiniteness and deduction completeness.

Chapter 5

Schematic Calculus for Integer Offsets

Contents

In this chapter, we introduce theoretical underpinnings that allows us to automatically analyse saturations computed by the paramodulation calculus modulo Integer Offsets. To this aim, we design a new schematic paramodulation calculus to describe saturations modulo Integer Offsets. Our approach requires a new form of schematization to cope with arithmetic expressions. Since paramodulation calculus for Integer Offsets presented in Section 3.4 considers only unitary clauses, the schematisation of this calculus will also. We show under which conditions the termination of schematic paramodulation implies the termination of (concrete) paramodulation. Again, the fact of considering Integer Offsets requires some specific proof arguments. In Chapter 7 we illustrate our contribution on the examples of theories considered in [NRR09c] – the theory of lists with length, the theory of lists with integer elements, and the theory of records with increment. Moreover, in this chapter we address the question of automatic modular termination for union of theories sharing function symbols of theory of Integer Offsets.

5.1 Schematic paramodulation calculus with counting operators

This section introduces a new schematic calculus taking into account the axioms of the theory of Integer Offsets within a framework based on schematic paramodulation [LRRT11,

TGRK12]. The theory of Integer Offsets allows us to build arithmetic expressions of the form $\mathsf{s}^n(t)$ for $n > 0$. The idea investigated here is to represent all terms of this form in a unique way. To this end, we consider a new operator $\mathsf{s}^+ : \text{INT} \to \text{INT}$ such that $\mathsf{s}^+(t)$ denotes the infinite set of terms $\{s^n(t) \mid n > 0\}$. The rewrite system

$$R\mathsf{s}^+ = \{\ \mathsf{s}^+(\mathsf{s}(x)) \to \mathsf{s}^+(x),\ \mathsf{s}(\mathsf{s}^+(x)) \to \mathsf{s}^+(x),\ \mathsf{s}^+(\mathsf{s}^+(x)) \to \mathsf{s}^+(x)\ \}$$

is used to simplify terms containing s^+. For each of these rules, one can easily check that the set of terms denoted by the left-hand side is included in the set of terms denoted by the right-hand side. These rules are applied eagerly whenever a new literal is generated.

5.1.1 Schematic calculus

The schematic paramodulation calculus handles schematic clauses whose definition is changing because of the new operator s^+.

Definition 55 (Schematic Clause)**.** *A* schematic clause *is a constrained clause built over the signature extended with* s^+*. An* instance of a schematic clause *is a constraint instance where each occurrence of* s^+ *is replaced by some* s^n *with* $n > 0$*.*

For a given theory T with the signature Σ_T, we define G_0 by

$$\begin{aligned} G_0 = \ & \{\bot, x = y \parallel const(x,y), x \neq y \parallel const(x,y), i = \mathsf{s}^+(j) \| const(i,j)\} \\ & \cup \bigcup_{f \in \Sigma_T} \{f(x_1, \ldots, x_n) = x_0 \parallel const(x_0, x_1, \ldots, x_n)\} \end{aligned}$$

where i, j are constrained variables of sort INT, and x, y are of the same sort.

Compared to the standard definition of G_0 introduced in [LRRT11], our G_0 contains in addition the schematic literal $i = \mathsf{s}^+(j) \| const(i,j)$.

To design a schematic calculus for Integer Offsets, we re-use the unitary version of the rules of $\mathcal{SPC}$ presented in Figure 5.1 and Figure 5.2 and complete them with two reduction rules – presented in Fig. 5.3 – which are simplification rules for Integer Offsets.

Superposition	$\dfrac{l[u'] \bowtie r \| \varphi \qquad u = t \| \psi}{\sigma(l[t] \bowtie r \| \varphi \wedge \psi)}$ **if** i) $\sigma(u) \not\preceq \sigma(t)$, ii) $\sigma(l[u']) \not\preceq \sigma(r)$, and iii) u' is not an unconstrained variable.
Reflection	$\dfrac{u' \neq u \| \psi}{\bot}$ **if** $\sigma(\psi)$ is satisfiable.

Above, u and u' are unifiable and σ is the most general unifier of u and u'.

Figure 5.1: Schematic expansion rules

Subsumption	$\dfrac{S \cup \{L \| \psi, L' \| \psi'\}}{S \cup \{L \| \psi\}}$
	if either a) $L \in Ax(T)$, ψ is empty and for some substitution σ, $L' = \sigma(L)$; or b) $L' = \sigma(L)$ and $\psi' = \sigma(\psi)$, where σ is a renaming or a mapping from constrained variables to constrained variables.
Simplification	$\dfrac{S \cup \{C[l'] \| \varphi, l = r\}}{S \cup \{C[\sigma(r)] \| \varphi, l = r\}}$
	if i) $l = r \in Ax(T)$, ii) $l' = \sigma(l)$, iii) $\sigma(l) > \sigma(r)$, and iv) $C[l'] > (\sigma(l) = \sigma(r))$.
Tautology	$\dfrac{S \cup \{u = u \| \varphi\}}{S}$
Deletion	$\dfrac{S \cup \{L \| \varphi\}}{S}$ **if** φ is unsatisfiable.
Schematic Del.	$\dfrac{S \cup \{C' \| \varphi, C[\mathsf{s}^+(t)] \| \psi\}}{S \cup \{C[\mathsf{s}^+(t)] \| \psi\}}$
	if $\sigma(\pi(C') \downarrow_{R\mathsf{s}^+}) = C[\mathsf{s}^+(t)]$, $\sigma(\varphi) = \psi$, for a renaming σ.

Figure 5.2: Schematic contraction rules

R1	$\dfrac{S \cup \{\mathsf{s}(u) = \mathsf{s}(v) \| \varphi\}}{S \cup \{u = v \| \varphi\}}$
	if u and v are ground terms.
R2	$\dfrac{S \cup \{\mathsf{s}(u) = t \| \varphi, \mathsf{s}(v) = t \| \psi\}}{S \cup \{\mathsf{s}(v) = t \| \psi, u = v \| \psi \wedge \varphi\}}$
	if u, v and t are ground terms, $\mathsf{s}(u) > t$, $\mathsf{s}(v) > t$ and $u > v$.

Figure 5.3: Ground reduction rules

Let $\mathcal{SUPC}_I$ denote the calculus depicted in Figs. 5.1, 5.2 and 5.3. Let us notice that two simplification rules $C1$ and $C2$ described in Section 3.4.2 that represent two remaining axioms of the theory of Integer Offsets do not appear in the schematic paramodulation

calculus modulo Integer Offsets. This is due to the fact that these rules produce only the empty clause $\bot$ which is already in the initial set G_0. Note that the *Reflection* rule could be omitted as well, but it is kept because it cannot be removed in the non-unitary case.

We assume that the ordering $>$ used in $\mathcal{SUPC}_I$ is $T_I - good$ (Definition 45).

In [NRR09c] the definition of *derivation* has been adapted to the paramodulation calculus for Integer Offsets. Similarly, we adapt the standard definition of derivation to the schematic paramodulation calculus modulo Integer Offsets.

Definition 56. *A derivation with respect to $\mathcal{SUPC}_I$ is a (finite or infinite) sequence of sets of literals $S_1, S_2, S_3, \ldots, S_i, \ldots$ such that, for every i, it holds that:*

1. *S_{i+1} is obtained from S_i by adding a literal obtained by the application of one of the rules in Figs. 4.1, 4.2 and 5.3 to some literals in S_i;*
2. *S_{i+1} is obtained from S_i by removing a literal according to one of the rules in Figs. 4.2 and 5.3.*

Schematic deletion

Unfortunately, the schematic saturation calculus diverges. To illustrate this point, let us take a look at the theory of lists with length (LLI) defined in Section 2.2.2 and the theory of records with increment (RII) defined in Section 2.2.4.

Example 13 (LLI)**.** *In fact, the calculus generates a schematic clause*

$$\mathsf{len}(a) = \mathsf{s}(\mathsf{len}(b)) \| const(a, b)$$

which will superpose with a renamed copy of itself, i.e. with

$$\mathsf{len}(a') = \mathsf{s}(\mathsf{len}(b')) \| const(a', b')$$

to generate a schematic clause of a new form

$$\mathsf{len}(a) = \mathsf{s}(\mathsf{s}(\mathsf{len}(b'))) \| const(a, b')$$

This process continues to generate deeper and deeper schematic clauses so that the Schematic Saturation will diverge. □

Example 14 (RII)**.** *The calculus generates a schematic clause*

$$\mathsf{rselect}_i(a) = \mathsf{s}(\mathsf{rselect}_i(b)) \| const(a, b)$$

which will superpose with a renamed copy of itself to generate a schematic clause of a new form

$$\mathsf{rselect}_i(a) = \mathsf{s}(\mathsf{s}(\mathsf{rselect}_i(b'))) \| const(a, b')$$

This process continues to generate deeper and deeper schematic clauses so that the Schematic Saturation will diverge. □

Since a term $\mathsf{s}^+(t)$ represents all the terms $\mathsf{s}^n(t)$ with $n > 0$, the idea is to replace all these terms by $\mathsf{s}^+(t)$ in the clauses containing them. To cope with this kind of clauses, we add the following *Schematic Deletion* rule, where π is a morphism replacing all the occurences of s by s^+ ($\pi(\mathsf{s}(t)) = \mathsf{s}^+(\pi(t))$ for any t, and $\pi(x) = x$ if x is a variable):

Schematic Deletion
$$\frac{C'\|\varphi \qquad C[\mathsf{s}^+(t)]\|\psi}{C[\mathsf{s}^+(t)]\|\psi}$$
if there is a renaming σ s.t.
$\sigma(\pi(C')\downarrow_{R\mathsf{s}^+}) = C[\mathsf{s}^+(t)]$ and $\sigma(\varphi) = \psi$

This rule removes a schematic literal $C'\|\varphi$ from a set of schematic literals that contains $C[\mathsf{s}^+(t)]\|\psi$ if $C'\|\varphi$ is an instance of $C[\mathsf{s}^+(t)]\|\psi$, modulo some renaming. Thus, this rule deletes constrained clauses that are not relevant for simulating inferences of $\mathcal{UPC}_I$.

The *Schematic Deletion* rule can be applied if and only if the initial set of schemas of ground flat literals G_0 is extended with the non-flat schematic literal $\mathsf{len}(a) = \mathsf{s}^+(\mathsf{len}(b)) \parallel const(a, b)$ for the theory of lists with length, and $\mathsf{rselect}_i(a) = \mathsf{s}^+(\mathsf{rselect}_i(b)) \parallel const(a, b)$ for the theory of records with increment.

Thanks to these two changes, the schematic saturation terminates for the theory of lists with length and for the theory of records with increment.

More generally, we propose to extend G_0 given above with all the non-flat schematic literals of the form $u = \mathsf{s}^+(v)\|\varphi$ where u and v are two flat terms of sort INT over the initial signature without s (and, of course, without s^+) whose variables are all constrained (are in φ). Finally, the set of ground schematic literals G_0 is defined as follows:

$$\begin{aligned} G_0 = \ & \{\bot, x = y \parallel const(x, y), x \neq y \parallel const(x, y), u = \mathsf{s}^+(v)\|\varphi\} \\ & \cup \bigcup_{f \in \Sigma_T} \{f(x_1, \ldots, x_n) = x_0 \parallel const(x_0, x_1, \ldots, x_n)\} \end{aligned}$$

where u, v are flat terms of sort INT whose variables are all constrained, and x, y are constrained variables of the same sort.

5.1.2 Adequation result

We show that any clause in a saturation obtained by $\mathcal{UPC}_I$ is an instance of a schematic clause in a schematic saturation obtained by $\mathcal{SUPC}_I$, under the following assumption.

Assumption 1. *Let SC be any set of schematic clauses generated by $\mathcal{SUPC}_I$. If an s^+-rooted term (resp. s-rooted term) occurs in a term u which is maximal in an equality $u = t$ in SC, then there is no s-rooted term (resp. s^+-rooted term) occurring in a term $l[u']$ which is maximal in a clause $l[u'] \bowtie r$ in SC.*

Without Assumption 1 we would need a specific unification algorithm to handle literals involving s and s^+. Thanks to it we can continue applying the superposition rule with syntactic unification.

Theorem 13. *Let T be a theory axiomatized by a finite set $Ax(T)$ of literals, which is saturated with respect to $\mathcal{UPC}_I$. Let G_∞ be the set of all schematic clauses in a saturation of $Ax(T) \cup G_0$ by $\mathcal{SUPC}_I$. Then for every set S of ground flat literals, every clause in a saturation of $Ax(T) \cup S$ by $\mathcal{UPC}_I$ is an instance of a schematic clause in G_∞.*

Proof. The proof is an adaptation of the one of [LRRT11, Theorem 2]. The proof is by induction on the length of derivations of $\mathcal{UPC}_I$. The base case is obvious. For the inductive case, we need to show two facts:

(1) each clause added in the process of saturation of $Ax(T) \cup S$ is an instance of a schematic clause in the saturation G_∞ of $Ax(T) \cup G_0$ by $\mathcal{SUPC}_I$, and

(2) if a clause is deleted by *Subsumption*, *Tautology* or *Deletion* from (or simplified by *Simplification*/reduced by *Reduction* in) G_∞, then all instances of the latter will also be deleted from (or simplified/reduced in) the saturation of $Ax(T) \cup S$ by $\mathcal{UPC}_I$.

Moreover, because of additional rewriting rules for terms containing s^+, we have to check another fact:

(3) Any such rule preserves the set of instances of any schematic clause.

Proof of (1). Consider the *Superposition* rule of $\mathcal{UPC}_I$. By induction hypothesis $l[u'] \bowtie r$ and $u = t$ are instances of schematic clauses in G_∞, i.e. there is some schematic clause $\hat{D}$ in G_∞ such that $l[u'] \bowtie r$ is an instance of $\hat{D}$, and a schematic clause $\hat{E}$ in G_∞ such that $u = t$ is an instance of $\hat{E}$. Two cases can be distinguished:

(∗) If there is no occurrence of s in u or u', then there exists a *Superposition* inference of $\mathcal{SUPC}_I$ in G_∞ whose premises are $\hat{D}$ and $\hat{E}$, and whose conclusion is a schematic clause $\hat{C}$ such that $\sigma(l[t] \bowtie r)$ is an instance of $\hat{C}$, where σ denotes the most general unifier of u and u'.

(∗∗) If there are occurrences of s in both u and u', two additional subcases can be considered. Assume that $\hat{u}$ and $\hat{u'}$ denote the schematic terms of u and u'.

1. If $\hat{u}$ and $\hat{u'}$ contain only s^+-rooted terms (resp. s-rooted terms), then we proceed as in (∗).
2. If $\hat{u}$ contains an s^+-rooted term (resp. s-rooted term) and $\hat{u'}$ contains an s-rooted term (resp. s^+-rooted term), then $\hat{u}$ may not unify with $\hat{u'}$ since we use syntactic unification, while u and u' may unify. This subcase is avoided by Assumption 1 and the side conditions of the *Superposition* rule.

Reflection of $\mathcal{UPC}_I$ can be handled in a way similar to *Superposition* and is therefore omitted.

Proof of (2). Let us consider *Subsumption* of $\mathcal{SUPC}_I$. For the case (a), let us assume that there are a schematic clause A deleted from G_∞ and a clause B in the saturation of $Ax(T) \cup S$ by $\mathcal{UPC}_I$, which is an instance of the schematic clause A. Then there must exist a clause $C \in Ax(T)$ and some substitution θ such that $\theta(C) \subseteq A$. Since all the clauses in $Ax(T)$ persist, there must be a substitution θ' such that $\theta'(C) \subseteq B$. Thereby B must also be deleted from the saturation of $Ax(T) \cup S$ by $\mathcal{UPC}_I$, and we are done. The case (b) of *Subsumption* is just a matter of deleting duplicates and leaving only more general constrained literals.

Since axioms do not contain the s^+ symbol, a similar argument can be used for *Simplification* of $\mathcal{SUPC}_I$. Assume that there is a schematic clause $C[l'] \parallel \varphi$ in G_∞ simplified by an equality $l = r$ ($l = r \in Ax(T)$) into $C[\theta(r)] \parallel \varphi$. Let σ be a substitution such that $\sigma(C[l'])$ is an instance of $C[l'] \parallel \varphi$. Since $l = r$ persists in the saturation of $Ax(T) \cup S$ by $\mathcal{UPC}_I$, there must be a simplification of $\sigma(C[l']) = \sigma(C)[\sigma(\theta(l))]$ by $l = r$ into $\sigma(C)[\sigma(\theta(r))] = \sigma(C[\theta(r)])$, which is an instance of $C[\theta(r)] \parallel \varphi$.

For the *Tautology Deletion* rule of $\mathcal{SUPC}_I$, it is easy to see that a constraint instance of a tautology is also a tautology. For the *Deletion* rule of $\mathcal{SUPC}_I$, notice that clauses with an unsatisfiable constraint have no instances.

For the reduction rule $R1$ of $\mathcal{SUPC}_I$, it is easy to see that an instance of a schematic clause $\mathsf{s}(u) = \mathsf{s}(v)$ will also reduce a root symbol s. For the reduction rule $R2$ of $\mathcal{SUPC}_I$, a similar argument can be given.

Proof of (3). Let $C \downarrow_{R\mathsf{s}^+}$ be the clause obtained from C by replacing the terms occurring in C with their normal forms w.r.t. $R\mathsf{s}^+$. The set of (concrete) clauses schematized by a schematic clause C is included in the set of (concrete) clauses schematized by $C \downarrow_{R\mathsf{s}^+}$, because a similar inclusion holds for all the terms in C and all the rules in $R\mathsf{s}^+$. □

5.1.3 Application to the analysis of paramodulation

Contrary to the standard case, a schematized saturation may represent an infinite set of clauses since the term $\mathsf{s}^+(t)$ represents all the terms $s^n(t)$ with $n \geq 1$. The difficulty is then to prove the termination in this case. In [NRR09c], the termination proofs do not only rely on the fact that there are finitely many forms of clauses generated by the paramodulation calculus. In addition, the following proof argument is used: any new ground literal is strictly smaller than the biggest ground literal in the input set. Similarly, whereas the schematic paramodulation allows computing the different forms of clauses generated by paramodulation, we still need an additional analysis to conclude that the paramodulation calculus terminates. Fortunately, this analysis can be easily performed for some cases. We investigate hereafter a new solution where the analysis is restricted to the (few) schematic equalities containing s^+ that occur in the (finite) schematic saturation.

Assumption 2. *A schematic equality containing* s^+ *cannot be instantiated with different values of the exponent of* s *in a saturation of a satisfiable input.*

Thanks to Assumption 2, there are only finitely many possible instances in the saturation of a satisfiable input. For instance, we cannot have both $i = \mathsf{s}(j)$ and $i = \mathsf{s}^2(j)$ in the saturation of a satisfiable input due to the acyclicity axiom.

With respect to disequalities, we restrict us to the case where $\mathcal{UPC}_I$ does not generate new disequalities having an occurrence of s: the simplification of an input disequality is the only way to have a disequality with an occurrence of s introduced in a derivation with $\mathcal{UPC}_I$. This restriction is satisfied in the following cases:

- The set of axioms of the theory contains only equalities.
- The set of axioms of the theory contains some disequalities of a given sort, say D, such that it is not possible to build terms of sort D containing s.

Hence, this restriction is satisfied for the theories we are interested in.

Theorem 14. *Assume that $\mathcal{UPC}_I$ does not generate new disequalities having an occurrence of* s. *If $\mathcal{SUPC}_I$ generates a finite schematic saturation such that all its schematic equalities satisfy Assumption 2, then $\mathcal{UPC}_I$ terminates on any input set of ground literals.*

Proof. Consider a satisfiable input. Let n_d (resp. n_e) be the number of disequalities (resp. equalities) obtained from the schematic disequalities (resp. equalities) in the finite schematic saturation by considering all possible instantiations of constrained variables by the finitely many constants in the input. The number of clauses occurring in the saturation of the input can be bounded as follows:

1. By hypothesis, the number of disequalities cannot be greater than $n_d + i_d$, where i_d denotes the number of disequalities in the input set.
2. Consider the equalities. According to Assumption 2, the number of equalities is bounded by n_e.

Consequently, the paramodulation calculus computes a finite saturated set of clauses and terminates. □

One can remark that our restriction on the generation of new disequalities in Theorem 14 is expressed with $\mathcal{UPC}_I$, but not with $\mathcal{SUPC}_I$. We adopt this solution because this restriction is easy to satisfy in practice and it is sufficient for our need. Of course, an interesting problem would be to find a way at the schematic level to ensure the boundedness of the generated disequalities. A possible solution could be envisioned when all the generated schematic literals are ground.

5.2 Automatic modular termination

The combination of theories extending T_I has been discussed in Section 3.4.3. Let us now study the question of automatic modular termination. Firstly let us define the notion of safe schematic saturation.

Definition 57 (Safe saturation)**.** *The schematic saturation is safe if it does not contain (1) variable-active clauses, and (2) non-ground clauses of the form* $\mathsf{s}(u) = t$ *(resp.* $\mathsf{s}^+(u) = t$*), where* $\mathsf{s}(u)$ *(resp.* $\mathsf{s}^+(u)$*) is maximal.*

The paramodulation modulo Integer Offsets terminates for the union of two theories sharing only function symbols of theory of Integer Offsets if these two theories are safely terminating.

Theorem 15 (Automatic modular termination)**.** *Let T_1 and T_2 be two theories sharing only function symbols of T_I, which are already saturated with respect to $\mathcal{UPC}_I$. Assume that any schematic saturation of $Ax(T_i) \cup G_0^{T_i}$ by $\mathcal{SUPC}_I$ is finite and safe, for $i = 1, 2$. Then, $\mathcal{UPC}_I$ is a satisfiability procedure for $T_1 \cup T_2$.*

The sufficient argument for the proof of this theory is that any instance of a safe schematic clause is safe.

5.3 Summary

In this chapter we have introduced a new schematic calculus integrating the axioms of the Integer Offsets theory into a framework based on schematic paramodulation. In this context, introducing the s^+ operator together with rewriting rules for terms containing s^+ fits well with automatic verification needs.

The calculus $\mathcal{SUPC}_I$ with a new form of schematization for arithmetic expressions is used in Chapter 7 to automatically prove the termination of paramodulation modulo Integer Offsets for data structures equipped with counting operators.

Thanks to schematic saturation, we can automatically check the modular termination for unions of theories sharing only function symbols of theory of Integer Offsets. As we could see, we need only to check that the saturation does not contain clauses of a specific form given in Definition 57.

Chapter 6

Implementation

Contents

This chapter describes the main ideas and principles of our implementation of schematic paramodulation calculi. The goal is to implement the calculi so that the user could easily modify the code corresponding to an executable specification. Implementing this schematic calculus in an off-the-shelf equational theorem prover like the E prover [**?**] or SPASS [**?**] would be a difficult and less interesting task, since the developer and the user would have to understand a complex piece of code which is the result of years of engineering and debugging. To make the task easier another quite natural solution is to use a logical framework since calculi are defined by inference systems. This is why we propose to prototype schematic superposition calculi by using a rule-based logical framework.

Our goal was to get a rule-based program as close as possible to the formal specification. To achieve this goal, we choose the Maude system because it includes support for unification and narrowing, which are key operations of the calculus of interest, and because the Maude meta-level provides a flexible way to control the application of rules and powerful search mechanisms.

Our tool contains about 3000 lines and 55 Maude modules. It is composed of 12 files including the main one. To define a theory the user needs to declare its signature, a set of axioms and an initial set of ground flat literal.

Our implementation makes use of many Full-Maude functions working at the meta-level:

- `metaUnify` for unifying two terms,
- `metaXmatch` for matching two terms,
- `metaFrewrite` for rewriting a given term by a given rule,
- `metaENarrowShowAll` for narrowing,
- `metaSearch` for searching for a saturated set of schematic clauses,
- and many others.

Moreover, to define some parameterized modules we use the genericity provided by the Maude system.

In the framework of this thesis, two schematic paramodulation calculi have been implemented: the standard one, and the schematic paramodulation calculus modulo Integer Offsets. A big part of the code is common to both calculi.

The chapter is organized as follows. Section 6.1 presents the implementation of literals, clauses and constrained clauses. Thanks to our tool the user can see how new constrained clauses are generated. Section 6.2 shows how this functionality is implemented. In Section 6.3 we explain how to proceed to define a theory within our tool. Section 6.4 details the implementation of contraction and expansion rules. The implementation of functions that allow us to get a saturated set of schematic clauses is presented in Section 6.5. Section 6.6 shows how the ordering used in rules of schematic paramodulation calculus is encoded. The functions that allow checking whether the considered theory is combinable are presented in Section 6.7. The last section summarizes this chapter.

6.1 Data representation

In order to represent terms, literals, clauses and constrained clauses, we exploit the Maude reflection feature by using the sort `Term` for two reasons: firstly, we would like to use the Maude existing features as much as possible, and secondly, our implementation works at the meta-level, i.e. its functions operate on meta-terms with Maude sort `Term`.

6.1.1 Term

The base cases in the metarepresentation of terms are given by the subsorts `Constant` and `Variable` of the sort `Term`. Constants in Maude are quoted identifiers composed of the constant name and its type separated by a ".", e.g., `'a.Lists`. Similarly, variables in Maude are quoted identifiers composed of variable name and its type separated by a ":", e.g., `'a:Lists`. A term of depth one or more in Maude is a quoted identifier of the form `_[_]`, where the first underscore is some quoted identifier and the second one is a list of terms separated with a comma. For example, the term `car(cons(E, V))`, where `V` is a variable of sort LISTS and `E` is a constant of sort ELEM, is represented by `'car[cons['E.Elem, 'V:Lists]]`.

6.1.2 Literals

To define the new sort `Literal` for literals, we exploit the Maude reflection feature by using the sort `Term`. A module `META-TERM` ("prelude.maude"), where this sort is defined, should be protected in the module `LITERAL` in order to use the sort `Term`.

```
fmod LITERAL is
  pr META-TERM .
  sort Literal .
  op _equals_ : Term Term -> Literal [comm] .
  op _!=_     : Term Term -> Literal [comm] .
endfm
```

The attribute `[comm]` declares that the infix binary symbols `equals` and `!=` for respectively equality and disequality are commutative.

For sets of literals we define the sort `SetLit` by instantiating the polymorphic sort `Set{X}` defined in the parameterized module `SET{X :: TRIV}` of the prelude of Maude, as follows:

```
view Literal from TRIV to LITERAL is
  sort Elt to Literal .
endv

fmod SETLIT is
  pr LITERAL .
  pr SET{Literal} * (sort Set{Literal} to SetLit) .
endfm
```

The first three lines declare that the sort `Literal` can be viewed as the sort of elements `Elt` provided by the theory `TRIV`. This Maude `view` is named `Literal`. It is used in the module `SETLIT` to instantiate `Set{X}` as `Set{Literal}`. Finally, the sort `SetLit` is a renaming of the sort `Set{Literal}`. Consequently, the sets in this sort can be built by using the constant `empty`, and by using an associative, commutative, and idempotent union operator, written `_,_`. A singleton set is identified with its element (`Literal` is a subsort of `SetLit`).

6.1.3 Clauses

A clause is a set of literals. The sort `Clause` is defined by the operator `clause` that takes a set of literals as a parameter.

```
fmod CLAUSE is
  pr SETLIT .
  sort Clause .
  op clause : SetLit -> Clause .
endfm
```

6.1.4 Constraints

A constraint is implemented as a set of atomic constraints of the form `const(`t`)` where t is a term.

```
fmod ATOMIC-CONSTRAINT is
  pr META-TERM .
  sort AtomicConstraint .
  op const : Term -> AtomicConstraint .
endfm
```

The same technique (as for sets of literals) is used to define sets of atomic constraints.

```
view AtomicConstraint from TRIV to ATOMIC-CONSTRAINT is
  sort Elt to AtomicConstraint .
endv

fmod SETCONSTR is
  pr ATOMIC-CONSTRAINT .
  pr SET{AtomicConstraint} * (sort Set{AtomicConstraint} to Constraint) .
endfm
```

The sort `Constraint` is a renaming of the sort `Set{AtomicConstraint}`.

6.1.5 Constrained clauses

The sort `SClause` of constrained clauses is declared by

```
sort SClause .
op _||_ : Clause Constraint -> SClause .
op emptySClause : -> SClause .
```

where the infix operator `||` constructs a constrained clause from a clause and a constraint of sort `Constraint`.

A simple example of constrained clause could be

```
clause('X:Lists equals 'b:Lists) || const('b:Lists)
```

where `X` is a universally quantified variable of sort LISTS, and b is a constrained variable of sort LISTS.

An axiom is represented as a constrained clause with an empty constraint. For example, the axiom $\mathsf{car}(\mathsf{cons}(X, Y)) = X$ of the theory of lists has the following profile:

```
clause('car['cons['X:Elem,'Y:Lists]] equals 'X:Elem) || empty
```

6.2 Traces

An important feature of our tool consists in providing a trace indicating the name of the applied rule and the constrained clauses it is applied to at each derivation step. This trace helps understanding the origin of each new constrained clause. With this information, the user could replay the derivation manually if necessary.

Each constrained clause carries its trace. Let us take as an example the superpositon rule:

$$\frac{C \vee l[u'] \bowtie r \| \varphi \qquad D \vee u = t \| \psi}{\sigma(C \vee D \vee l[t] \bowtie r \| \varphi \wedge \psi)}$$

The expression

$$\texttt{sup(}C_1\texttt{,}\ C_2\texttt{,}\ u\texttt{,}\ l[u']\texttt{,}\ Ctx\texttt{)}\ \texttt{gives}\ C_3$$

means that the constrained clause $C_3 = \sigma(C \vee D \vee l[t] \bowtie r \| \varphi \wedge \psi)$ is derived from the constrained clauses $C_1 = (C \vee l[u'] \bowtie r \| \varphi)$ and $C_2 = (D \vee u = t \| \psi)$ by superposing the term u from C_2 in the term $l[u']$ from C_1 at the context $Ctx = l[\,]$, where the rewriting has taken place.

The sorts `STrace` and `TracedSClause` of traces and traced constrained clauses are defined by

```
sort STrace .
sort TracedSClause .
subsort SClause < STrace .

op sup : STrace STrace Substitution Substitution -> STrace .
op sup : STrace STrace Term Term Context -> STrace .
op refl : STrace -> STrace .
op ef : STrace -> STrace .
op simpl : STrace STrace -> STrace .

op _gives_ : STrace SClause -> TracedSClause .
```

The subsort condition considers constrained clauses as traces. It is needed in order to declare the initial set of traced constrained clauses and axioms, where the constrained clause and its trace are the same (see Section 6.3.2 and 6.3.3).

The operators `sup`, `refl` and `ef` associate a trace to each expansion rule (respectively, superposition, reflection and equality factoring). One can remark that the operator `sup` has two profiles. The first one is needed in order to store the traces of the contrained clauses between which Superposition has been applied and also it stores two substitutions. It is needed because of the renaming of the constrained clauses which is performed before applying the superposition rule. The second one is more detailed: It shows not only the traces of the superposed constrained clauses but also the terms to which superposition has been applied, and the context of the application. More details can be found in Section 6.4.3.

The operator `simpl` associates a trace with the *Simplification* rule, a contraction rule that does not eliminate a clause but rewrites it into a simpler one. The infix operator `gives` builds a traced constrained clause from a trace of sort `STrace`, and a constrained clause.

The sort `SetTracedSClause` of sets of traced constrained clauses is defined in a similar way as the sort `SetLit` of sets of literals.

Let us now show an example and consider the trace `sup` that stands for the superposition rule. Let us consider two constrained clauses from the theory of lists with length presented in Section 2.2.2: an axiom

```
clause(len(cons(X,Y)) = succ(len(Y))) || empty gives
  clause(len(cons(X,Y)) = succ(len(Y))) || empty
```

and a traced constrained clause from G_0

```
clause(cons(e,a) = b) || const(e,a,b) gives
  clause(cons(e,a) = b) || const(e,a,b)
```

Superposition between these two clauses yields a traced constrained clause of the following form

```
sup(
  clause(len(cons(X,Y)) = succ(len(Y))) || empty,
```

```
  clause(cons(e,a) = b) || const(e,a,b),
  cons(e,a),
  len(cons(X,Y)),
  len([])
) gives
clause(len(b) = succ(len(a))) || const(a,b)
```

From this trace, we can see that the new constrained clause is obtained by superposing the term `cons(e,a)` in the term `len(cons(X,Y))` at the position shown by the context `len([])`.

6.2.1 Clause labelling

At one point the trace can become so long and heavy that it would be unreadable. That is why, we propose to give the label to each clause and to rewrite the clauses in the trace by their labels. The function `givesIndex` does it. It gives a number to each traced constrained clause. This function is encoded as follows:

```
op givesIndex : SetTracedSClause Nat -> SetTracedSClause .
eq givesIndex(empty, N) = empty .
eq givesIndex((Tr gives SC, STSC), N) =
 Tr gives SC as N, givesIndex(STSC, N + 1) .
```

The first equation returns an empty set if the empty set of traced constrained clauses is given. The second equation returns the set of traced constrained clauses with the following profile:

```
op _gives_as_ : STrace SClause Nat -> TracedSClause .
```

where the last underscore stands for the natural number. After indexing the clauses, the following two rules are repeatedly applied until all the traces are flattened:

```
rl [initial] : SC1 gives SC1 as N => SC1 as N .

crl [trace] : Tr1 gives SC1 as N1, Tr2 gives SC2 as N2 =>
 Tr1 gives SC1 as N1,
 newTrace(Tr1 gives SC1 as N1, Tr2) gives SC2 as N2
  if mFr(Tr1 gives SC1 as N1, Tr2) =/= failure .
```

The first rule `initial` replaces the initial traced constrained clauses of the form `SC1 gives SC1 as N` by traced constrained clauses of the form `SC1 as N`, where `SC1` is a constrained clause and `N` is its number. The second rule `trace` makes one step towards flattening the trace `Tr2` of the second traced constrained clause by the trace `Tr1` of the first traced constrained clause by replacing the trace `Tr1` in `Tr2` with its label (number) `N1`. The condition of this rule determines whether the rewriting can be performed with the given data. The rewriting is performed by the function `mFr` that uses the Maude function `metaFrewrite`.

6.2.2 Flattening

To get fully flattened traces, i.e. to determine when no rule can be applied anymore, we apply a function `searchP` that uses the Maude function `metaSearch` with the `'*` parameter.

```
op searchP : SetTracedSClause Nat -> SetTracedSClause .
 eq searchP(STSC, N) = downTerm(getTerm(
     metaSearch(upModule('TRACE, false), upTerm(STSC),
     'S:SetTracedSClause, nil, '*, unbounded, N)), errorTr) .
```

It tries to reach `N`-th set of traced constrained clauses from an initial one, called `STSC`, by applying the flattening rules.

The function `satP` implements a fix point algorithm in order to reach a set of traced constrained clauses with flattened traces. When no clause can be rewriten by its label, it returns the unchanged set.

```
op satP : SetTracedSClause -> SetTracedSClause .
eq satP(STSC) =
  if searchP(STSC, 1) == errorTr
    then STSC
    else
      if searchP(STSC, 1) =/= STSC
       then satP(searchP(STSC, 1))
      else STSC
      fi
   fi .
```

6.3 Theories

The schematic paramodulation calculus takes as input the set G_0 of constrained clauses and the set Ax of axioms. Thus, to declare a theory, the user has to define a file for its signature, a file for its set of axioms, and a file for the set of initial constrained clauses. Let us show what each of these files contains.

6.3.1 Signature

To declare the signature, the user has to define the set of sorts of the theory, the set of function symbols, and the precedence order on symbols.

Sorts

The underlying logic of Maude is order-sorted, admitting a subsort ordering, whereas the underlying logic of our calculus $\mathcal{SPC}$ is many-sorted, i.e. there is no subsort relation between sorts in the addressed theories. For instance, the theory of lists with length presented in Section 2.2.2 has the following set of sorts $S = \{\text{LISTS}, \text{ELEM}, \text{INTS}\}$ and the following signature $\Sigma = \{\mathsf{car} : \text{LISTS} \to \text{ELEM}, \mathsf{cdr} : \text{LISTS} \to \text{LISTS}, \mathsf{cons} : \text{ELEM} \times$

LISTS → LISTS, len : LISTS → INT, s : INT → INT}. The sorts LISTS, ELEM and INTS are implemented in Maude in the module `SORT` by the declaration

```
sorts Lists Elem Ints .
```

Moreover, no subsort relation is declared between these Maude sorts. This condition guarantees that the order-sorted features of Maude (pattern-matching, unification, etc) behave as many-sorted ones on the set of Maude sorts associated with S.

We take profit of subsorting by declaring all these Maude sorts as subsorts of a new sort, named `Tops`, as follows:

```
sort Tops .
subsorts Lists Elem Ints < Tops .
```

The sort `Tops` is a main sort in our tool. Firstly, it is used as a sort for unsorted theories to declare the signature. Secondly, it is used as a "super" sort for many-sorted theories in order to avoid the definition of, for example, an equality for each sort in the considered theory.

Function symbols

A module called `SIGNATURE` contains the signature of the theory. For example, the many-sorted theory of lists with length has the following signature

```
op car : Lists -> Elem .
op cdr : Lists -> Lists .
op cons : Elem Lists -> Lists .
op len : Lists -> Ints .
op succ : Ints -> Ints .
```

where `succ` stands for `s`. The unsorted theory of lists with length has the same functional symbols but only one sort. Its signature is declared by

```
op car : Tops -> Tops .
op cdr : Tops -> Tops .
op cons : Tops Tops -> Tops .
op len : Tops -> Tops .
op succ : Tops -> Tops .
```

Precedence order

A module `PRECEDENCE` declares the precedence order on symbols. This precedence is specified by a list of quoted identifiers, named `listOfSymbols`. For example, this list is defined by

```
op listOfSymbols : -> QidList .
eq listOfSymbols = ('cons 'cdr 'car 'c.Lists 'c.Elem 'len 'c.Ints 'succ) .
```

for the theory of lists with length. Note that for each sort $s \in S$, there should exist at least one constant of this sort in the precedence, e.g., `'c.Lists` is a constant of sort LISTS.

This precedence order is used in the implementation of ordering (explained in Section 6.6).

6.3.2 Axioms

An axiom is a constrained clause whose constraint is empty. A traced axiom is a traced constrained clause whose trace is the clause itself. Since subsumption and simplification use the axioms in their rules, the axioms are declared globally. The modules where subsumption and simplification auxiliary functions are defined (such as SUBSUM-AUX-FCT and SIMPL-AUX-FCT), are parameterized with the view

```
view Ax-view from THAXIOMS to AX-MOD is
  op ax to ax .
endv
```

This view maps entities from the interface theory (the theory THAXIOMS that declares the operator ax)

```
fth THAXIOMS is
  pr SET-SCLAUSE-TRACE .
  op ax : -> SetTracedSClause .
endfth
```

to the corresponding entities in the parameter module (the module AX-MOD, where the operator ax is defined). For instance, the set of axioms for the theory of lists with length is defined as follows:

```
fmod AX-MOD is
  pr SET-SCLAUSE-TRACE .
  op ax : -> SetTracedSClause .

  eq ax = (
   clause('X1:Elem equals 'car['cons['X1:Elem,'Y1:Lists]]) || empty gives
    clause('X1:Elem equals 'car['cons['X1:Elem,'Y1:Lists]]) || empty,
   clause('X2:Lists equals 'cdr['cons['Y2:Elem,'X2:Lists]]) || empty gives
    clause('X2:Lists equals 'cdr['cons['Y2:Elem,'X2:Lists]]) || empty,
   clause('len['cons['X3:Elem, 'Y3:Lists]] equals 'succ['len['Y3:Lists]])
    || empty gives
    clause('len['cons['X3:Elem, 'Y3:Lists]] equals 'succ['len['Y3:Lists]])
     || empty
  ) .
endfm
```

where X1, Y2 and X3 are universally quantified variables of sort ELEM, and Y1, X2 and Y3 are universally quantified variables of sort LISTS. Note that we use capital letters to declare universally quantified variables.

Then, when a parameterized module such as SUBSUM-AUX-FCT{X :: THAXIOMS} is imported into another module, we write pr SUBSUM-AUX-FCT{Ax-view}. This helps in specifying and declaring some modules globally, for their different use.

6.3.3 Initial set of constrained clauses

The initial set of constrained clauses is declared in the module `INIT-SET`. It is composed of traced constrained clauses, whose trace is the clause itself. The empty clause is denoted by `emptySClause`, and therefore, the traced empty constrained clause is of the form `emptySClause gives emptySClause`. An extract from the initial set of constrained clauses for the theory of lists with length can be found below:

```
fmod INIT-SET is
  pr SET-SCLAUSE-TRACE .

  op initSet : -> SetTracedSClause .
  eq initSet = (
   emptySClause gives emptySClause,
   clause('g:Lists != 'f:Lists) || (const('g:Lists), const('f:Lists)) gives
    clause('g:Lists != 'f:Lists) || (const('g:Lists), const('f:Lists)),
   clause('car['e:Lists] equals 'k:Elem) || (const('e:Lists), const('k:Elem))
    gives
    clause('car['e:Lists] equals 'k:Elem) || (const('e:Lists), const('k:Elem)),

   ...
  ) .
endfm
```

The whole theory which is composed of the set of axioms and the set of initial constrained clauses is declared by

```
fmod THEORY is
  pr AX-MOD .
  pr INIT-SET .

  op thList : -> SetTracedSClause .
  eq thList = (ax, initSet) .
endfm
```

6.4 Inference rules

This section presents the encoding of $\mathcal{SPC}$. Let us emphasize two main ideas of this encoding: 1) inference rules are translated into rewrite rules, and 2) rule application is controlled thanks to specially designed states. The following description of this encoding starts with the translation of the contraction rules into rewrite rules. Then, it continues with the expansion rules, whose fair application strategy is encoded by using a notion of state together with rules to specify the transitions between states.

6.4.1 Contraction rules

We first present the encoding of the contraction rules used for both $\mathcal{SPC}$ and $\mathcal{SUPC}_I$, namely *Tautology*, *Deletion*, *Subsumption* and *Simplification*. Then, we present the enconding of *Schematic Deletion* rules for $\mathcal{SPC}$ and for $\mathcal{SUPC}_I$. And finally, we present the encoding of *Reduction* rules for $\mathcal{SUPC}_I$. All these rules are defined in the module called `CONTR`.

Tautology rule

The inference rule *Tautology* is simply encoded by the rewrite rule

```
rl [tautology] : Tr gives clause((SL, U equals U)) || Phi => empty .
```

where `Tr` is a clause trace, `SL` is a set of literals, `U` is a term and `Phi` is a constraint.

Deletion rule

The inference rule *Deletion* is encoded by the conditional rewrite rule

```
crl [del] : Tr gives C || Phi => empty
  if isSatisfiableSet(Phi) == false .
```

where the function `isSatisfiableSet` checks if a given constraint holds, i.e. none of the terms it constraints is compound. It is encoded by

```
op isSatisfiableSet : Constraint -> Bool .
eq isSatisfiableSet(empty) = true .
eq isSatisfiableSet((const(U), Phi)) =
  isVariable(U) and isSatisfiableSet(Phi) .
```

where the function `isVariable` checks whether an atomic constraint contains a variable or not.

```
var Vr : Variable .
op isVariable : Term -> Bool .
eq isVariable(Vr) = true .
eq isVariable(T) = false [owise] .
```

Subsumption rule

The first case of the *Subsumption* rule uses the global variable `ax` (presented in Section 6.3.2) that represents the set of axioms of the current theory.

```
crl [subsum1] : Tr gives C || Phi => empty
  if ax isSubsum (Tr gives C || Phi) .
```

The rule condition checks whether the clause `C` can be subsumed by one of the axioms in `ax`. The function `isSubsum` is encoded as follows:

```
op _isSubsum_ : SetTracedSClause SClause -> Bool .
eq empty isSubsum C' = false .
eq (Tr gives SC1, STSC) isSubsum C' =
  clauseMatch(SC1, C') or (STSC isSubsum C') .
```

The function `clauseMatch` calls the Maude function `metaXmatch` that tries to match two clauses represented as terms.

The other two cases of the *Subsumption* rule are similar. In the first case, the constrained clause is eliminated if it is a renaming of another one.

```
crl [subsum2] : Tr1 gives C1 || Phi1, Tr2 gives C2 || Phi2
                => Tr1 gives C1 || Phi1
  if isRename(C1 || Phi1, C2 || Phi2) .
```

In the last case of the *Subsumption* rule one constrained clause should be an elementary instance of another one and therefore eliminated.

```
crl [subsum3] : Tr1 gives C1 || Phi1, Tr2 gives C2 || Phi2
                => Tr1 gives C1 || Phi1
  if isElementaryInstance(C1 || Phi1, C2 || Phi2) .
```

The function `isElementaryInstance` checks whether the constrained clause `C2 || Phi2` is an elementary instance of `C1 || Phi1` by determining the existence of a substitution mapping the first clause into the second one, and the constraint of the first clause into the constraint of the second one. Moreover, the obtained substitution should replace variables by variables or constants. The mapping is performed by the Maude function `metaXmatch` that returns an element of sort `MatchPair` composed of a substitution and a context.

The function `isRename` is similar to the function `isElementaryInstance` except that in addition it checks whether the correspondence between the replaced variables and the replacing ones in the "matching" substitution is one to one.

Simplification rule

The *Simplification* rule does not eliminate the clause, but reduces it to a simpler one thanks to some equality axiom. This rule is encoded as a conditional rewrite rule using the set of axioms `ax`.

```
var newTC : TracedSClause .

crl [simpl] : Tr gives C || Phi
              => simpl(Tr, axTr) gives newC || newPhi
  if newTC := applyUnitAx(C, Phi, ax) /\
     newTC =/= NoSimpl /\
     axTr gives newC || newPhi := newTC .
```

The function `applyUnitAx` considers each literal in the clause `C` and tries to rewrite it into a simpler one by using one of the equality axioms from `ax`. This function uses the Maude function `metaXmatch` that returns a context and a substitution. By knowing the context, we can rewrite the given literal into a simpler one at the right position. The function `applyUnitAx` returns a simplified constrained clause whose trace describes the axiom. If *Simplification* does not apply, the function `applySimpl` returns the constant `NoSimpl` of sort `TracedSClause`.

Schematic Deletion rules for $\mathcal{SPC}$

The first case of *Schematic Deletion* rule for $\mathcal{SPC}$ is encoded by

```
crl [sd1] : Tr gives C || Phi => empty
  if sd1Cond(C || Phi) .
```

It removes clauses containing only equalities and disequalities between terms of depth 0. In such literals all the variables are necessarily constrained. The function `sd1Cond` checks these requirements.

```
op sd1Cond : SClause -> Bool .
eq sd1Cond(clause(SL) || Phi) =
  if aritySL(SL) > 1 and depthSL(SL) == 0 then
    const&constrVars(SL, Phi) else false fi .
```

In order this rule to apply, the given clause should contain more than one literal. Thus, it is the first condition checked by the function `aritySL`. The function `depthSL` checks whether all literals in the given clause are of depth 0. If these two conditions are satisfied, then the function `const&constrVars` verifies whether the given clause contains only equalities and disequalities between constants and/or constrained variables. In order to check whether the variables are constrained, the additional parameter `Phi` is added to this function.

The second case of the *Schematic Deletion* rule for $\mathcal{SPC}$

```
crl [sd2] : Tr1 gives D' || Phi',
  Tr gives clause((D, L, SL)) || Phi =>
  Tr1 gives D' || Phi'
 if sd2Cond(D' || Phi', D, Phi, SL, L) .
```

requires a more sophisticated condition. It removes clauses that are composed of some literal `L`, some sets of literals `D` and `SL` (that can be empty), if

- `L` is a non-maximal literal in `D` $\vee$ `L`,
- a constrained clause composed of `D` and `L` is an elementary instance of some existing constrained clause `D' || Phi'`, and
- each literal in `SL` is an elementary instance of `L`.

```
op sd2Cond : SClause SetLit Constraint SetLit Literal -> Bool .
eq sd2Cond(D' || Phi', D, Phi, SL, L) =
  if isMax(L, Phi, D) == false and
     isElementaryInstance(D' || Phi', ((D, L)), Phi) and
     isElementaryInstanceSet(L, SL, Phi) then
    true
  else false fi .
```

All these required conditions are checked by the following functions. The maximality property is checked by the Boolean function `isMax`. It uses the function implementing lexicographic path ordering discussed in Section 6.6. To check whether the constrained clause composed of `D` and `L` with appropriate constraint is an elementary instance of `D' || Phi'`, we use the discussed above function `isElementaryInstance`. And finally, the Boolean function `isElementaryInstanceSet` returns `true` if all the literals in `SL` are elementary instances of `L`. This function also uses the function `isElementaryInstance`.

Schematic Deletion for $\mathcal{SUPC}_I$

The *Schematic Deletion* rule for $\mathcal{SUPC}_I$ is encoded by the following Maude conditional rewrite rule

```
crl [sdel] : L || Phi1, L' || Phi2 => L' || Phi2
  if sdelCond(L || Phi1, L' || Phi2) .
```

It leaves the second constrained clause if the following three conditions checked by the `sdelCond` function are satisfied: a) one of the terms of the literal `L` contains an `s` function symbol, b) one of the terms of the literal `L'` contains an $\mathtt{s}^+$ function symbol, c) after replacing `s` with $\mathtt{s}^+$ in `L` and normalizing the result by function `nfLit` described below, literals `L` and `L'` are renamings of each other.

The function `nfLit` (`nfLit : Literal -> Literal`) normalizes both sides of a given literal. The normalization of a term is encoded by the function `nf` (`op nf : Term -> Term`) that computes a normal form by applying the rules from the convergent rewrite system $R\mathtt{s}^+$ presented in Section 5.1.

Reduction rules for $\mathcal{SUPC}_I$

The *R1* reduction rule is encoded by the following conditional rewrite rule:

```
crl [red1] :
 'succ[U] equals 'succ[U'] || Phi => U equals U' || Phi
   if isGround(U, Phi) and isGround(U', Phi) .
```

This rule removes the root symbol in both sides of a literal if this root symbol is `'succ` (`'succ` stands for `s`) and their subterms `U` and `U'` are ground terms w.r.t. constraint, i.e. all the variables of `U` and `U'` are constrained. This condition is checked by the function `isGround` defined by

```
op isGround : Term Constraint -> Bool .
eq isGround(T, Phi) = vars(T) inTL varsOfSC(Phi) .
```

The function checks whether all variables of term `T` are in the list of variables of constraint `Phi`. This inclusion is checked by the `inTL` function.

The following Maude conditional rewrite rule encodes the *R2* reduction rule.

```
crl [red2] :
 'succ[U] equals T || Phi1, 'succ[V] equals T || Phi2 =>
 'succ[V] equals T || Phi2,
 U equals V || cleanConstraint(U, V, Phi1, Phi2)
   if isGround(U, Phi1) and isGround(V, Phi2) and
      isGround(T, Phi1) and gtLPO('succ[U], Phi1, T) and
      gtLPO('succ[V], Phi2, T) and gtLPO(U, (Phi1, Phi2), V) .
```

The ordering $>$ on terms is extended with a rule saying that compound terms are greater than constrained variables. That is why it is implemented as a Boolean function `gtLPO` such that $\texttt{gtLPO}(u, \texttt{Phi}, t) = \texttt{true}$ iff $u > t$ with an additional parameter `Phi` that collects the constrained variables. The function `cleanConstraint` aims at removing the constrained variables that do no occur in $u = v$.

6.4.2 States for rule application control

Since in particular contraction rules should be given a higher priority than expansion ones, the order of rule applications has to be controlled. An expected solution could be to control rule applications with the strategy language described in [MOMV05, EMOMV07], but unfortunately it appeared not to be compatible with the Full Maude version 2.5b required for narrowing (see details in Sect. 6.4.3). To circumvent this technical problem we propose to control rules with states.

In order to detect redundant clauses generated by expansion rules, we consider two distinct states defined as follows:

```
sort State .
op state : SetTracedSClause -> State .
op _redundancy_ : SetTracedSClause TracedSClause -> State .
```

The input state of the expansion rules of $\mathcal{SPC}$ is expected to be of the form `state(`S`)`, where S is a set of traced constrained clauses. A state of the form `_redundancy_` is entered after each application of an expansion rule. The state S `redundancy` C is the input state for checking whether the constrained clause C is redundant with respect to the set of constrained clauses S. If this is not the case, the clause is added to the set and this leads to a new state of the form `state(`$S \cup \{C\}$`)`. Otherwise, the next state is `state(`S`)`.

6.4.3 Superposition rule

The *Superposition* rule produces a new clause of the form $\sigma((C \vee D \vee l[t] \bowtie r) \parallel \varphi \wedge \psi)$ from any set containing two constrained clauses of the form $(C \vee l[u'] \bowtie r) \| \varphi$ and $(D \vee u = t) \| \psi$, if the side conditions given in Fig. 4.1 are satisfied with the most general unifier σ of u and u'. This notion of superposition is close to the notion of narrowing. The idea is to use

the literal $u = t$ from the second clause as a rewriting rule $u \to t$ to narrow the left-hand side term $l[u']$ of some literal in the first clause. If the narrowing succeeds it produces the term $\sigma(l[t])$, where σ is the most general unifier of u and u'. It remains to apply σ to the right-hand side term r of the literal in the first clause, to the clauses C and D, and to the conjunction of the two constraints φ and ψ.

To narrow we have developed a function `narrow` that uses a function `metaENarrowShowAll` implemented in Full Maude. In the standard version the narrowing is restricted to non-variable positions, along its standard definition. But the *Superposition* rule of $\mathcal{SPC}$ requires an unusual feature: narrowing should also be applied at the positions of the variables schematizing constants. This is why we use a dedicated version of Full Maude provided by Santiago Escobar to implement this unusual feature.

A second difficulty is that narrowing supports only rewriting rules where the variables of the right-hand side term of the rule are included in the set of variables that are on the left. To overcome this problem, we propose to build a rule that would contain only the variables in u. The one we use is $u \to f(u)$, where f is a special functional symbol that is not used anywhere else. When rewriting a term $l[u']$ by this rule, we obtain a new term of the form $\sigma(l[f(u)])$, where σ is the most general unifier of u and u'. The last step is to replace $f(u)$ by t. Thus, we get the expected term $\sigma(l[t])$.

The function `narrow` is encoded by the following four equations. It takes two elements of sort `NCLit` and two constraints as parameters, and returns an element of sort `ResultContextSet`.

```
op narrow : NCLit Constraint NCLit Constraint -> ResultContextSet .

eq narrow(nclit(Lu', r, b1), empty, nclit(u, t, b2), empty) =
  metaENarrowShowAll(addEq(u, buildTerm(u)),
  Lu', 1, full BuiltIn-unify noStrategy E-normalize-terms) .

eq narrow(nclit(Lu', r, b1), empty, nclit(u, t, b2), Phi) =
  metaENarrowShowAll(addEq(u, buildTerm(u)),
  Lu', 1, full BuiltIn-unify noStrategy E-normalize-terms) .

eq narrow(nclit(Lu', r, b1), Phi, nclit(u, t, b2), empty) =
  metaENarrowShowAll(addEq(u, buildTerm(u)),
  Lu', 1, full BuiltIn-unify noStrategy E-normalize-terms alsoAtVarPosition) .

eq narrow(nclit(Lu', r, b1), Phi1, nclit(u, t, b2), Phi2) =
  metaENarrowShowAll(addEq(u, buildTerm(u)),
  Lu', 1, full BuiltIn-unify noStrategy E-normalize-terms alsoAtVarPosition) .
```

The new sort `NCLit` defines a non-commutative literal by the function `nclit` that takes as parameters two terms and a boolean, where `true` stands for equality and `false` stands for disequality. The function `addEq` returns a module including the rewriting rule $u \to f(u)$, where the function `buildTerm` adds the special function symbol f to the term u. In the first two equations the standard version of the function `metaENarrowShowAll` is used.

The first equation presents a narrowing between two literals that are axioms, the second one presents a narrowing between an axiom and a literal from a constrained clause. The next two equations present narrowing between a literal from a constrained clause and a literal that is an axiom, and between two literals from constrained clauses. In both cases, the narrowing can be performed at the position of variables schematizing constants, that is why a new flag `alsoAtVarPosition` developed by Santiago Escobar is added to the function `metaENarrowShowAll`.

A third difficulty is that the `metaENarrowShowAll` function applied to the term $l(u')$ and the rule $u \to t$ generates all the possible narrowings at all the positions, whereas one application of the *Superposition* rule should produce only one clause. In order to consider each candidate clause one by one (among the set of results obtained by narrowing) we introduce an additional state `add_withLitFrom_to_` defined by

```
op add_withLitFrom_to_ : TracedSClause SetLSC SetTracedSClause
                         -> State .
```

where the sort `SetLSC` corresponds to the set of results computed by narrowing. The state "`add` C `withLitFrom` N `to` S" stores a part C of the new clause under construction, the set N of narrowing results and the current set of clauses S. Its use is detailed below.

A last difficulty is related to the substitution σ. In fact, narrowing renames all the variables in u by fresh variables and does not return this substitution. Let us call it σ_r. When applying narrowing, the actual rule becomes $\sigma_r(u) \to \sigma_r(f(u))$. To get σ_r, we use a term of the form $\sigma_r(l[f(u)])$ obtained by narrowing. Due to the fact that the functional symbol f is unique and there is only one subterm containing f, the substitution σ_r can be obtained by matching $f(\sigma_r(u))$ with $f(u)$. The narrowing output *ResultContext* contains two substitutions s_1 and s_2 that we use in order to build narrowing substitution σ_n, i.e. $\sigma_n = s_1; \mathit{cleaned}(s_2)$, where function *cleaned* removes the substitutions with left-hand side variables not appearing in the considered terms. Finally, the needed substitution σ is a composition of σ_r and σ_n. We use the Maude function `_;_` to build σ.

In order to better understand how the `Superposition` rule works, let us draw a graph (Figure 6.1) showing the intermediate states and guarded transitions. In this figure, the superposition rule applies between two clauses. We start from the state containing an initial set S of clauses. This set is decomposed into $T \cup \{L_1 \vee C_1, L_2 \vee C_2\}$, where L_1 and L_2 are two literals to which superposition will be applied, C_1 and C_2 are two remaining clauses, and T is a set of clauses. After applying narrowing between L_1 and L_2 (transition `[sup2]`), a new state `add` $C_1 \vee C_2$ `withLitFrom` U `to` S is constructed, where $C_1 \vee C_2$ is a disjunction of the remaining clauses, U is a set of all the possible narrowings of L_1 by L_2 at all the positions, and S is the initial set of clauses. If U is empty then by transition `[no-sup]` we go back to the input state. Otherwise, from this state the transition `[select]` will select some literal L in the set U of narrowing results decomposed into $\{L\} \cup R$, where R is the set of other narrowing results. If the constraint of L obtained from narrowing is satisfiable, then it builds (transition `[select]` (`then` part)) a new state S `redundancy` N, where N is the new clause $C_1 \vee C_2 \vee L$. Otherwise, another narrowing result is considered (transition `[select]` (`else` part)). If the new clause is redundant with respect to S, i.e. S can be obtained from $S \cup \{C\}$ by a sequence of applications of contraction rules ($S \cup \{C\} \to^* S$), then the state S remains

unchanged (transition `[pick]` (`else` part)). Otherwise, the new clause N is added to the state (transition `[pick]` (`then` part)).

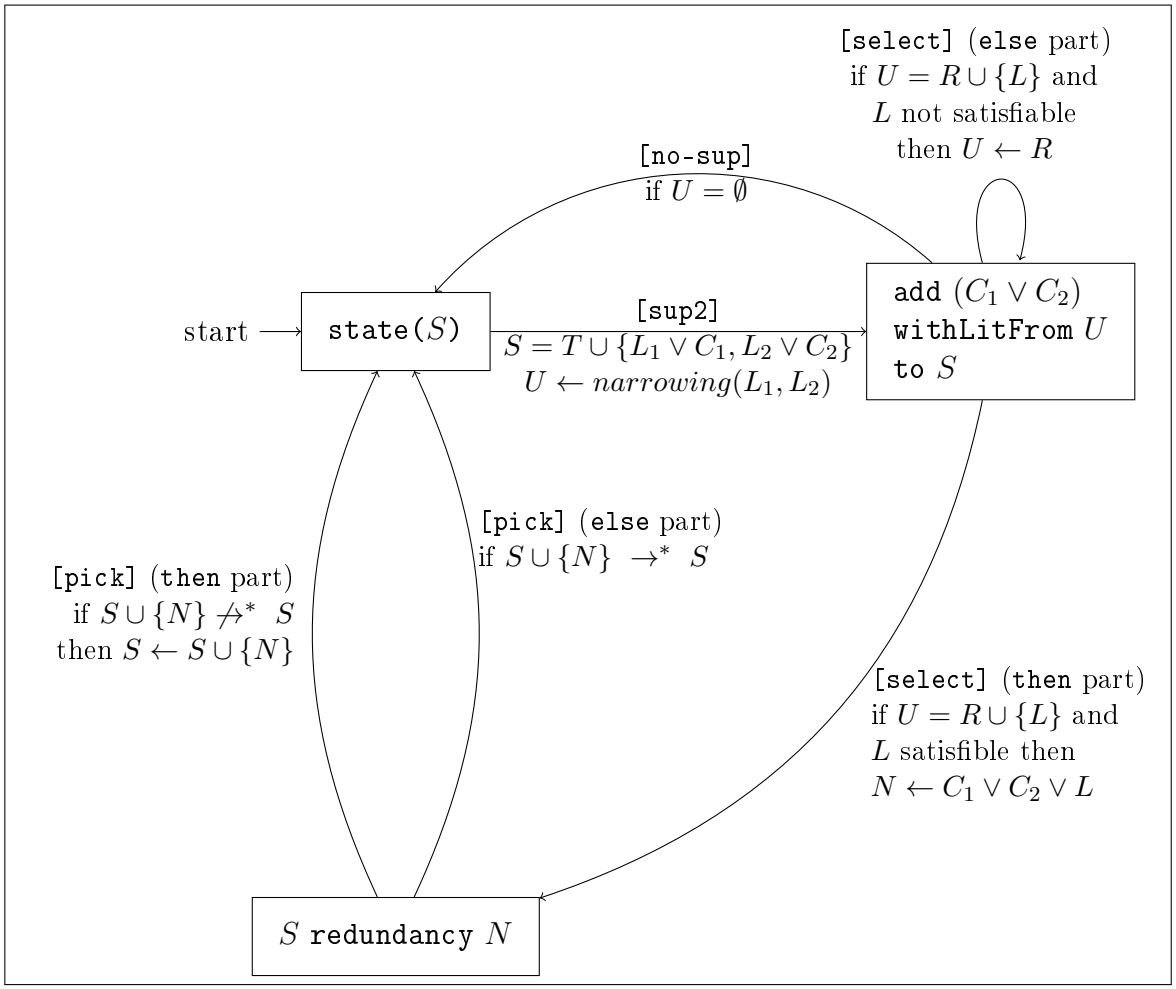

Figure 6.1: Intermediate states and transitions

Let us now implement these transitions by Maude rewriting rules. So, the `Superposition` rule is encoded by five rules named `sup1`, `sup2`, `select`, `no-sup` and `pick`, where `sup1` applies `Superposition` between a clause and itself. These rules are encoded as follows.

- Implementation of `[sup2]` and `[sup1]` rules:

  The `sup2` rule applies *Superposition* between two distinct traced constrained clauses.

```
var renamedSC2 : SClause .

crl [sup2] : state((STSC, Tr1 gives SC1, Tr2 gives SC2)) =>
  add
    sup(Tr1, Tr2, SubstR1, SubstR2) gives clause((SL1, SL2)) || (Phi1,Phi2)
  withLitFrom
```

```
    applySup(L1, SL1, Phi1, L2, SL2, Phi2)
  to (STSC, Tr1 gives SC1, Tr2 gives SC2)
  if renamedSC2 := renameSClause(SC1, SC2) /\
     SubstR1 := none /\
     SubstR2 := getSubstRename(renamedSC2, C2) /\
     clause((SL1, L1)) || Phi1 := SC1 /\
     clause((SL2, L2)) || Phi2 := renamedC2 /\
     isSelected(L1, SC1) /\ isSelected(L2, renamedSC2) .
```

The first condition `renamedC2 := ...` renames the second clause in order to guarantee that its variables are distinct from the ones of the first clause `SC1`. The substitutions `SubstR1` and `SubstR2` represent the renaming substitutions. The first substitution is empty (`SubstR1 := none`), because the renaming of the first clause has not been performed. The function `getSubstRename` returns a renaming substitution by filtering the given clause with the renamed one. The fourth (resp. fifth) condition decomposes the constrained clause `SC1` (resp. `renamedSC2`) and distinguishes a literal `L1` (resp. `L2`) in it. The *Superposition* rule applies only to literals that are selected in their clauses. The last two conditions check whether the literals `L1` and `L2` are selected in their clause. The function `applySup` generates a set of results (composed of two terms, a literal, a substitution, a constraint and a context) by calling the narrowing function and checking the ordering conditions of the *Superposition* rule. The ordering conditions invoke a function implementing the ordering detailed in Section 6.6.

The `sup1` rule superposes a traced constrained clause with a renamed copy of itself. It is very similar to `sup2`.

```
var renamedSC : SClause .

crl [sup1] : state((STSC, Tr gives SC)) =>
  add
    sup(Tr, Tr, SubstR1, SubstR2) gives clause((SL1, SL2)) || (Phi1, Phi2)
  withLitFrom
    applySup(L1, SL1, Phi1, L2, SL2, Phi2)
  to (STSC, Tr gives SC)
if renamedSC := renameSClause(SC) /\
   renamedSC2 := renameSClause(renamedSC) /\
   SubstR1 := getSubstRename(renamedSC, C1) /\
   SubstR2 := getSubstRename(renamedSC2, C1) /\
   clause((SL1, L1)) || Phi1 := renamedSC /\
   clause((SL2, L2)) || Phi2 := renamedSC2 /\
   isSelected(L1, renamedSC) /\
   isSelected(L2, renamedSC2) .
```

In order to get two constrained clauses with distinct variables, we rename the given constrained clause `SC` twice.

- Implementation of `[select]` rule:

  When the set of checked narrowing results is not empty, the following rule `[select]` considers these results one by one until either the set is empty, or a clause with a satisfiable constraint is found. Each narrowing result is of the form `superpose T in T' gives L withSubst Sigma andConstraint Phi at Ctx` of sort `LSC` and means that superposing a term `T` in a term `T'` gives a literal `L`. The superposition is performed at the context `Ctx` with substitution `Sigma`. The sort also stores a constraint of the clause `Phi`. The set of such results is of the sort `SetLSC`.

```
var S' : SetLSC .
rl [select] :
  add
    sup(Tr1,Tr2, SubstR1, SubstR2) gives clause(SL) || Phi'
  withLitFrom (superpose T1 in T2 gives L3
    withSubst Sigma andConstraint Phi at Ctx, S')
  to
    STSC =>
  if isSatisfiableSet(Phi applyToConstraint Sigma) then
     STSC redundancy
     buildTrace(sup(Tr1, Tr2), T1 << SubstR2, T2 << SubstR1, Ctx)
       gives newSClause((SL, L3), Phi, Sigma)
  else
    add sup(Tr1,Tr2) gives clause(SL) || Phi'
    withLitFrom S'
    to STSC
  fi .
```

  Satisfiability is checked by the function `isSatisfiableSet`. If the constraint of a new clause is satisfiable then a state `_redundancy_` is constructed. The first element of this state is the set `STSC` of traced constrained clauses generated so far. Its second element is a new traced constrained clause. The function `buildTrace` adds to the stored constrained clauses, between which superposition is performed, two terms substituted with the renaming substitutions and a context. The function `newSClause` applies the substitution `Sigma` to a set of literals and to the constraint `Phi`, and leaves only the atomic constraints built over variables occurring in the new clause. It returns the resulting constrained clause.

- Implementation of `[no-sup]` rule:

  When the set of results of narrowing is empty, the rule

```
rl [no-sup] : add TSC withLitFrom (empty).SetLSC to STSC
              => state(STSC) .
```

  returns the input set in a state ready for another expansion.

- Implementation of `[pick]` rule:

  Eventually, the rule

```
rl [pick] : STSC redundancy Tr gives SC =>
            if (Tr gives SC) isRedundant STSC == false then
              state((STSC, Tr gives SC))
            else state(STSC) fi .
```

checks whether the newly generated traced constrained clause `Tr gives SC` is redundant with respect to the set `STSC` of traced constrained clauses. If the new traced constrained clause is not redundant then it is added to the state, otherwise, the state remains unchanged. The redundancy is checked by the function `isRedundant` that uses the Maude function `metaSearch`.

```
op error : -> [SetTracedSClause] .

op _isRedundant_ : TracedSClause SetTracedSClause -> Bool .
ceq (Tr gives SC) isRedundant STSC = true
  if (Tr gives SC) in STSC .
eq (Tr gives SC) isRedundant STSC =
  if downTerm(getTerm(
     metaSearch(
       upModule('CONTR, false),
       upTerm((STSC, (Tr gives SC))),
       upTerm(STSC),
       nil, '*, unbounded, 0)), error) =/= error
   then true
   else false
 fi .
```

This function tries to reach the set `STSC` from the union of `STSC` and {`Tr gives SC`} by applying contraction rules defined in the module `CONTR`. When the Maude function `downTerm` fails in moving down the meta-represented term given as its first argument, it returns its second argument, namely `error`, declared as a constant of sort `SetTracedSClause`.

6.4.4 Reflection and Eq. Factoring rules

The implementation of the *Reflection* rule is divided into two cases. Let us consider the case where the clause consists of only one disequality:

```
crl [reflection1] :
  state((STSC, Tr gives clause(U' != U) || Phi)) =>
    state((STSC, Tr gives clause(U' != U) || Phi,
      emptySClause gives emptySClause))
      if MU := unif(U' != U) /\
         isSatisf(MU, Phi) .
```

In this case the empty constrained clause `emptySClause` is added to the state (with itself as trace). The function `unif` unifies two terms by using the Maude function `metaUnify`.

```
op unif : Literal -> UnificationPair .
eq unif(U' != U) = metaUnify(upModule('ALL-SYMBOLS,false), U' =? U, Max, 0) .
```

Since it is convenient to reuse variable names from unifiers in new problems, for example in narrowing, the `metaUnify` function has its third argument, which is the largest number n appearing in a unificand metavariable of the form $\#n : Sort$. The fresh metavariables in the computed unifiers will all be numbered from $n + 1$ on. We use as third argument the constant `Max` which is equal to 10. If the variables given to unification problem are greater than `Max`, for example, one of the variables is `#11`, then the Maude will display the following warning

```
Warning: unsafe variable name #11:Lists in unification problem.
```

that will make user understand that `Max` is too small for the unification problem. The last argument in this function (0 in our case) means that the first result is wanted.

The function `isSatisf` defined by

```
op isSatisf : UnificationPair Constraint -> Bool .
eq isSatisf(MU, Phi) =
  if MU =/= noUnifier then
    isSatisfiableSet(cleanConstr(Phi applyToConstraint getSubst(MU)))
    else false fi .
```

checks whether the substitution obtained from the unification pair satisfies the constraint.

In the general case, when the clause is not a unit one, the rule is encoded by using the same technique as the one developed for *Superposition*.

```
crl [reflection2] :
  state((STSC, Tr gives clause((S, U' != U)) || Phi)) =>
   (STSC, Tr gives clause((S, U' != U)) || Phi) redundancy
   refl(Tr) gives applicSubstToSClause(clause(S) || Phi, getSubst(MU))
    if S =/= empty /\
       MU := unif(U' != U) /\
       isSatisf(MU, Phi) /\
       isSelected(U != U', clause((U != U', S)) || Phi) .
```

The first condition `S =/= empty` checks whether the clause is not unit. The second condition `MU := unif(U' != U)` unifies two terms `U` and `U'` and returns the unification pair containing the needed substitution. The third condition checks, by the function `isSatisf`, whether the substitution satisfies the constraint `Phi`, and the last condition determines whether the disequality `U != U'` is selected in its clause. The trace of the new traced constrained clause is `refl(Tr)`, where `Tr` is the trace of the considered clause. The new constrained clause is obtained by the function `applicSubstToSClause` that applies the substitution obtained from the unification to the given constrained clause without the considered disequality.

In the same way, the implementation of the inference rule *Eq. Factoring* makes use of the same constructs, such as the function `isSelected`, together with a specific side condition involving ordering expressions.

```
var NewPhi : Constraint .
crl [eqfactor] :
state((STSC, Tr gives clause((SL, U equals T, U' equals T')) || Phi)) =>
STSC redundancy
ef(Tr) gives
applicSubstToSClause(clause((SL, T != T', U equals T')) || Phi, Sigma)
if isSelected(U equals T, clause((SL, U equals T, U' equals T')) || Phi) /\
  MU := metaUnify(upModule('ALL-SYMBOLS,false), U' =? U, Max, 0) /\
   MU =/= noUnifier /\
   Sigma := getSubst(MU) /\
   NewPhi := cleanConstr(Phi applyToConstraint Sigma) /\
   isSatisfiableSet(NewPhi) /\
   condEF(U << Sigma equals T << Sigma,
          U' << Sigma equals T' << Sigma, NewPhi) .
```

Firstly, we check whether the literal `U equals T` is selected in its clause. Then, we unify `U` and `U'` by using the function `metaUnify`. If the unification is possible, we get the substitution by the function `getSubst` and apply it to constraint `Phi`. The new constrained `NewPhi` is a substituted constrained `Phi` where all the atomic constraints containing constants (e.g., `const('nil.Lists)`) are eliminated by the function `cleanConstr`. After having checked that the obtained constraint is satisfiable, we check whether the ordering conditions are also satisfied by the function `condEF`. As the *Eq. Factoring* rule is an expansion rule, it generates a new constrained clause and therefore redundancy of the new constrained clause with respect to the given set of traced schematic clauses has to be checked. Thus, this rule constructs a new state `_redundancy_`. The trace of the new constrained clause is of the form `ef(Tr)`, where `Tr` is the trace of the given constrained clause. The new constrained clause is the constrained clause `clause((SL, T != T', U equals T')) || Phi` substituted with obtained substitution `Sigma` by the function `applicSubstToSClause`.

6.5 Saturation

A forward search for generated sets of traced constrained clauses is performed by a function `searchState` defined by

```
op searchState : State Nat -> State .
eq searchState(St, N) = downTerm(getTerm(
    metaSearch(upModule('SP, false), upTerm(St),
      'state['S:SetTracedSClause], nil, '*, unbounded, N)),
    error) .
```

where `SP` is a module where all the expansion rules are defined. The function call `searchState(St,N)` tries to reach the `N`-th state from an initial state `St` by applying

the expansion rules. It performs a breadth-first exploration of the reachable state space by calling the Maude function `metaSearch` with the `'*` parameter. When the Maude function `downTerm` fails in moving down the meta-represented term given as its first argument, it returns its second argument, namely `error`, declared as a constant of sort `State` (`op error : -> [State] .`).

Then the principle of saturation is implemented by the function `saturate` defined by

```
op saturate : State -> State .
eq saturate(St) = if searchState(St, 1) == error1 then St else
     if searchState(St, 1) =/= St then
       saturate(searchState(St, 1))
     else St fi fi .
```

which implements a fixpoint algorithm in order to reach a state where the set of constrained clauses is saturated. If the initial state is already saturated, then the function returns it unchanged.

A saturated set of constrained clauses could alternatively be computed from an initial state by the Maude function `metaSearch` with a `'!` parameter (searching for a state that cannot be further rewritten), but the function `searchState` computing intermediary states is also interesting for debugging purposes.

6.6 Orderings

A fundamental feature of paramodulation calculi is the usage of a simplification ordering which is total on ground terms. We use a lexicographic path ordering (Definition 52) with a precedence on function symbols. This section describes the implementation of this ordering.

The LPO ordering is implemented as a Boolean function `gtLPO` such that `gtLPO(s,Phi, t) = true` if and only if `s` $>_{lpo}$ `t`. The additional parameter `Phi` collects the constrained variables, that should be viewed as constants in the ordering definition.

Let us present the rules implementing `gtLPO(s,Phi,t)`.

1. Rule (1) is encoded by

   ```
   ceq gtLPO(F[NeSL], Phi, F[NeTL]) = true
     if gtLexLPO(NeSL, Phi, NeTL) /\ termGtList(F[NeSL], Phi, NeTL) .
   ```

   where `NeSL` and `NeTL` are non-empty lists of terms. Here the head symbols of both terms are equal. Then the list of subterms `NeSL` of `F[NeSL]` should be greater than the list of subterms `NeTL` and the term `F[NeSL]` should be greater than all the elements in the list of subterms of `F[NeTL]`. The first condition is checked by the function `gtLexLPO`

   ```
   op gtLexLPO : TermList Constraint TermList -> Bool .
   eq gtLexLPO(SL, Phi, empty) = true .
   ceq gtLexLPO((S1, SL), SC, (T1, TL)) = true
     if gtLPO(S1, Phi, T1) or (S1 == T1 and gtLexLPO(SL, Phi, TL)) .
   ```

that considers one by one the terms in the first list of terms and in the second one. The function checks whether either the first term `S1` from the first list of terms is greater than the first term `T1` from the second one, or they are equal and the rest `SL` of the first list of terms is greater than the rest `TL` of the second list of terms. The second condition is checked by the function `termGtList` that determines whether the term `F[NeSL]` is greater than each element in the list of terms `NeTL` by using the function `gtLPO`:

```
op termGtList : Term Constraint TermList -> Bool .
eq termGtList(T, Phi, empty) = true .
ceq termGtList(T, Phi, (T1, TL)) = true
  if gtLPO(T, Phi, T1) /\ termGtList(T, Phi, TL) .
```

2. Rule (2) is encoded by

```
ceq gtLPO(F[NeSL], Phi, G[NeTL]) = true
  if gtSymb(F, G) /\ termGtList(F[NeSL], Phi, NeTL) .
```

Here the heads of the compared terms are not equal. Then the head `F` of the first one should be greater than the head `G` of the second one and `F[NeSL]` should be greater than all the direct subterms of `G[NeTL]`. The following function `gtSymb` is used to check the first condition.

```
op gtSymb : Qid Qid -> Bool .
ceq gtSymb(Q1, Q2) = true
  if precedesInList(Q1, Q2, listOfSymbols) .
```

This function determines whether the first quoted identifier precedes the second one in the list of symbols `listOfSymbols` defined by the user.

3. Rule (3) is encoded by

```
ceq gtLPO(F[NeSL], Phi, V) = true
  if V in Phi /\
     C := cteOfSort(listOfSymbols, getType(V)) /\
     gtSymb(F, C) .
```

where the function `cteOfSort` returns the constant from the defined by user `listOfSymbols` of the given sort. This function is encoded by

```
op cteOfSort : QidList Qid -> Qid [memo] .
ceq cteOfSort((Q QL), S) = Q
  if getType(Q) == S .
eq cteOfSort((Q QL), S) = cteOfSort(QL, S) [owise] .
```

The constant `Q` is returned if its sort is the given sort `S`. Otherwise, we continue to search for the right constant. Thanks to the attribute `memo`, the constants associated with each sort are stored in some memoization table. This allows us to avoid the search for the right constant each time when this function is called, but to use the stored one.

4. Rule (4) is encoded by

```
ceq gtLPO(F[NeSL], Phi, C) = true
  if gtSymb(F, C) .
```

where it is checked whether the function symbol `F` precedes the constant `C` in the precedence of symbols.

5. Rule (5) is encoded by

```
ceq gtLPO(V, Phi, G[NeTL]) = true
  if V in Phi /\
     C := cteOfSort(listOfSymbols, getType(V)) /\
     gtSymb(C, G) /\
     termGtList(V, Phi, NeTL) .
```

where the function `_in_` checks whether the variable `V` is constrained, i.e. this variable is in the constraint `Phi`. After finding the constant `C` that has the same sort as the given constrained variable `V`, we determine whether it precedes the function symbol `G` in the precedence. The last condition checks whether the constrained variable is greater than each subterm of the term `G[NeTL]`.

6. Rule (6) is encoded by

```
ceq gtLPO(C, Phi, G[NeTL]) = true
  if gtSymb(C, G) and termGtList(C, Phi, NeTL) .
```

This rule is similar to the previous one, except that now the constant is compared with a compound term.

7. The rules (7), (8), (9) and (10) are similar in the sense that all these rules compare terms of depth 0. Rule (7) compares two constrained variables of different sorts by comparing the appropriate constants.

```
ceq gtLPO(V1, Phi, V2) = true
  if V1 in Phi /\ V2 in Phi /\
     getType(V1) =/= getType(V2) /\
     C1 := cteOfSort(listOfSymbols, getType(V1)) /\
     C2 := cteOfSort(listOfSymbols, getType(V2)) /\
     gtSymb(C1, C2) .
```

Rule (8) compares two constants by looking if the first one precedes the second one in the precedence defined by the user.

```
ceq gtLPO(C1, Phi, C2) = true
  if gtSymb(C1, C2) .
```

Rule (9) compares a constant and a constrained variable of distinct sorts by comparing the given constant with the constant representing the given constrained variable.

```
ceq gtLPO(C, Phi, V) = true
  if V in Phi /\
     getType(C) =/= getType(V) /\
     C1 := cteOfSort(listOfSymbols, getType(V)) /\
     gtSymb(C, C1) .
```

Rule (10) compares a constrained variable with a constant:

```
ceq gtLPO(V, Phi, C) = true
  if V in Phi /\
     getType(C) =/= getType(V) /\
     C1 := cteOfSort(listOfSymbols, getType(V)) /\
     gtSymb(C1, C) .
```

All these rules use the same functions: the function `_in_` to check whether the variable is constrained, the function `cteOfSort` to find a constant of the given sort in the list of symbols, and the function `gtSymb` to check whether one quoted identifier precedes another one in the precedence defined by the user.

8. Rule (11) is encoded by

   ```
   ceq gtLPO(F[UL1, Uk, UL2], Phi, t) = true
     if gtLPO(Uk, Phi, t) == true .
   ```

   whose condition checks whether a direct subterm of `F[UL1, Uk, UL2]` is greater than `t`.

9. Rule

   ```
   eq gtLPO(F[UL1, Uk, UL2], Phi, Uk) = true .
   ```

   encodes (12), when a compound term `F[UL1, Uk, UL2]` is compared with one of its direct subterms `Uk`.

To compare literals we use the ordering $>_{lpo}$ on terms extended to literals thanks to the multiset extension of $>_{lpo}$. An equality $l = r$ is represented as a multiset $[l, r]$ while a disequality $l \neq r$ is represented as a multiset $[l, l, r, r]$. The multiset extension of $>_{lpo}$ specified as an inference system is similarly encoded in Maude. The module `BAG` declares sorts and operators for manipulating bags of terms.

```
fmod BAG is
  pr META-TERM .
  sorts Bag NeBag .
  subsorts Term < NeBag < Bag .
  op emptyBag : -> Bag .
  op _&_ : Bag Bag -> Bag [assoc comm id: emptyBag] .
  op _&_ : Bag NeBag -> NeBag [ditto] .
endfm
```

For example the equivalence relation on bags is encoded as follows:

```
  op _equivBag_ : Bag Bag -> Bool .
  eq emptyBag equivBag emptyBag = true .
  eq (B1 & T1) equivBag (B2 & T2) =
    B1 equivBag B2 and T1 == T2 .
```

where `T1`,`T2` are two terms, and `B1`, `B2` are two bags.

6.7 Automatic combinability

As explained in Section 4.6, checking the combinability of a theory reduces to checking the existence of a variable-active clause in the saturation of G_0 for this theory. For the set of maximal literals in a clause, the following function detects whether the variable X does not occur in t when this literal is of the form $X = t$:

```
  eq isVarActiveClause(empty, Phi) = false .
  eq isVarActiveClause((X equals T, SL), Phi) =
    if not (X inTL vars(T)) and not (X inTL varsOfSC(Phi)) then
      true
    else isVarActiveClause(SL, Phi) fi .
  eq isVarActiveClause((L, SL), Phi) =
    isVarActiveClause(SL, Phi) [owise] .
```

For the theories sharing function symbols of theory of Integer Offsets, the combinability depends on the safety of saturations. As explained in Section 5.2, the saturation is safe if it does not contains variable-active clauses and non-ground clauses of the form $\mathsf{s}(u) = v$, where $\mathsf{s}(u)$ is maximal. The following function checks the latter condition:

```
  op isGroundLitWithSClause : SetLit Constraint -> Bool .
  eq isGroundLitWithSClause(empty, Phi) = true .
  eq isGroundLitWithSClause((L, SL), Phi) =
    isGroundLitWithS(L,Phi) and isGroundLitWithSClause(SL, Phi) .
```

where the function `isGroundLitWithS` checks whether the literal is a ground equality and one of its term is maximal and `s`-rooted.

Then the function checking whether the saturation is safe or not is encoded as follows:

```
  op isSafe : SetTracedSClause -> Bool .
  eq isSafe(STSC) =
    isGroundLitWithSSet(STSC) and not isVarActiveSet(STSC) .
```

6.8 Summary

This chapter has reported on a prototyping environment for designing and verifying decision procedures. This environment, based on the theoretical studies in [LM02, Tra07, LRRT11], is the first implementation of the schematic paramodulation calculus for theories of classical data-types such as lists, arrays, records, and for theories with counting operators. It has been implemented from scratch on the firm basis provided by Maude. To implement the schematic paramodulation calculus modulo Integer Offsets, we use the same rules as for the standard schematic paramodulation calculus, except the *Schematic Deletion* rule, and the reduction rules representing the axioms of the theory of Integer Offsets. Our tool will help testing new saturation strategies and experimenting new extensions of the original schematic paramodulation calculus. In the next section we present our experimental results obtained thanks to this tool.

Chapter 7

Experimentation

Contents

This chapter reports on experiments to compare the schematic saturations computed by our tool with corresponding results found in the literature. We consider the following theories:

- Unitary theories such as the theory of lists with and without extensionality and the theory of records without extensionality,
- Unitary theories that share Integer Offsets such as the theory of lists with length, the theory of lists with integer elements and the theory of records with increment,
- Non-unitary theories such as the theory of possibly empty lists, the theory of arrays and the theory of recursive data structures.

For all considered theories the paramodulation is known to terminate.

7.1 Theory of lists without extensionality

The many-sorted signature Σ_L of the theory of lists is the set of function symbols $\{\mathsf{car} : \text{LISTS} \to \text{ELEM}, \mathsf{cdr} : \text{LISTS} \to \text{LISTS}, \mathsf{cons} : \text{ELEM} \times \text{LISTS} \to \text{LISTS}\}$.

The set G_0 consists of the empty clause $\bot$ and the following constrained clauses over the signature Σ_L:

1. Schematic literals for ELEM
   a) $\mathsf{car}(a) = e \parallel const(a, e)$
   b) $e_1 = e_2 \parallel const(e_1, e_2)$
   c) $e_1 \neq e_2 \parallel const(e_1, e_2)$
2. Schematic literals for LISTS
   a) $\mathsf{cons}(e, a) = b \parallel const(e, a, b)$
   b) $\mathsf{cdr}(a) = b \parallel const(a, b)$
   c) $a = b \parallel const(a, b)$
   d) $a \neq b \parallel const(a, b)$

where e, e_1, e_2 are constrained variables of sort ELEM, a and b are constrained variables of sort LISTS.

This theory is axiomatized by the following set of axioms:

$$\mathsf{car}(\mathsf{cons}(X, Y)) = X \tag{7.1}$$
$$\mathsf{cdr}(\mathsf{cons}(X, Y)) = Y \tag{7.2}$$

where X is a universally quantified variable of sort ELEM, and Y is a universally quantified variable of sort LISTS.

The LPO ordering $>$ over the symbols of the signature Σ_L respects the following requirement: $\mathsf{cons} > \mathsf{cdr} > \mathsf{car} > l > e$ for every constant l of sort LISTS, and every constant e of sort ELEM.

Lemma 2. *The set $G_0 \cup \{(7.1), (7.2)\}$ is saturated by $\mathcal{SPC}$.*

Proof. Eq. Factoring applies to a clause with at least two positive literals, and therefore does not apply to unitary clauses. No rule can be applied to the set of axioms $\{(7.1), (7.2)\}$, and therefore it is saturated. All the applications of the *Superposition* rule between two constrained clauses produce clauses that are redundant w.r.t. G_0. The application of the *Reflection* rule produces the empty clause, which is also redundant w.r.t. G_0. Thus, the set G_0 is saturated. It remains to show the same property for the union of G_0 and $\{(7.1), (7.2)\}$.

Superposition between (7.1) and (2.a) yields a renaming of (1.a), which is immediately removed by the *Subsumption* rule. Similarly, *Superposition* between (7.2) and (2.a) yields a renaming of (2.b), which is removed by the *Subsumption* rule as well. *Superposition* between any axiom and (2.c) or (1.b) yields constrained clauses that are immediately removed by the *Subsumption* rule. Since no other rule can be applied between an axiom and a constrained clause, we conclude that the set $G_0 \cup \{(7.1), (7.2)\}$ is saturated. □

From an encoding of $G_0 \cup \{(7.1), (7.2)\}$ our tool generates no new constrained clauses. Notice that for this example the abstraction by schematization is exact, in the following sense: the saturated set computed by $\mathcal{SPC}$ is the schematization of any saturated set computed by $\mathcal{PC}$. The same result is shown in [LM02].

7.2 Theory of lists with extensionality

For this theory the signature, G_0 and the ordering requirement are the same as in 7.1. The set of axioms is extended with one extensionality axiom:

$$\mathsf{cons}(\mathsf{car}(X), \mathsf{cdr}(X)) \quad = \quad X \tag{7.3}$$

where X is a universally quantified variable of sort LISTS.

Lemma 3. *The saturation of* $G_0 \cup \{(7.1), (7.2), (7.3)\}$ *by* $\mathcal{SPC}$ *consists of* G_0, (7.1), (7.2), (7.3) *and the following constrained clauses:*

$$\mathsf{cons}(e, \mathsf{cdr}(a)) = b \quad \| \quad const(e, a, b) \tag{7.4}$$
$$\mathsf{cons}(\mathsf{car}(a), b) = c \quad \| \quad const(a, b, c) \tag{7.5}$$
$$\mathsf{car}(a) = \mathsf{car}(b) \quad \| \quad const(a, b) \tag{7.6}$$
$$\mathsf{cdr}(a) = \mathsf{cdr}(b) \quad \| \quad const(a, b) \tag{7.7}$$
$$\mathsf{cons}(\mathsf{car}(a), \mathsf{cdr}(b)) = c \quad \| \quad const(a, b, c) \tag{7.8}$$

Proof. Eq. Factoring applies to a clause with at least two positive literals, and therefore does not apply to unitary clauses. No rule is applicable to the set of axioms $\{(7.1), (7.2), (7.3)\}$ and, therefore, it is saturated.

All the applications of the *Superposition* rule between two constrained clauses in G_0 generate clauses that are redundant w.r.t. G_0. The application of the *Reflection* rule generates the empty clause which is already in G_0. Thus, the set G_0 is also saturated.

Let us now consider all the applications of the *Superposition* rule between an axiom and a constrained clause. *Superposition* between (7.1) and (2.a) and between (7.2) and (2.a) respectively yields renamings of (1.a) and (2.b), which are immediately removed by the *Subsumption* rule. *Superposition* between (7.3) and (1.a) yields the new constrained clause

$$\mathsf{cons}(e, \mathsf{cdr}(a)) = a \quad \| \quad const(e, a). \tag{7.9}$$

Then, *Superposition* between (7.9) and (2.c) gives the constrained clause (7.4), which subsumes (7.9). Similarly, *Superposition* between (7.3) and (2.b) yields the new constrained clause

$$\mathsf{cons}(\mathsf{car}(a), b) = a \quad \| \quad const(a, b) \tag{7.10}$$

and *Superposition* between (7.10) and (2.c) gives the constrained clause (7.5), which subsumes (7.10). *Superposition* between (7.1) and (7.5) and between (7.2) and (7.4) respectively gives the constrained clauses (7.6) and (7.7). *Superposition* between (7.3) and (7.6) gives the new constrained clause

$$\mathsf{cons}(\mathsf{car}(a), \mathsf{cdr}(b)) = b \quad \| \quad const(a, b) \tag{7.11}$$

and *Superposition* between (7.11) and (2.c) gives the constrained clause (7.8), which subsumes (7.11). *Superposition* between any axiom and (2.c) or (1.b) yields constrained clauses that are redundant w.r.t. the set $G_0 \cup \{(7.1), (7.2), (7.3), (7.4), (7.5), (7.6), (7.7), (7.8)\}$. Therefore, this set is saturated. □

The example given in [LRRT11] is not complete. In that paper, it is said that the saturation of $G_0 \cup \{(7.1), (7.2), (7.3)\}$ by $\mathcal{SPC}$, consists of the constrained clauses (7.4) and (7.5), while it also contains (7.6), (7.7) and (7.8). From an encoding of $G_0 \cup \{(7.1)$, (7.2), (7.3)$\}$ our tool generates five new constrained clauses corresponding to the ones given in Lemma 3:

$\mathbf{sup}($
  $\mathbf{sup}(label(7.3), label(1.a),\ \mathsf{car}(a),\ \mathsf{cons}(\mathsf{car}(X), \mathsf{cdr}(X)),\ \mathsf{cons}([], \mathsf{cdr}(X))),$
  $label(2.c),\ b,\ \mathsf{cons}(e, \mathsf{cdr}(a)),\ \mathsf{cons}(e, \mathsf{cdr}([])),)$ **gives**
    $clause(b = \mathsf{cons}(e, \mathsf{cdr}(a))) \parallel const(e, a, b)$
$\mathbf{sup}($
  $\mathbf{sup}(label(7.3), label(2.b),\ \mathsf{cdr}(a),\ \mathsf{cons}(\mathsf{car}(X), \mathsf{cdr}(X)),\ \mathsf{cons}(\mathsf{car}(X), [])),$
  $label(2.c),\ b,\ \mathsf{cons}(\mathsf{car}(a), b),\ \mathsf{cons}(\mathsf{car}([]), b))$ **gives**
    $clause(c = \mathsf{cons}(\mathsf{car}(a), b)) \parallel const(a, b, c)$
$\mathbf{sup}(label(7.1), label(7.5),\ \mathsf{cons}(\mathsf{car}(a), b),\ \mathsf{car}(\mathsf{cons}(X, Y)),\ \mathsf{car}([]))$ **gives**
    $clause(\mathsf{car}(a) = \mathsf{car}(b)) \parallel const(a, b)$
$\mathbf{sup}(label(7.2), label(7.4),\ \mathsf{cons}(e, \mathsf{cdr}(a)),\ \mathsf{cdr}(\mathsf{cons}(X, Y)),\ \mathsf{cdr}([]))$ **gives**
    $clause(\mathsf{cdr}(a) = \mathsf{cdr}(b)) \parallel const(a, b)$
$\mathbf{sup}($
  $\mathbf{sup}(label(7.3), label(7.6),\ \mathsf{car}(a),\ \mathsf{cons}(\mathsf{car}(X), \mathsf{cdr}(X)),\ \mathsf{cons}([], \mathsf{cdr}(X))),$
  $label(2.c),\ b,\ \mathsf{cons}(\mathsf{car}(b), \mathsf{cdr}(a)),\ \mathsf{cons}(\mathsf{car}(b), \mathsf{cdr}([])))$ **gives**
    $clause(c = \mathsf{cons}(\mathsf{car}(a), \mathsf{cdr}(b))) \parallel const(a, b, c)$

On this example we can see that the abstraction by schematization is an over-approximation: the abstract saturation computed by $\mathcal{SPC}$ is larger than any concrete saturation computed by $\mathcal{PC}$.

7.3 Theory of records without extensionality

A record can be considered as a special form of array where the number of elements is fixed. Contrary to the theory of arrays, the theory of records can be specified by unit clauses. The termination of superposition for the theories of records with and without extensionality is shown in [ABRS09]. We consider here the theory of records of length 3 without extensionality given by the many-sorted signature $\Sigma_{Rec} = \bigcup_{i=1}^{3}\{\mathsf{rstore}_i : \text{REC} \times \text{INT} \to \text{REC}, \mathsf{rselect}_i : \text{REC} \to \text{INT}\}$, and axiomatized by the following set of axioms $Ax(Rec)$:

$$\mathsf{rselect}_1(\mathsf{rstore}_1(X, Y)) = Y \tag{7.12}$$
$$\mathsf{rselect}_2(\mathsf{rstore}_2(X, Y)) = Y \tag{7.13}$$
$$\mathsf{rselect}_3(\mathsf{rstore}_3(X, Y)) = Y \tag{7.14}$$
$$\mathsf{rselect}_1(\mathsf{rstore}_2(X, Y)) = \mathsf{rselect}_1(X) \tag{7.15}$$
$$\mathsf{rselect}_1(\mathsf{rstore}_3(X, Y)) = \mathsf{rselect}_1(X) \tag{7.16}$$
$$\mathsf{rselect}_2(\mathsf{rstore}_1(X, Y)) = \mathsf{rselect}_2(X) \tag{7.17}$$
$$\mathsf{rselect}_2(\mathsf{rstore}_3(X, Y)) = \mathsf{rselect}_2(X) \tag{7.18}$$
$$\mathsf{rselect}_3(\mathsf{rstore}_1(X, Y)) = \mathsf{rselect}_3(X) \tag{7.19}$$
$$\mathsf{rselect}_3(\mathsf{rstore}_2(X, Y)) = \mathsf{rselect}_3(X) \tag{7.20}$$

where X is a universally quantified variable of sort REC, and Y is a universally quantified variable of sort INT.

Let G_0 be composed of the empty clause $\bot$ and the following constrained clauses:

1. Constrained clauses for sort REC
   a) $\mathsf{rstore}_1(a,i) = b \parallel const(a,b,i)$
   b) $\mathsf{rstore}_2(a,i) = b \parallel const(a,b,i)$
   c) $\mathsf{rstore}_3(a,i) = b \parallel const(a,b,i)$
   d) $a = b \parallel const(a,b)$
   e) $a \neq b \parallel const(a,b)$
2. Constrained clauses for sort INT
   a) $\mathsf{rselect}_1(a) = i \parallel const(a,i)$
   b) $\mathsf{rselect}_2(a) = i \parallel const(a,i)$
   c) $\mathsf{rselect}_3(a) = i \parallel const(a,i)$
   d) $i_1 = i_2 \parallel const(i_1,i_2)$
   e) $i_1 \neq i_2 \parallel const(i_1,i_2)$

where a, b are constrained variables of sort REC, and i, i_1, i_2 are constrained variables of sort INT.

The ordering over the terms is the LPO ordering $>$ whose underlying precedence over the symbols of the signature Σ_{Rec} respects the following requirements: $\mathsf{rstore}_i > \mathsf{rselect}_j > r > c$ for all $i, j \in \{1,2,3\}$, and for every constrained variable r of sort REC and every constrained variable c of sort INT.

Lemma 4. *The saturation of $G_0 \cup Ax(Rec)$ by $\mathcal{SPC}$ consists of G_0, $Ax(Rec)$ and the following constrained clauses:*

$$\mathsf{rselect}_1(a) = \mathsf{rselect}_1(b) \parallel const(a,b) \tag{7.21}$$
$$\mathsf{rselect}_2(a) = \mathsf{rselect}_2(b) \parallel const(a,b) \tag{7.22}$$
$$\mathsf{rselect}_3(a) = \mathsf{rselect}_3(b) \parallel const(a,b) \tag{7.23}$$

Proof. Eq. Factoring applies to a clause with at least two positive literals, and therefore does not apply to unitary clauses. No rules are applicable to the set of axioms $Ax(Rec)$, and therefore it is saturated.

All the applications of the *Superposition* rule between two constrained clauses produce clauses that are redundant w.r.t. G_0. The application of the *Reflection* rule produces the empty clause, which is also redundant w.r.t. G_0. Thus, the set G_0 is saturated.

Let us now consider all the applications of the *Superposition* rule between an axiom and a constrained clause. *Superposition* between (7.12) (resp. (7.13), (7.14)) and (1.a) (resp. (1.b), (1.c)) yields a renaming of (2.a) (resp. (2.b), (2.c)), which is immediately removed by the *Subsumption* rule. *Superposition* between (7.15) and (1.b) yields the constrained clause (7.21). Afterwards, *Superposition* between (7.16) and (1.c) yields a renaming of (7.21), which is immediately removed by the *Subsumption* rule. It is similar for the indices 2 and 3, between (7.17) and (1.a) and between (7.19) and (1.a). *Superposition* between any axiom and (1.d) or (2.d) yields constrained clauses that are immediately removed by the *Subsumption* rule. *Superposition* between new clauses (7.21), (7.22), (7.23) and (2.a), (2.b), (2.c) yields respectively renaming copies of (2.a), (2.b), (2.c) that are immediately removed by the *Subsumption* rule. Since no other rule applies we conclude that the set $G_0 \cup Ax(Rec) \cup \{(7.21), (7.22), (7.23)\}$ is saturated for $\mathcal{SPC}$. □

From an encoding of $G_0 \cup Ax(Rec)$ our tool generates the schematic saturation given in Lemma 4 which corresponds to the form of saturations described in [ABRS09]:

$\mathbf{sup}(label(7.15), label(1.b),\ \mathsf{rstore}_2(a,i),\ \mathsf{rselect}_1(\mathsf{rstore}_2(X,Y)),\ \mathsf{rselect}_1([]))$ **gives**
$\quad clause(\mathsf{rselect}_1(a) = \mathsf{rselect}_1(b)) \parallel const(a,b)$
$\mathbf{sup}(label(7.17), label(1.a),\ \mathsf{rstore}_1(a,i),\ \mathsf{rselect}_2(\mathsf{rstore}_1(X,Y)),\ \mathsf{rselect}_2([]))$ **gives**
$\quad clause(\mathsf{rselect}_2(a) = \mathsf{rselect}_2(b)) \parallel const(a,b)$
$\mathbf{sup}(label(7.19), label(1.a),\ \mathsf{rstore}_1(a,i),\ \mathsf{rselect}_3(\mathsf{rstore}_1(X,Y)),\ \mathsf{rselect}_3([]))$ **gives**
$\quad clause(\mathsf{rselect}_3(a) = \mathsf{rselect}_3(b)) \parallel const(a,b)$

7.4 Theory of lists with length

The many-sorted signature Σ_{LLI} of the theory of lists with length is the set $\{\mathsf{car} : \text{LISTS} \to \text{ELEM}, \mathsf{cdr} : \text{LISTS} \to \text{LISTS}, \mathsf{cons} : \text{ELEM} \times \text{LISTS} \to \text{LISTS}, \mathsf{len} : \text{LISTS} \to \text{INT}, \mathsf{nil} :\to \text{LISTS}, 0 :\to \text{INT}, \mathsf{s} : \text{INT} \to \text{INT}\}$. Let $\Sigma^+_{LLI} = \Sigma_{LLI} \cup \{\mathsf{s}^+ : \text{INT} \to \text{INT}\}$.

This theory is axiomatized by the following set of axioms $Ax(LLI)$:

1. Axioms for lists
   a) $\mathsf{car}(\mathsf{cons}(X,Y)) = X$
   b) $\mathsf{cdr}(\mathsf{cons}(X,Y)) = Y$
   c) $\mathsf{cons}(X,Y) \neq \mathsf{nil}$
2. Axiom for the length
   a) $\mathsf{len}(\mathsf{cons}(X,Y)) = \mathsf{s}(\mathsf{len}(Y))$
   b) $\mathsf{len}(\mathsf{nil}) = 0$

where X is a universally quantified variable of sort ELEM, and Y is a universally quantified variable of sort LISTS.

The set G_0 consists of the empty clause $\bot$ and the following schemas of literals:

3. Schematic literals of sort ELEM
   a) $\mathsf{car}(a) = e \parallel const(a,e)$
   b) $e_1 = e_2 \parallel const(e_1,e_2)$
   c) $e_1 \neq e_2 \parallel const(e_1,e_2)$
4. Schematic literals of sort LISTS
   a) $\mathsf{cons}(e,a) = b \parallel const(e,a,b)$
   b) $\mathsf{cdr}(a) = b \parallel const(a,b)$
   c) $a = b \parallel const(a,b)$
   d) $a \neq b \parallel const(a,b)$
5. Schematic literals of sort INT
   a) $\mathsf{len}(a) = \mathsf{s}^+(i) \parallel const(a,i)$
   b) $\mathsf{len}(a) = \mathsf{s}^+(\mathsf{len}(b)) \parallel const(a,b)$
   c) $i = \mathsf{s}^+(\mathsf{len}(a)) \parallel const(a,i)$
   d) $i_1 = \mathsf{s}^+(i_2) \parallel const(i_1,i_2)$
   e) $i = \mathsf{len}(a) \parallel const(a,i)$
   f) $i_1 = i_2 \parallel const(i_1,i_2)$
   g) $i_1 \neq i_2 \parallel const(i_1,i_2)$

where e, e_1, e_2 are constrained variables of sort ELEM, a,b are constrained variables of sort LISTS, and i, i_1, i_2 are constrained variables of sort INT.

We consider the LPO ordering on the terms whose underlying precedence over the symbols of the signature Σ^+_{LLI} respecting the following requirement: $\mathsf{cons} > \mathsf{cdr} > \mathsf{car} > c > e > \mathsf{len} > i > \mathsf{s} > \mathsf{s}^+$ for every constant c of sort LISTS, every constant e of sort ELEM, and every constant i of sort INT. These precedence requirements guarantee that

every compound term of sort LISTS or ELEM is bigger than any constant, and that $>$ is a $T_I - good$ ordering (Definition 45).

Lemma 5. *The saturation of $Ax(LLI) \cup G_0$ by $\mathcal{SUPC}_I$ consists of $Ax(LLI)$, G_0 and the following schematic literals:*

$$\mathsf{s}^+(i_1) = \mathsf{s}^+(i_2) \quad \| \quad const(i_1, i_2) \tag{7.24}$$
$$i_1 \neq \mathsf{s}^+(i_1) \quad \| \quad const(i_1, i_2) \tag{7.25}$$
$$\mathsf{s}^+(i_1) \neq \mathsf{s}^+(i_2) \quad \| \quad const(i_1, i_2) \tag{7.26}$$
$$\mathsf{s}^+(i_1) = \mathsf{s}^+(\mathsf{len}(a)) \quad \| \quad const(i_1, a) \tag{7.27}$$
$$\mathsf{len}(a) = \mathsf{len}(b) \quad \| \quad const(a, b) \tag{7.28}$$
$$\mathsf{s}^+(\mathsf{len}(a)) = \mathsf{s}^+(\mathsf{len}(b)) \quad \| \quad const(a, b) \tag{7.29}$$

Proof. The six new schematic literals are generated by applications of the *Superposition* rule between two schematic literals in the initial set G_0, as follows: *Superposition* between (5.d) and a renamed copy of itself yields the new schematic literal (7.24). *Superposition* between (5.g) and (5.d) yields the new schematic literal (7.25) that can be superposed with (5.d) to generate (7.26). *Superposition* between (5.a) and (5.b) yields the new schematic literal (7.27). *Superposition* between (5.b) and a renamed copy of itself yields two new schematic literals (7.28) and (7.29), and a literal $\mathsf{len}(a) = \mathsf{s}^+(\mathsf{s}^+(len(b)))$ that becomes a renaming copy of (5.b) after applying the rules from $R\mathsf{s}^+$ (defined in Section 5.1) and is eliminated thanks to *Subsumption* rule.

Let us now consider all the applications of the *Superposition* rule between an axiom in $Ax(LLI)$ and a schematic literal in G_0. *Superposition* between (1.a) (resp. (1.b)) and (4.a) yields a renaming of (3.a) (resp. (4.b)) which is immediately removed by the *Subsumption* rule. *Superposition* between (2.a) and (4.a) yields the new schematic literal

$$\mathsf{len}(a) = \mathsf{s}(\mathsf{len}(b)) \quad \| \quad const(a, b)$$

which is immediately removed by applying the *Schematic Deletion* rule between it and (5.b). *Superposition* between (1.c) and (4.a) yields an elementary instance of (4.d) that is immediately eliminated by the *Subsumption* rule.

The set of axioms $Ax(LLI)$ is saturated. Moreover, any other application of the *Superposition* rule between two schematic literals or between an axiom and a schematic literal yields a schematic literal that is redundant with respect to $G_0 \cup Ax(LLI) \cup \{(7.24), (7.25), (7.26), (7.27), (7.28), (7.29)\}$. Therefore, this set of schematic literals is saturated. □

From an encoding of $G_0 \cup Ax(LLI)$ our tool generates the schematic saturation given in Lemma 5. Moreover, its trace

$\mathbf{sup}(label(5.d), label(5.d),\ i_1,\ i_1,\ [])$ **gives** $\mathsf{s}^+(i_1) = \mathsf{s}^+(i_2) \parallel const(i_1, i_2)$
$\mathbf{sup}(label(5.g), label(5.d),\ i_1,\ i_1,\ [])$ **gives** $i_1 \neq \mathsf{s}^+(i_2) \parallel const(i_1, i_2)$
$\mathbf{sup}(label(7.25), label(5.d),\ i_1,\ i_1,\ [])$ **gives** $\mathsf{s}^+(i_1) \neq \mathsf{s}^+(i_2) \parallel const(i_1, i_2)$
$\mathbf{sup}(label(5.a), label(5.b),\ \mathsf{len}(a),\ \mathsf{len}(a),\ [])$ **gives** $\mathsf{s}^+(i_1) = \mathsf{s}^+(\mathsf{len}(a)) \parallel const(i_1, a)$
$\mathbf{sup}(label(5.b), label(5.b),\ \mathsf{s}^+(\mathsf{len}(b)),\ \mathsf{s}^+(\mathsf{len}(b)),\ [])$ **gives** $\mathsf{len}(a) = \mathsf{len}(b) \parallel const(a, b)$
$\mathbf{sup}(label(5.b), label(5.b),\ \mathsf{len}(a),\ \mathsf{len}(a),\ [])$ **gives** $\mathsf{s}^+(\mathsf{len}(a)) = \mathsf{s}^+(\mathsf{len}(b)) \parallel const(a, b)$

shows that the new schematic clauses are generated as described in the lemma proof.

7.5 Theory of lists with integer elements

The many-sorted signature Σ_{LIE} of the theory of lists with integer elements is the set $\{\mathsf{car} : \text{LISTS} \to \text{INT}, \mathsf{cdr} : \text{LISTS} \to \text{LISTS}, \mathsf{cons} : \text{INT} \times \text{LISTS} \to \text{LISTS}, \mathsf{len} : \text{LISTS} \to \text{INT}, \mathsf{nil} :\to \text{LISTS}, 0 :\to \text{INT}, \mathsf{s} : \text{INT} \to \text{INT}\}$. Let $\Sigma^{+}_{LIE} = \Sigma_{LIE} \cup \{\mathsf{s}^{+} : \text{INT} \to \text{INT}\}$.

This theory is axiomatized by the following set of axioms $Ax(LIE)$:

1. Axioms of sort LISTS
   a) $\mathsf{cdr}(\mathsf{cons}(X, Y)) = Y$
   b) $\mathsf{cons}(X, Y) \neq \mathsf{nil}$
2. Axiom of sort INT
   a) $\mathsf{car}(\mathsf{cons}(X, Y)) = X$
   b) $\mathsf{len}(\mathsf{cons}(X, Y)) = \mathsf{s}(\mathsf{len}(Y))$
   c) $\mathsf{len}(\mathsf{nil}) = 0$

where X is a universally quantified variable of sort INT, and Y is a universally quantified variable of sort LISTS.

The set G_0 consists of the empty clause $\bot$ and the following schemas of literals

3. Schematic literals of sort LISTS
   a) $\mathsf{cons}(i, a) = b \parallel const(i, a, b)$
   b) $\mathsf{cdr}(a) = b \parallel const(a, b)$
   c) $a = b \parallel const(a, b)$
   d) $a \neq b \parallel const(a, b)$
4. Schematic literals of sort INT
   a) $\mathsf{car}(a) = i \parallel const(a, i)$
   b) $\mathsf{len}(a) = i \parallel const(a, i)$
   c) $\mathsf{len}(a) = \mathsf{s}^{+}(i) \parallel const(a, i)$
   d) $i = \mathsf{s}^{+}(\mathsf{len}(a)) \parallel const(a, i)$
   e) $\mathsf{len}(a) = \mathsf{s}^{+}(\mathsf{len}(b)) \parallel const(a, b)$
   f) $\mathsf{car}(a) = \mathsf{s}^{+}(i) \parallel const(a, i)$
   g) $i = \mathsf{s}^{+}(\mathsf{car}(a)) \parallel const(a, i)$
   h) $\mathsf{car}(a) = \mathsf{s}^{+}(\mathsf{car}(b)) \parallel const(a, b)$
   i) $i_1 = \mathsf{s}^{+}(i_2) \parallel const(i_1, i_2)$
   j) $i_1 = i_2 \parallel const(i_1, i_2)$
   k) $i_1 \neq i_2 \parallel const(i_1, i_2)$

where a,b are constrained variables of sort LISTS, and i, i_1, i_2 are constrained variables of sort INT.

We choose an order $>$ over the symbols of the signature Σ^{+}_{LIE} that respects the following requirement: $\mathsf{cons} > \mathsf{cdr} > c > \mathsf{len} > \mathsf{car} > i > \mathsf{s} > \mathsf{s}^{+}$ for every constant c of sort LISTS, and every constant i of sort INT. This order $>$ is a $T_I - good$ order.

Lemma 6. *The saturation of $Ax(LIE) \cup G_0$ by $\mathcal{SUPC}_I$ consists of $Ax(LIE)$, G_0 and the following schematic literals:*

$$\mathsf{s}^+(i_1) = \mathsf{s}^+(i_2) \quad \| \quad const(i_1, i_2) \tag{7.30}$$
$$i_1 \neq \mathsf{s}^+(i_2) \quad \| \quad const(i_1, i_2) \tag{7.31}$$
$$\mathsf{s}^+(i_1) \neq \mathsf{s}^+(i_2) \quad \| \quad const(i_1, i_2) \tag{7.32}$$
$$a = \mathsf{cons}(\mathsf{s}^+(i), b) \quad \| \quad const(a, i, b) \tag{7.33}$$
$$\mathsf{s}^+(i) = \mathsf{s}^+(\mathsf{car}(a)) \quad \| \quad const(i, a) \tag{7.34}$$
$$\mathsf{car}(a) = \mathsf{car}(b) \quad \| \quad const(a, b) \tag{7.35}$$
$$\mathsf{s}^+(\mathsf{car}(a)) = \mathsf{s}^+(\mathsf{car}(b)) \quad \| \quad const(a, b) \tag{7.36}$$
$$\mathsf{s}^+(i) = \mathsf{s}^+(\mathsf{len}(a)) \quad \| \quad const(i, a) \tag{7.37}$$
$$\mathsf{len}(a) = \mathsf{len}(b) \quad \| \quad const(a, b) \tag{7.38}$$
$$\mathsf{s}^+(\mathsf{len}(a)) = \mathsf{s}^+(\mathsf{len}(b)) \quad \| \quad const(a, b) \tag{7.39}$$

Proof. The new schematic literals are generated by application of the *Superposition* rule between two schematic literals in the initial set G_0 as follows. *Superposition* between (4.i) and a renamed copy of itself yields the new schematic literal (7.30). *Superposition* between (4.k) and (4.i) yields the new schematic literal (7.31). *Superposition* between (7.32) and (4.i) yields the new schematic literal (7.32). *Superposition* between (3.a) and (4.i) yields the new schematic literal (7.33). *Superposition* between (4.f) and (4.h) yields the new schematic literal (7.34). *Superposition* between (4.h) and a renamed copy of itself yields two new schematic literals (7.35) and (7.36), and the schematic literal

$$\mathsf{car}(a) = \mathsf{s}^+(\mathsf{s}^+(\mathsf{car}(b))) \parallel const(a, b)$$

that becomes a renaming copy of (4.h) after application of the rules from $R\mathsf{s}^+$ (defined in Section 5.1), and is therefore eliminated by the *Subsumption* rule. *Superposition* between (4.c) and (4.e) yields the new schematic literal (7.37). *Superposition* between (4.e) and a renamed copy of itself yields two new schematic literal (7.38) and (7.39), and the schematic literal

$$\mathsf{len}(a) = \mathsf{s}^+(\mathsf{s}^+(\mathsf{len}(b))) \parallel const(a, b)$$

that becomes a renaming copy of (4.e) after application of the rules from $R\mathsf{s}^+$, and is therefore eliminated by the *Subsumption* rule.

Let us now consider all the applications of the *Superposition* rule between an axiom in $Ax(LIE)$ and a schematic literal in G_0. *Superposition* between (2.a) (resp. (1.a)) and (3.a) yields a renaming of (4.a) (resp. (3.b)) which is immediately removed by the *Subsumption* rule. *Superposition* between (2.b) and (3.a) yields the new schematic literal

$$\mathsf{len}(a) = \mathsf{s}(\mathsf{len}(b)) \quad \| \quad const(a, b)$$

which is immediately removed by applying the *Schematic Deletion* rule between it and (4.e). *Superposition* between (1.b) and (3.a) yields an elementary instance of (3.d) that is immediately eliminated by the *Subsumption* rule.

The set of axioms $Ax(LIE)$ is saturated. Moreover, any other application of the *Superposition* rule between two schematic literals or between an axiom and a schematic

literal yields a schematic literal that is redundant with respect to $G_0 \cup Ax(LIE) \cup \{(7.30), (7.31), (7.32), (7.33), (7.34), (7.35), (7.36), (7.37), (7.38), (7.39)\}$. Therefore, this set of schematic literals is saturated. □

From an encoding of $G_0 \cup Ax(LIE)$ our tool generates the schematic saturation given in Lemma 6. Moreover, the system traces show that the new schematic clauses are generated as described in the lemma proof.

$\mathbf{sup}(label(4.i), label(4.i),\ i_1,\ i_1,\ [])$ **gives** $\mathsf{s}^+(i_1) = \mathsf{s}^+(i_2) \parallel const(i_1, i_2)$
$\mathbf{sup}(label(4.k), label(4.i),\ i_1,\ i_1,\ [])$ **gives** $i_1 \neq \mathsf{s}^+(i_2) \parallel const(i_1, i_2)$
$\mathbf{sup}(label(7.31), label(4.i),\ i_1,\ i_1,\ [])$ **gives** $\mathsf{s}^+(i_1) \neq \mathsf{s}^+(i_2) \parallel const(i_1, i_2)$
$\mathbf{sup}(label(3.a), label(4.i),\ i_1,\ \mathsf{cons}(i, a),\ \mathsf{cons}([], a))$ **gives** $a = \mathsf{cons}(\mathsf{s}^+(i), b) \parallel const(a, i, b)$
$\mathbf{sup}(label(4.f), label(4.h),\ \mathsf{car}(a),\ \mathsf{car}(a),\ [])$ **gives** $\mathsf{s}^+(i) = \mathsf{s}^+(\mathsf{car}(a)) \parallel const(a, i)$
$\mathbf{sup}(label(4.h), label(4.h),\ \mathsf{s}^+(\mathsf{car}(b)),\ \mathsf{s}^+(\mathsf{car}(b)),\ [])$ **gives** $\mathsf{car}(a) = \mathsf{car}(b) \parallel const(a, b)$
$\mathbf{sup}(label(4.h), label(4.h),\ \mathsf{car}(a),\ \mathsf{car}(a),\ [])$ **gives** $\mathsf{s}^+(\mathsf{car}(a)) = \mathsf{s}^+(\mathsf{car}(b)) \parallel const(a, b)$
$\mathbf{sup}(label(4.c), label(4.e),\ \mathsf{len}(a),\ \mathsf{len}(a),\ [])$ **gives** $\mathsf{s}^+(i) = \mathsf{s}^+(\mathsf{len}(a)) \parallel const(a, i)$
$\mathbf{sup}(label(4.e), label(4.e),\ \mathsf{s}^+(\mathsf{len}(b)),\ \mathsf{s}^+(\mathsf{len}(b)),\ [])$ **gives** $\mathsf{len}(a) = \mathsf{len}(b) \parallel const(a, b)$
$\mathbf{sup}(label(4.e), label(4.e),\ \mathsf{len}(a),\ \mathsf{len}(a),\ [])$ **gives** $\mathsf{s}^+(\mathsf{len}(a)) = \mathsf{s}^+(\mathsf{len}(b)) \parallel const(a, b)$

7.6 Theory of records with increment

We consider the theory of records of length 3 (without extensionality) with increment defined by the many-sorted signature $\Sigma_{RII} = \bigcup_{i=1}^{3} \{\mathsf{rstore}_i : \text{REC} \times \text{INT} \to \text{REC}, \mathsf{rselect}_i : \text{REC} \to \text{INT}, \mathsf{incr} : \text{REC} \to \text{REC}, \mathsf{s} : \text{INT} \to \text{INT}\}$. Let $\Sigma^+_{RII} = \Sigma_{RII} \cup \{\mathsf{s}^+ : \text{INT} \to \text{INT}\}$.

This theory is axiomatized by the following set of axioms $Ax(RII)$:

1. Axioms for records
   a) $\mathsf{rselect}_i(\mathsf{rstore}_i(X, Y)) = Y$ for all $i \in \{1, 2, 3\}$
   b) $\mathsf{rselect}_j(\mathsf{rstore}_i(X, Y)) = \mathsf{rselect}_j(X, Y)$ for all $i, j \in \{1, 2, 3\},\ i \neq j$
2. Axiom for the increment
   a) $\mathsf{rselect}_i(\mathsf{incr}(X)) = \mathsf{s}(\mathsf{rselect}_i(X))$ for all $i \in \{1, 2, 3\}$

where X is a universally quantified variable of sort REC, and Y is a universally quantified variable of sort INT. The set G_0 consists of the empty clause $\bot$ and the following schemas of literals:

3. Schematic literals of sort REC
   a) $b = \mathsf{rstore}_i(a, e) \parallel const(a, b, e)$
   b) $b = \mathsf{incr}(a) \parallel const(a, b)$
   c) $a = b \parallel const(a, b)$
   d) $a \neq b \parallel const(a, b)$
4. Schematic literals of sort INT

a) $e = \mathsf{rselect}_i(a) \parallel const(a, e)$

b) $\mathsf{rselect}_i(a) = \mathsf{s}^+(e) \parallel const(a, e)$

c) $\mathsf{rselect}_i(a) = \mathsf{s}^+(\mathsf{rselect}_i(b)) \parallel const(a, b)$

d) $e = \mathsf{s}^+(\mathsf{rselect}_i(a)) \| const(a, e)$

e) $e_1 = \mathsf{s}^+(e_2) \parallel const(e_1, e_2)$

f) $e_1 = e_2 \parallel const(e_1, e_2)$

g) $e_1 \neq e_2 \parallel const(e_1, e_2)$

where a, b are constrained variables of sort REC, e, e_1, e_2 are constrained variables of sort INT, and $i \in \{1, 2, 3\}$.

We consider an LPO ordering $>$ whose underlying precedence over the symbols of the signature Σ^+_{RII} satisfies the following requirements: for all i, j in $\{1, .., n\}$ $\mathsf{incr} > \mathsf{rstore}_i$, $\mathsf{rstore}_i > \mathsf{rselect}_j$, $\mathsf{rselect}_i > c$ for every constant c, and every constant c is such that $c > \mathsf{s} > \mathsf{s}^+$.

Lemma 7. *The saturation of $G_0 \cup Ax(RII)$ by $\mathcal{SUPC}_I$ consists of G_0, $Ax(RII)$ and the following schematic literals, where $i \in \{1, 2, 3\}$:*

$$\mathsf{s}^+(e_1) = \mathsf{s}^+(e_2) \quad \| \quad const(e_1, e_2) \tag{7.40}$$

$$e_1 \neq \mathsf{s}^+(e_2) \quad \| \quad const(e_1, e_2) \tag{7.41}$$

$$\mathsf{s}^+(e_1) \neq \mathsf{s}^+(e_2) \quad \| \quad const(e_1, e_2) \tag{7.42}$$

$$\mathsf{rselect}_i(a) = \mathsf{rselect}_i(b) \quad \| \quad const(a, b) \tag{7.43}$$

$$\mathsf{s}^+(\mathsf{rselect}_i(a)) = \mathsf{s}^+(\mathsf{rselect}_i(b)) \quad \| \quad const(a, b) \tag{7.44}$$

$$\mathsf{s}^+(e_1) = \mathsf{s}^+(\mathsf{rselect}_i(a)) \quad \| \quad const(a, e_1) \tag{7.45}$$

$$\mathsf{rstore}_i(a, \mathsf{s}^+(e)) = b \quad \| \quad const(a, b, e) \tag{7.46}$$

Proof. The new schematic literals are generated by application of the *Superposition* rule between two schematic literals in the initial set G_0, as follows: *Superposition* between (4.e) and the renamed copy of itself yields the new schematic literal (7.40). *Superposition* between (4.g) and (4.e) yields the new schematic literal (7.41) that can be superposed with (4.e) to generate the new schematic literal (7.42). *Superposition* between (4.c) and its renamed copy yields two new schematic literals (7.43) and (7.44) for $i \in \{1, 2, 3\}$, and the schematic literals $\mathsf{rselect}_i(a) = \mathsf{s}^+(\mathsf{s}^+(\mathsf{rselect}_i(b))) \parallel const(a, b)$ for $i \in \{1, 2, 3\}$ that become renaming copies of (4.c) after application of the rules from Rs^+ (defined in Section 5.1), and are therefore removed. *Superposition* between (4.c) and (4.b) yields the new schematic literals (7.45) for $i \in \{1, 2, 3\}$. *Superposition* between (3.a) and (4.e) yields the new schematic literals (7.46) for $i \in \{1, 2, 3\}$.

Let us now consider all the applications of the *Superposition* rule between an axiom in $Ax(RII)$ and a schematic literal in G_0. For $i \in \{1, 2, 3\}$, *Superposition* between (1.a) and (3.a) yields a renaming of (4.a), which is immediately removed by the *Subsumption* rule. For $i, j \in \{1, 2, 3\}$ with $i \neq j$, *Superposition* between (1.b) and (3.a) yields a renaming of (7.43), which is immediately removed by the *Subsumption* rule.

The set of axioms $Ax(RII)$ is saturated. Moreover, any other application of *Superposition* rule between an axiom and a schematic literal, or between two schematic literals yields a schematic literal that is redundant with respect to $G_0 \cup Ax(RII) \cup \{(7.40), (7.41), (7.42), (7.43), (7.44), (7.45), (7.46)\}$. Therefore, this set of schematic literals is saturated. □

From an encoding of $G_0 \cup Ax(RII)$ our tool generates the schematic saturation given in Lemma 7 and provides the following trace in conformity with the proof of this lemma:

$\mathbf{sup}(label(4.e), label(4.e),\ e_1,\ e_1,\ [])$ **gives** $\mathsf{s}^+(e_1) = \mathsf{s}^+(e_2) \parallel const(e_1, e_2)$
$\mathbf{sup}(label(4.g), label(4.e),\ e_1,\ e_1,\ [])$ **gives** $e_1 \neq \mathsf{s}^+(e_2) \parallel const(e_1, e_2)$
$\mathbf{sup}(label(7.41), label((4.e),\ e_1,\ e_1,\ [])$ **gives** $\mathsf{s}^+(e_1) \neq \mathsf{s}^+(e_2) \parallel const(e_1, e_2)$
$\mathbf{sup}(label(4.c), label(4.c),\ \mathsf{s}^+(\mathsf{rselect}_i(b)),\ \mathsf{s}^+(\mathsf{rselect}_i(b)),\ [])$ **gives**
$\mathsf{rselect}_i(a) = \mathsf{rselect}_i(b) \parallel const(a, b)$
$\mathbf{sup}(label(4.c), label(4.c),\ \mathsf{rselect}_i(a),\ \mathsf{rselect}_i(a),\ [])$ **gives**
$\mathsf{s}^+(\mathsf{rselect}_i(a)) = \mathsf{s}^+(\mathsf{rselect}_i(b)) \parallel const(a, b)$
$\mathbf{sup}(label(4.c), label(4.b),\ \mathsf{rselect}_i(a),\ \mathsf{rselect}_i(a),\ [])$ **gives**
$\mathsf{s}^+(e) = \mathsf{s}^+(\mathsf{rselect}_i(a)) \parallel const(a, e)$
$\mathbf{sup}(label(3.a), label(4.e),\ e_1,\ \mathsf{rstore}_i(a, e),\ \mathsf{rstore}_i(a, []),)$ **gives**
$\mathsf{rstore}_i(a, \mathsf{s}^+(e)) = b \parallel const(a, b, e)$

7.7 Theory of possibly empty lists

The signature of the theory of possibly empty lists (PEL, for short, introduced in Sect. 4.4) is composed of the binary function symbol cons, the unary function symbols car and cdr, and the constant nil, denoting the empty list. This theory is axiomatized by the following set $Ax(PEL)$ of axioms:

$$\mathsf{car}(\mathsf{cons}(X, Y)) = X \tag{7.47}$$
$$\mathsf{cdr}(\mathsf{cons}(X, Y)) = Y \tag{7.48}$$
$$\mathsf{cons}(X, Y) \neq \mathsf{nil} \tag{7.49}$$
$$\mathsf{cons}(\mathsf{car}(Y), \mathsf{cdr}(Y)) = Y \vee Y = \mathsf{nil} \tag{7.50}$$
$$\mathsf{car}(\mathsf{nil}) = \mathsf{nil} \tag{7.51}$$
$$\mathsf{cdr}(\mathsf{nil}) = \mathsf{nil} \tag{7.52}$$

where X, Y are universally quantified variables.

The set G_0 consists of the empty clause $\bot$ and the following constrained clauses:

$$x = y \parallel const(x, y) \tag{7.53}$$
$$x \neq y \parallel const(x, y) \tag{7.54}$$
$$\mathsf{car}(x) = y \parallel const(x, y) \tag{7.55}$$
$$\mathsf{cdr}(x) = y \parallel const(x, y) \tag{7.56}$$
$$\mathsf{cons}(x, y) = z \parallel const(x, y, z) \tag{7.57}$$

where x, y and z are constrained variables.

We choose an LPO ordering $>$ over the terms, whose underlying precedence over the symbols of the signature Σ_{PEL} respects the following requirement: $\mathsf{cons} > \mathsf{cdr} > \mathsf{car} > \mathsf{nil} > c$, where nil is a constant and c is any other constant than nil.

Lemma 8. *The saturation of $G_0 \cup Ax(PEL)$ by $\mathcal{SPC}$ consists of G_0, $Ax(PEL)$ and the following constrained clauses:*

$$\mathsf{car}(x) = y \vee z = \mathsf{nil} \quad \| \quad const(x, y, z) \tag{7.58}$$
$$\mathsf{cdr}(x) = y \vee z = \mathsf{nil} \quad \| \quad const(x, y, z) \tag{7.59}$$
$$\mathsf{cons}(x, y) = z \vee z = \mathsf{nil} \quad \| \quad const(x, y, z) \tag{7.60}$$
$$\mathsf{cons}(x, \mathsf{cdr}(y)) = z \vee z = \mathsf{nil} \quad \| \quad const(x, y, z) \tag{7.61}$$
$$\mathsf{cons}(\mathsf{car}(x), y) = z \vee z = \mathsf{nil} \quad \| \quad const(x, y, z) \tag{7.62}$$
$$\mathsf{car}(x) = \mathsf{car}(y) \vee z = \mathsf{nil} \quad \| \quad const(x, y, z) \tag{7.63}$$
$$\mathsf{cdr}(x) = \mathsf{cdr}(y) \vee z = \mathsf{nil} \quad \| \quad const(x, y, z) \tag{7.64}$$
$$\mathsf{cons}(\mathsf{car}(x), \mathsf{cdr}(y)) = z \vee z = \mathsf{nil} \quad \| \quad const(x, y, z) \tag{7.65}$$

Proof. The only rule that applies to the set of axioms is the *Superposition* rule. *Superposition* between (7.50) and (7.47) yields $\mathsf{cons}(x, \mathsf{cdr}(\mathsf{cons}(x, y))) = \mathsf{cons}(x, y) \vee \mathsf{cons}(x, y) = \mathsf{nil}$, that is simplified by (7.48) to $\mathsf{cons}(x, y) = \mathsf{cons}(x, y) \vee \mathsf{cons}(x, y) = \mathsf{nil}$ that is immediately eliminated by the *Tautology* rule. *Superposition* between (7.47) and (7.50) yields $\mathsf{car}(y) = \mathsf{car}(y), y = \mathsf{nil}$, which is immediately eliminated by the *Tautology* rule. Similarly, *Superposition* between (7.50) and (7.48) yields $\mathsf{cons}(\mathsf{car}(\mathsf{cons}(x, y)), y) = \mathsf{cons}(x, y) \vee \mathsf{cons}(x, y) = \mathsf{nil}$, that is simplified by (7.47) to $\mathsf{cons}(x, y) = \mathsf{cons}(x, y) \vee \mathsf{cons}(x, y) = \mathsf{nil}$ that is immediately eliminated by the *Tautology* rule. *Superposition* between (7.48) and (7.50) yields $\mathsf{cdr}(y) = \mathsf{cdr}(y), y = \mathsf{nil}$, which gets deleted by the *Tautology* rule. *Superposition* between (7.50) and (7.51) gives $\mathsf{cons}(\mathsf{nil}, \mathsf{cdr}(\mathsf{nil})) = \mathsf{nil} \vee \mathsf{nil} = \mathsf{nil}$ which is eliminated by the *Tautology* rule. Similarly, *Superposition* between (7.50) and (7.52) gives $\mathsf{cons}(\mathsf{car}(\mathsf{nil}), \mathsf{nil}) = \mathsf{nil} \vee \mathsf{nil} = \mathsf{nil}$ which is also eliminated by the *Tautology* rule.

All the applications of the *Superposition* rule between two constrained clauses in G_0 generate clauses that are redundant w.r.t. G_0. *Superposition* between (7.55) (resp. (7.56) or (7.57)) and (7.53) generates a renamed copy of (7.55) (resp. (7.56) or (7.57)) that is immediately subsumed by the *Subsumption* rule. The application of the *Reflection* rule generates an empty clause which is already in G_0. The *Eq. Factoring* rule does not apply to the unitary clauses in G_0. Thus, the set G_0 is saturated.

Let us now consider all the applications of the *Superposition* rule between an axiom and a constrained clause. *Superposition* between (7.50) and (7.55) yields a new constrained clause

$$\mathsf{cons}(y, \mathsf{cdr}(x)) = x \vee x = \mathsf{nil} \quad \| \quad const(x, y) \tag{7.66}$$

Superposition between (7.47) and (7.66) yields a new constrained clause

$$\mathsf{car}(x) = y \vee x = \mathsf{nil} \quad \| \quad const(x, y) \tag{7.67}$$

that can be superposed with (7.53) to generate the new constrained clause (7.58). *Superposition* between (7.50) and (7.56) yields a new constrained clause

$$\mathsf{cons}(\mathsf{car}(x), y) = y \vee y = \mathsf{nil} \quad \| \quad const(x, y) \tag{7.68}$$

Superposition between (7.48) and (7.68) yields a new constrained clause

$$\mathsf{cdr}(x) = y \vee x = \mathsf{nil} \quad \| \quad const(x, y) \tag{7.69}$$

that can be superposed with (7.53) to generate the new constrained clause (7.59). *Superposition* between (7.66) and (7.56) yields the new constrained clause (7.60). *Superposition* between (7.66) and (7.53) yields the new constrained clause (7.61). *Superposition* between (7.68) and (7.53) yields the new constrained clause (7.62). *Superposition* between (7.47) and (7.62) gives a constrained clause

$$\mathsf{car}(x) = \mathsf{car}(y) \vee x = \mathsf{nil} \quad \| \quad const(x, y) \tag{7.70}$$

Superposition between (7.50) and (7.70) gives a constrained clause

$$\mathsf{cons}(\mathsf{car}(x), \mathsf{cdr}(y)) = y \vee y = \mathsf{nil} \quad \| \quad const(x, y) \tag{7.71}$$

whose superposition with (7.53) generates the new constrained clause (7.65) that subsumes (7.71). *Superposition* between (7.70) its renamed copy produces the new constrained clause (7.63) that subsumes (7.70). *Superposition* between (7.48) and (7.61) gives a constrained clause

$$\mathsf{cdr}(x) = \mathsf{cdr}(y) \vee x = \mathsf{nil} \quad \| \quad const(x, y) \tag{7.72}$$

whose superposition with its renamed copy generates the new constrained clause (7.64) that subsumes (7.72). *Superposition* between (7.50) and (7.65) gives a constrained clause $x = y \vee y = \mathsf{nil} \parallel const(x, y)$ that is eliminated by the *Schematic Deletion* rule. *Superposition* between (7.50) and (7.58) gives a constrained clause $\mathsf{cons}(x, \mathsf{cdr}(y)) = y \vee y = \mathsf{nil} \vee x = \mathsf{nil} \parallel const(x, y)$ that is eliminated by *Schematic Deletion*. Similarly, *Superposition* between (7.50) and (7.59) gives a constrained clause $\mathsf{cons}(\mathsf{car}(x), y) = y \vee x = \mathsf{nil} \vee y = \mathsf{nil} \parallel const(x, y)$ that is also eliminated by *Schematic Deletion*. *Superposition* between (7.63) (resp. (7.62), (7.65)) and (7.51) generates an elementary instance of (7.58) (resp. (7.60), (7.61)). Similarly, *Superposition* between (7.64) (resp. (7.61), (7.65)) and (7.52) yields an elementary instance of (7.59) (resp. (7.60), (7.61)). All these instances are eliminated by *Subsumption*. *Superposition* between (7.58) and (7.51) and between (7.59) and (7.52) gives clauses that are composed of equalities between constrained variables and constants, that are deleted by the *Schematic Deletion* rule. All the possible applications of the *Superposition* rule between new generated clauses give clauses that are redundant with respect to the set $Ax(PEL) \cup G_0 \cup \{(7.58), (7.59), (7.60), (7.61), (7.62), (7.63), (7.64), (7.65)\}$. The *Eq. Factoring* rule cannot be applied to this set because there is no clause satisfying the side condition of that rule. Therefore, we can conclude that the obtained set is saturated.

□

Lemma 8 is consistent with the termination proof in [ABRS09] for any concrete saturation by $\mathcal{PC}$, but one can remark that descriptions of saturations slightly differ. For example, clauses of the form $\mathsf{car}(e_1) = e_2 \vee \bigvee_{i=1}^{n} c_i = d_i$ with $n \geq 1$ generated by $\mathcal{PC}$ [ABRS09] correspond to the constrained clause $\mathsf{car}(x) = y \vee z = \mathsf{nil} \parallel const(x, y, z)$ generated by $\mathcal{SPC}$.

From an encoding of $G_0 \cup Ax(PEL)$ our tool generates the schematic saturation given in Lemma 8. Moreover, the system traces show how the new constrained clauses are generated:

$\mathbf{sup}($
$\mathbf{sup}($
$label(7.47),$
$\mathbf{sup}(label(7.50), label(7.55), \mathsf{car}(x), \mathsf{cons}(\mathsf{car}(Y), \mathsf{cdr}(Y)), \mathsf{cons}([], \mathsf{cdr}(Y))),$
$\mathsf{cons}(y, \mathsf{cdr}(x)), \mathsf{car}(\mathsf{cons}(X, Y)), \mathsf{car}([])$
$), label(7.53), x, \mathsf{car}(x), \mathsf{car}([])$
$)$ **gives** $clause((\mathsf{car}(x) = y, z = \mathsf{nil})) \parallel const(x, y, z)$
$\mathbf{sup}($
$\mathbf{sup}($
$label(7.48),$
$\mathbf{sup}(label(7.50), label(7.56), \mathsf{cdr}(x), \quad \mathsf{cons}(\mathsf{car}(Y), \mathsf{cdr}(Y)), \mathsf{cons}(\mathsf{car}(Y), [])),$
$\mathsf{cons}(\mathsf{car}(x), y), \mathsf{cdr}(\mathsf{cons}(X, Y)), \mathsf{cdr}([])$
$), label(7.53), x, \mathsf{cdr}(x), \mathsf{cdr}([])$
$)$ **gives** $clause((\mathsf{cdr}(x) = y, z = \mathsf{nil})) \parallel const(x, y, z)$
$\mathbf{sup}($
$\mathbf{sup}(label(7.50), label(7.55), \mathsf{car}(x), \mathsf{cons}(\mathsf{car}(Y), \mathsf{cdr}(Y)), \mathsf{cons}([], \mathsf{cdr}(Y))),$
$label(7.56), \mathsf{cdr}(x), \mathsf{cons}(y, \mathsf{cdr}(x)), \mathsf{cons}(y, [])$
$)$ **gives** $clause((\mathsf{cons}(x, y) = z, z = \mathsf{nil})) \parallel const(x, y, z)$
$\mathbf{sup}($
$\mathbf{sup}(label(7.50), label(7.55), \mathsf{car}(x), \mathsf{cons}(\mathsf{car}(Y), \mathsf{cdr}(Y)), \mathsf{cons}([], \mathsf{cdr}(Y))),$
$label(7.53), x, \mathsf{cons}(y, \mathsf{cdr}(x)), \mathsf{cons}(y, \mathsf{cdr}([]))$
$)$ **gives** $clause((\mathsf{cons}(x, \mathsf{cdr}(y)) = z, z = \mathsf{nil})) \parallel const(x, y, z)$
$\mathbf{sup}($
$\mathbf{sup}(label(7.50), label(7.56), \mathsf{cdr}(x), \mathsf{cons}(\mathsf{car}(Y), \mathsf{cdr}(Y)), \mathsf{cons}(\mathsf{car}(Y), [])),$
$label(7.53), x, \mathsf{cons}(\mathsf{car}(x), y), \mathsf{cons}(\mathsf{car}([]), y))$
$)$ **gives** $clause((\mathsf{cons}(\mathsf{car}(x), y) = z, z = \mathsf{nil})) \parallel const(x, y, z)$
$\mathbf{sup}(\ \mathbf{sup}((7.47), (7.62), \mathsf{cons}(\mathsf{car}(x), y), \mathsf{car}(\mathsf{cons}(X, Y)), \mathsf{car}([])),$
$\mathbf{sup}((7.47), (7.62), \mathsf{cons}(\mathsf{car}(x), y), \mathsf{car}(\mathsf{cons}(X, Y)), \mathsf{car}([]))$
$)$ **gives** $clause((\mathsf{car}(x) = \mathsf{car}(y), z = \mathsf{nil})) \parallel const(x, y, z)$
$\mathbf{sup}(\ \mathbf{sup}((7.48), (7.61), \mathsf{cons}(x, \mathsf{cdr}(y)), \mathsf{cdr}(\mathsf{cons}(X, Y)), \mathsf{cdr}([])),$
$\mathbf{sup}((7.48), (7.61), \mathsf{cons}(x, \mathsf{cdr}(y)), \mathsf{cdr}(\mathsf{cons}(X, Y)), \mathsf{cdr}([]))$
$)$ **gives** $clause(\mathsf{cdr}(x) = \mathsf{cdr}(y) z = \mathsf{nil}) \parallel const(x, y, z)$
$\mathbf{sup}($
$\mathbf{sup}($
$label(7.50),$
$\mathbf{sup}(label(7.47), label(7.62), \mathsf{cons}(\mathsf{car}(x), y), \mathsf{car}(\mathsf{cons}(X, Y)), \mathsf{car}([])),$
$\mathsf{car}(x), \mathsf{cons}(\mathsf{car}(Y), \mathsf{cdr}(Y)), \mathsf{cons}([], \mathsf{cdr}(Y))$
$), label(7.53), x, \mathsf{cons}(\mathsf{car}(x), \mathsf{cdr}(y)), \mathsf{cons}(\mathsf{car}(x), \mathsf{cdr}([]))$
$)$ **gives** $clause((\mathsf{cons}(\mathsf{car}(x), \mathsf{cdr}(y)) = z, z = \mathsf{nil})) \parallel const(x, y, z)$

7.8 Theory of arrays

The many-sorted signature Σ_A of the theory of arrays is the set of function symbols $\{\mathsf{select} : \text{ARRAY} \times \text{INDEX} \to \text{ELEM}, \mathsf{store} : \text{ARRAY} \times \text{INDEX} \times \text{ELEM} \to \text{ARRAY}\}$.

This theory is axiomatized by the following set $Ax(A)$ of axioms

$$\mathsf{select}(\mathsf{store}(V, I, E), I) = E \tag{7.73}$$
$$\mathsf{select}(\mathsf{store}(V, I, E), J) = \mathsf{select}(V, J) \vee I = J \tag{7.74}$$

where V is a universally quantified variable of sort ARRAY, I, J are universally quantified variables of sort INDEX, and E is a universally quantified variable of sort ELEM.

The set G_0 consists of the empty clause $\bot$ and the following constrained clauses over the signature Σ_A:

1. Sort ARRAY
   a) $\mathsf{store}(a_1, i, e) = a_2 \parallel const(a_1, i, e, a_2)$
   b) $a_1 = a_2 \parallel const(a_1, a_2)$
   c) $a_1 \neq a_2 \parallel const(a_1, a_2)$
2. Sort INDEX
   a) $i_1 = i_2 \parallel const(i_1, i_2)$
   b) $i_1 \neq i_2 \parallel const(i_1, i_2)$
3. Sort ELEM
   a) $\mathsf{select}(a, i) = e \parallel const(a, i, e)$
   b) $e_1 = e_2 \parallel const(e_1, e_2)$
   c) $e_1 \neq e_2 \parallel const(e_1, e_2)$

where a, a_1, a_2 are constrained variables of sort ARRAY, i, i_1, i_2 are constrained variables of sort INDEX, and e, e_1, e_2 are constrained variables of sort ELEM.

The LPO ordering $>$ is used with the following requirement: $\mathsf{store} > \mathsf{select} > a > e > i$ for all constants a of sort ARRAY, e of sort ELEM and i of sort INDEX.

Lemma 9. *The saturation of $G_0 \cup Ax(A)$ by $\mathcal{SPC}$ consists of G_0, $Ax(A)$ and the following constrained clauses:*

$$\mathsf{select}(a_1, I) = \mathsf{select}(a_2, I) \vee i = I \parallel const(a_1, a_2, i) \tag{7.75}$$
$$\mathsf{select}(a, i) = e \vee i_1 = i_2 \parallel const(a, i, e, i_1, i_2) \tag{7.76}$$

where I is a universally quantified variable of sort INDEX.

Proof. The only rule that applies to the set of axioms is the *Superposition* rule. This application between two axioms generates $\mathsf{select}(A, J) = E \vee J = J$ that is immediately eliminated by the *Tautology* rule. Thus, the set of axioms $Ax(A)$ is saturated.

All the applications of the *Superposition* rule between two constrained clauses in G_0 generate clauses that are redundant w.r.t. G_0. *Superposition* between (1.a) and (1.b) (resp. ((2.a) or 3.b)) yields a renamed copy of (1.a) which is subsumed by the *Subsumption* rule. Similarly, *Superposition* between (3.a) and (1.b) (resp. (2.a)) yields a renamed copy of (1.a) which gets subsumed by the *Subsumption* rule. The application of the *Reflection*

rule generates the empty clause which is already in G_0. The *Eq. Factoring* rule does not apply to the unitary clauses in G_0. Thus, the set G_0 is also saturated.

Let us now consider all the applications of the *Superposition* rule between an axiom and a constrained clause. *Superposition* between (7.73) and (1.a) yields a renamed copy of (3.a), which is immediately removed by the *Subsumption* rule. *Superposition* between (7.74) and (1.a) yields the new constrained clause (7.75). *Superposition* between (7.75) and its renamed copy yields $\mathsf{select}(a_1, I) = \mathsf{select}(a_2, I) \vee i_1 = I \vee i_2 = J \parallel \mathit{const}(a_1, a_2, i_1, i_2)$, which is removed by the *Schematic Deletion* rule. *Superposition* between (7.75) and (3.a) yields $\mathsf{select}(a_1, j) = e \vee i = j \parallel \mathit{const}(a_1, e, i, j)$, which can be superposed with (2.a) to generate the new constrained clause (7.76). *Superposition* between (7.76) and its renamed copy yields $e_1 = e_2 \vee i_1 = i_2 \parallel \mathit{const}(e_1, e_2, i_1, i_2)$, removed by *Schematic Deletion*. *Superposition* between any axiom and (1.b), (3.b) or (2.a) yields constrained clauses that are redundant w.r.t. $G_0 \cup Ax(A) \cup \{(7.75), (7.76)\}$. Similarly, *Superposition* between any new constrained clause and (1.b), (3.b) or (2.a) generates constrained clauses that are also redundant w.r.t. $G_0 \cup Ax(A) \cup \{(7.75), (7.76)\}$. Since no other rule can be applied to this set, we conclude that it is saturated. □

We can observe that the schematic saturation computed by $\mathcal{SPC}$ corresponds to the description given in [ABRS09] for any concrete saturation computed by $\mathcal{PC}$.

From an encoding of $G_0 \cup Ax(A)$ our tool generates the schematic saturation given in Lemma 9. Moreover, the system traces show that the new constrained clauses are generated by the *Superposition* rule as follows:

$\mathbf{sup}((7.74), (1.a),\ \mathsf{store}(a_1, i, e),\ \mathsf{select}(\mathsf{store}(V, I, E), J),\ \mathsf{select}([], J))$
$\quad \mathbf{gives}\ \mathit{clause}(\mathsf{select}(a_1, I) = \mathsf{select}(a_2, I), i = I) \parallel \mathit{const}(a_1, a_2, i)$
$\mathbf{sup}(\mathbf{sup}((3.a), (7.75),\ \mathsf{select}(a_1, I),\ \mathsf{select}(a, i),\ []), (2.a),\ i_1,\ \mathsf{select}(a_1, i),\ \mathsf{select}(a_1, []))$
$\quad \mathbf{gives}\ \mathit{clause}(\mathsf{select}(a, i) = e,\ i_1 = i_2) \parallel \mathit{const}(a, i, e, i_1, i_2)$

7.9 Theory of recursive data structures

We consider here the unsorted theory of recursively-defined data structures [LM02] of length 3 given by the signature $\Sigma_{RDS} = \bigcup_{i=1}^{3}\{\mathsf{sel}_i\} \cup \{\mathsf{injc}\}$ and axiomatized by the following set $Ax(RDS)$ of axioms:

$$\mathsf{sel}_1(\mathsf{injc}(X_1, X_2, X_3)) = X_1 \tag{7.77}$$
$$\mathsf{sel}_2(\mathsf{injc}(X_1, X_2, X_3)) = X_2 \tag{7.78}$$
$$\mathsf{sel}_3(\mathsf{injc}(X_1, X_2, X_3)) = X_3 \tag{7.79}$$
$$\mathsf{injc}(X_1, X_2, X_3) \neq \mathsf{injc}(X_1, X_2, X_3), X_1 = Y_1 \tag{7.80}$$
$$\mathsf{injc}(X_1, X_2, X_3) \neq \mathsf{injc}(X_1, X_2, X_3), X_2 = Y_2 \tag{7.81}$$
$$\mathsf{injc}(X_1, X_2, X_3) \neq \mathsf{injc}(X_1, X_2, X_3), X_3 = Y_3 \tag{7.82}$$

where X_1, X_2, X_3, Y_1, Y_2, Y_3 are universally quantified variables.

Let G_0 be the set composed of the empty clause $\bot$ and the following constrained clauses:

$$x = y \quad \| \quad const(x, y) \tag{7.83}$$
$$x \neq y \quad \| \quad const(x, y) \tag{7.84}$$
$$\mathsf{sel}_1(x) = y \quad \| \quad const(x, y) \tag{7.85}$$
$$\mathsf{sel}_2(x) = y \quad \| \quad const(x, y) \tag{7.86}$$
$$\mathsf{sel}_3(x) = y \quad \| \quad const(x, y) \tag{7.87}$$
$$\mathsf{injc}(x, y, z) = w \quad \| \quad const(x, y, z, w) \tag{7.88}$$

where x, y, z, w are constrained variables.

The LPO ordering $>$ is used with the following requirement: $\mathsf{sel}_i > \mathsf{injc} > c$ for all $i \in \{1, 2, 3\}$ and for every constant c.

Lemma 10. *The saturation of $G_0 \cup Ax(RDS)$ by $\mathcal{SPC}$ consists of G_0, $Ax(RDS)$ and the following constrained clauses:*

$$\mathsf{injc}(X1, X2, X3) \neq x,\ y = X1 \quad \| \quad const(x, y) \tag{7.89}$$
$$\mathsf{injc}(X1, X2, X3) \neq x,\ y = X2 \quad \| \quad const(x, y) \tag{7.90}$$
$$\mathsf{injc}(X1, X2, X3) \neq x,\ y = X3 \quad \| \quad const(x, y) \tag{7.91}$$

Proof. The set of axioms $Ax(RDS)$ is saturated. The set G_0 is also saturated.

Let us now consider all the applications of the *Superposition* rule between an axiom and a constrained clause. *Superposition* between (7.77) and (7.88) yields a renaming copy of (7.85), which is immediately removed by the *Subsumption* rule. Similarly, *Superposition* between (7.78) and (7.88) and between (7.79) and (7.88) yields respectively renaming copies of (7.86) and (7.87). *Superposition* between (7.80) and (7.88) yields (7.89) that can be superposed with (7.88) to generate

$$w \neq x \vee y = z \quad \| \quad const(x, y, z, w) \tag{7.92}$$

But this clause is immediately removed by the *Schematic Deletion* rule. *Superposition* between (7.81) and (7.88) yields (7.90) whose superposition with (7.88) generates a new clause (7.92) that is immediately eliminated by the *Schematic Deletion* rule. Similarly, *Superposition* between (7.82) and (7.88) yields (7.91) whose superposition with (7.88) gives (7.92) that is immediately removed by the *Schematic Deletion* rule.

Since no other rule can be applied to the set $G_0 \cup Ax(RDS) \cup \{(7.89), (7.90), (7.91)\}$, we conclude that it is saturated. □

From an encoding of $G_0 \cup Ax(RDS)$ our tool generates the schematic saturation given in Lemma 10. Moreover, the system traces show that the new constrained clauses are generated by the *Superposition* rule as follows:

$$\mathbf{sup}(label(7.80), label(7.88), \mathsf{injc}(x, y, z), \mathsf{injc}(X1, X2, X3), []) \ \mathbf{gives}$$
$$clause((\mathsf{injc}(X1, X2, X3) \neq x,\ y = X1)) \| const(x, y)$$
$$\mathbf{sup}(label(7.81), label(7.88), \mathsf{injc}(x, y, z), \mathsf{injc}(X1, X2, X3), []) \ \mathbf{gives}$$
$$clause((\mathsf{injc}(X1, X2, X3) \neq x,\ y = X2)) \| const(x, y)$$
$$\mathbf{sup}(label(7.82), label(7.88), \mathsf{injc}(x, y, z), \mathsf{injc}(X1, X2, X3), []) \ \mathbf{gives}$$
$$clause((\mathsf{injc}(X1, X2, X3) \neq x,\ y = X3)) \| const(x, y)$$

	Theory	Reference	Conformity
1	Lists without extensionality	[LM02]	✓
2	Lists with extensionality	[LRRT11]	×
3	Records	[ABRS09]	✓
4	Lists with length	[NRR09c]	✓
5	Lists with integer elements	[NRR09c]	✓
6	Records with increment	[NRR09c]	✓
7	Possibly empty lists	[ABRS09]	✓
8	Arrays	[ABRS09]	✓
9	Recursive data structure	[LRRT11]	✓

Figure 7.1: Experimental results

7.10 Combinability

This experimentation has handled theories for lists, records, arrays and recursive data structure. We have considered two classes of theories:

1. The first one uses standard schematic paramodulation calculus. These are the classical theories such as the theory of lists with and without extensionality, the theory of records without extensionality, the theory of possibly empty lists, the theory of arrays and the theory of recursive data structures.

2. The second one uses schematic paramodulation calculus modulo Integer Offsets. These are the theory of lists with length, the theory of lists with integer element and the theory of records with increment.

In [ARR03, LM02, ABRS09] we can find pen-and-paper proofs that any saturation of the theories of the first class is finite with respect to $\mathcal{PC}$. Our implementation of schematic paramodulation provides mechanical proofs of these results. After computing a finite schematic saturation for all these theories, our tool automatically checks that it does not contain any variable-active clause. According to Theorem 10, we can conclude that all these theories are combinable with $\mathcal{PC}$. In [NRR09c] we can find proofs that any saturation of the theories of the second class is finite with respect to $\mathcal{UPC}_I$. Our implementation of schematic paramodulation modulo Integer Offsets provides mechanical proofs of these results. After computing a finite schematic saturation for all these theories, our tool automatically checks that this saturation is safe. According to Theorem 15, we can conclude that all these theories are combinable with $\mathcal{PC}$.

7.11 Summary

This chapter has presented our experimental results. We have tested nine theories for which paramodulation is known to terminate. In Figure 7.1 the reader can see that for eight out of nine considered theories our tool has automatically proved that the results

given in the literature are correct. More surprisingly, for the theory of lists with extensionality (Section 7.2), our implementation reveals that the saturation given in [LRRT11] is incomplete. One can notice that we have only considered the theory of records of length 3. In fact, the schematic paramodulation calculus considers only finite set G_0 schematizing any set of ground flat literals built over the signature. But of course the length can be increased.

Chapter 8

Modular specification of generic Java methods and classes

Contents

The work presented in this chapter has been done in the framework of the INRIA CeProMi[2] "Action de Recherche Collaborative" (ARC). One of the objectives of the ARC

[2] `http://www.lri.fr/cepromi`

is modular specification and proof of properties of Java or C programs. A well conceived program is developed in a modular way, that is by the structured assembly of simpler components. The goal is also to get modularity to prove modular programs.

A specification language is a formal language used in computer science during requirement analysis and system design. Most programming languages are directly executable formal languages. They are used to implement a system. Specification languages are generally not directly executed. They describe the system at a much higher level than a programming language. There are many specification languages like CASL [ABK+01], JML [?, LC06], Spec# [LP09], Z [Spi92], B [Abr05], etc.

Some members of the ARC project develop a specification language for Java programs called the Krakatoa Modeling Language (KML). The main features of this language are presented in Section 8.1. The question of automaticity of algorithm proofs is addressed in Section 8.2 through the typical example of a sorting algorithm.

A new feature introduced in Java 5 is genericity. Generics enable types (classes and interfaces) to be parameters when defining classes, interfaces and methods. Type parameters provide a way to re-use the same code with different inputs. This notion is supported neither by upstream JML (Java Modeling Language) nor KML. Supporting genericity naturally requires adding type parameters to specifications, but not only: More complex issues arise when one tries to formally specify generic programs. The last two sections address the question of modular specification of generic Java classes and methods. They propose extensions of the KML language to allow specifications of these generic classes and methods, which essentially amount to

- add type parametricity in that specification language;
- add a notion of instantiation of *theories* used to model programs.

Our proposal is illustrated by two examples. The former is the specification of an algorithm to sort a Java array. It is presented in Section 8.3. The latter is the specification of the `java.util.HashMap` class and its use for memoization. It is presented in Section 8.4.

8.1 Overview of Krakatoa Modeling Language (KML)

This section describes a specification language for the Java programming language, named Krakatoa Modeling Language (KML). KML is a new specification language for Java programs. It is designed to reduce the distance between programming and proving activities.

Why is a generic platform for program verification [FM07]. From a source program annotated by definite specifications, the Why platform extracts the proof obligations and transmits them to provers like Simplify, Yices, Alt-Ergo, etc. The Krakatoa tool is a part of the Why platform. Krakatoa expects a Java source file as input, annotated with the Krakatoa Modeling Language. KML is largely inspired from JML. KML specifications are given as annotations in the source code, in a special style of comments after `//@ ...` or between `/*@` and `@*/`. KML also shares many features with the ANSI/ISO C Specification Language [BFM+09].

8.1.1 Basic standard features

Method contracts

Method contracts are made of a precondition and a set of behaviors. The precondition is a proposition introduced by `requires` keyword which is supposed to hold in the pre-state of the method, i.e. when it is called. It must be checked valid by the caller.

A normal behavior has the form:

```
/*@ requires R;
  @ behavior b:
  @  assumes A;
  @  assigns L;
  @  ensures E;
  @*/
```

where `R` and `E` are logical assertions, `R` is a precondition and `E` is a postcondition, `L` is a set of memory locations, that may be modified by the method. In `E`, the notation `\result` denotes the returned value. The semantics of such a behavior is the following: The callee guarantees that if it returns normally, then in the post-state:

- `\old(A)` $\Rightarrow$ `E` holds
- If `\old(A)` holds, each location of the pre-state not in `L` remains allocated and unchanged in the post-state

where `\old(A)` denotes the value of `A` in the pre-state. An exceptional behavior has the form

```
/*@ requires R;
  @ behavior b:
  @  assumes A;
  @  assigns L;
  @  signals (Exc x) E;
  @*/
```

The semantics is similar to normal behaviors, but here properties must hold when the method terminates abruptly with exception of class `Exc`.

Statement annotations

A **loop annotation** can be given just in front of a loop construct (while, for, etc.). It is of the form:

```
/*@ loop_invariant I
  @ for b: loop_invariant Ib;
  @ loop_variant V;
  @*/
```

It states that `I` is an inductive invariant: it must hold at loop entry and be preserved by any iteration of the loop body. The loop invariant `Ib` must also be an inductive invariant, but under `assumes A` of behavior `b`. The loop variant, if given, must be an expression of integer type, which must decrease at each loop iteration, and remain non-negative.

Class invariants

A class `invariant` is a property attached to a class. It is of the form:

```
/*@ invariant id: e; @*/
```

This property must be established by constructors, and preserved by each method of the class.

8.1.2 Logical specifications

Logic functions and predicates

KML **does not allow pure methods** to be used in annotations. However, it permits to declare new logic functions and predicates. They must be placed at the global level, i.e. outside any class declaration, and are respectively of the form

```
//@ logic m id(m1 x1, .. , mn xn) = e;
```

and

```
//@ predicate id(m1 x1, .. , mn xn) = p;
```

where `x1`, ..., `xn` are variables, `e` must have type `m`, and `p` must be a proposition. The types `m` and `mi` can be either Java types or purely logic types, such as integer or real.

Logic functions and predicates can also be hybrid. It means that they depend on some memory state. More generally, they can depend on several memory states, by attaching several labels to them. A hybrid function and a predicate definition are of the following general form

```
//@ logic m id{L1, .. , Ln}(m1 x1, .. , mn xn) = e;
//@ predicate id{L1, .. , Ln}(m1 x1, .. , mn xn) = p;
```

where `L1`, ..., `Ln` are memory state labels on which the predicate or function depends, and `m`, `m1`, ... , `mn`, `e`, `p` have the same definition as presented before.

Lemmas

Lemmas are user-given propositions, a facility that might help theorem provers to establish the validity of KML specifications. A lemma is declared as

`//@ lemma id: p;`

Obviously, a complete verification of a KML specification must provide a proof for each lemma.

Inductive definitions

A predicate may also be defined by an *inductive* definition

```
/*@ inductive P(x1, ..., xn) {
  @   case c1 : p1;
  @    ...
  @   case cn : pn;
  @ }
  @*/
```

where `c1`, ..., `cn` are identifiers and `p1`, ..., `pn` are propositions. The semantics of this definition is that `P` is the least fixpoint of the cases, i.e. the smallest predicate (in the sense that it is false the most often) satisfying the propositions `p1`, ..., `pn`. To ensure existence of a least fixpoint, each of these propositions is required to be of the form

`\forall y1, ..., ym, h1 ==> ...==> hl ==> P(t1, ..., tn)`

where `P` occurs only positively in hypotheses `h1`, ..., `hl`.

Theories

Logical specifications were supported by Krakatoa/Why under the form of axiomatic blocks in Java source files within specification comments by declaring a set of types, a set of predicates and functions with expected profiles, and a set of axioms. Now it is defined in a separated file with the ".spec" extension. The syntax for defining a `theory` is the same as for an axiomatic block but without comments, as in the following example:

```
theory Th {
  type new_type;
  logic new_type func1;
  logic integer func2(new_type v, integer k);
  axiom axiom_name: axiom_body;
}
```

where `Th` is the theory name, and `axiom_body` is a closed formula.

Unlike inductive definitions, there is no syntactic conditions which would guarantee axiomatic definitions to be consistent. It is usually up to the user to ensure that the introduction of axioms does not lead to a logical inconsistency.

Construct \at and default logic labels

The construct `\at`(e, L) refers to the value of the expression e in the program state at label L. There exist predefined labels, e.g. `Old` and `Here`. `\old`(e) is in fact syntactic sugar for `\at`$(e,$`Old`$)$. The label `Here` is visible in all statement annotations, where it refers to the state where the annotation appears. It refers to the pre-state in a method precondition (`requires` clause), and to the post-state in a method postcondition (`ensures` clause). The label `Old` is visible in `ensures` clauses and refers to the pre-state of the method's contract.

More details about KML notions could be found in [Mar09].

In this section we have shortly described the Krakatoa Modeling Language. In the next section this specification language is used to prove a sorting algorithm.

8.2 Specification of a sorting algorithm

A sorting algorithm is an algorithm that puts elements of an array in a certain order. The resulting array must satisfy the following two properties:

1. The elements are in increasing order with respect to some ordering relation.
2. The elements in the sorted array are a permutation of the elements of the initial array.

In [FM99] several algorithms for sorting have been studies. The authors specify and prove them correct within the Why tool, but only on the particular instance of an array of integers and the usual "less-than" order. The first condition is specified by a predicate `(sorted t i j)` which expresses that array `t` is sorted in increasing order between the bounds `i` and `j`. The second condition is specified by a predicate `(permut t tt)` where `t` and `tt` are permutations of each other. They describe many ways to define such a predicate, but the best solution is to express that the set of permutations is the smallest equivalence relation containing the transpositions, i.e. exchanges of two elements. The predicate `(exchange t tt i j)` is defined for two arrays `t` and `tt` and two indexes `i` and `j`, and the predicate `(permut t tt)` is defined inductively for the following properties: reflexivity, symmetry and transitivity. The proofs are performed within the Coq proof assistant [The06].

A selection sorting algorithm is written in Java by Marché [Mar09] with a similar specification in KML. It is also specific to integers and the usual less-than order. The proof is done fully automatically within SMT solvers (namely Simplify and Alt-Ergo provers).

Our proposal is to re-use the bag datatype defined in [Mar07] and to rewrite the second condition by saying that the initial array and the resulting array have the same content.

This section is organized as follows: Section 8.2.1 presents the sorting algorithm by selection in Java, Section 8.2.2 presents this algorithm completed with a specification in KML. In Section 8.2.3 we specify the array content with a bag and discusses whether it can be proved automatically

try to prove the sorting algorithm automatically.

8.2.1 Selection sort in Java

The sorting algorithm by selection in Figure 8.1 is written in Java. There are two methods: the `swap` method just exchanges two array elements of given indexes. In the `selectionSort` method the integers `i` and `mi` are indexes for the current element and the minimal element respectively. The integer `mv` serves to store this minimal element. The

```
1. class Sort {
2.   /** method swapping 2 elements */
3.   void swap(int t[], int i, int j) {
4.     int tmp = t[i];
5.     t[i] = t[j];
6.     t[j] = tmp;
7.   }
9.   void selectionSort(int t[]) {
10.    int i, j;
11.    int mi, mv;
12.    for (i = 0; i < t.length - 1; i++) {
13.      mv = t[i];
14.      mi = i;
15.      for (j = i + 1; j < t.length; j++) {
16.        if (t[j] < mv) {
17.          mi = j;
18.          mv = t[j];
19.        }
20.      }
21.      swap(t, i, mi);
22.    }
23.  }
24.}
```

Figure 8.1: Selection sort in Java

minimal element is found in the remainder of the array and swapped with the current element.

This algorithm can be tested with different array examples, but it is not sure that it is always correct, i.e. it satisfies properties 1 and 2 described above for any array. A formal specification of these two properties constitutes the first step towards a formal proof of its correctness.

8.2.2 Sorting algorithm with a KML specification

In [Mar09] two postconditions for the method `selectionSort` are proved:

```
behavior sorts:
   ensures Sorted(t,0,t.length-1);
```

which means that the resulting array is in increasing order, and

```
behavior permuts:
   ensures Permut{Old,Here}(t,0,t.length-1);
```

which means that the resulting array is a permutation of the initial array.

```
predicate Sorted{L}(int a[], integer l, integer h) =
    \forall integer i; l <= i < h ==> \at(a[i] <= a[i+1], L) ;
```

Figure 8.2: Specification of the first property

```
inductive Permut{L1,L2}(int a[], integer l, integer h) {
  case Permut_refl{L}:
      \forall int a[], integer l h; Permut{L,L}(a, l, h) ;
  case Permut_sym{L1,L2}:
      \forall int a[], integer l h;
          Permut{L1,L2}(a, l, h) ==> Permut{L2,L1}(a, l, h) ;
  case Permut_trans{L1,L2,L3}:
      \forall int a[], integer l h;
          Permut{L1,L2}(a, l, h) && Permut{L2,L3}(a, l, h) ==> Permut{L1,L3}(a, l, h) ;
  case Permut_swap{L1,L2}:
      \forall int a[], integer l h i j;
          l <= i <= h && l <= j <= h &&
              Swap{L1,L2}(a, i, j) ==> Permut{L1,L2}(a, l, h) ;
}
```

Figure 8.3: Inductive predicate `Permut`

```
predicate Swap{L1,L2}(int a[], integer i, integer j) =
    \at(a[i],L1) == \at(a[j],L2) &&
    \at(a[j],L1) == \at(a[i],L2) &&
    \forall integer k; k != i && k != j ==>
        \at(a[k],L1) == \at(a[k],L2);
```

Figure 8.4: Predicate `Swap`

The `Sorted` predicate is presented in Figure 8.2. It is a hybrid predicate. It means that its value depends on the memory heap in some state `L`.

The `Permut` predicate presented in Figure 8.3 has two labels. This predicate is true whenever the slice of the array `a` from lower bound `l` to upper bound `h` in the state `L1` is a permutation of the same slice in the state `L2`. The predicate defines four properties: reflexivity, symmetry, transitivity and swap. The last case tells us that swapping two elements in the slice is a permutation.

The `Swap` predicate is reproduced in Figure 8.4. `Swap{L1,L2}(a,i,j)` is true if and only if the value of `a[i]` in the state of label `L2` equals the value of `a[j]` in the state of label `L1`, the value of `a[j]` in the state of label `L2` equals the value of `a[i]` in the state of label `L1`, and the value of `a[k]` is the same in both states, if `k` is different from `i` and `j`.

```
/*@ loop_invariant 0 <= i;
  @ for sorts:
  @  loop_invariant Sorted(t,0,i) &&
  @   (\forall integer k1 k2 ;
  @      0 <= k1 < i <= k2 < t.length ==> t[k1] <= t[k2]);
  @ for permuts:
  @  loop_invariant
  @   Permut{Pre,Here}(t,0,t.length-1);
  @*/
for (i = 0; i < t.length-1; i++) {
    mv = t[i];
    mi = i;
    /*@ loop_invariant
      @   i < j &&
      @   i <= mi < t.length &&
      @   mv == t[mi];
      @ for sorts:
      @  loop_invariant
      @   (\forall integer k; i <= k < j ==> t[k] >= mv);
      @ for permuts:
      @  loop_invariant
      @   Permut{Pre,Here}(t,0,t.length-1);
      @*/
    for (j = i + 1; j < t.length; j++) {
     ...
    }
 ...
}
```

Figure 8.5: Loop invariants

Figure 8.5 presents two loop invariants for the two loops in Figure 8.1. The loop invariants for the `sorts` behavior tell that the array is sorted up to index `i` in the external loop and that `mv` is a minimal element between `a[i]` and `a[j]` in the internal loop. The loop invariant for the `permuts` behavior is the same for the external and internal loops. It tells that the current array is a permutation of the initial array.

The algorithm with this specification is proved within the Simplify prover, except the postcondition for the `selectionSort` method which tells that the resulting array is a permutation of the initial array. This postcondition is proved within the Alt-Ergo prover. So, this algorithm is proved within the Simplify and the Alt-Ergo provers.

The proof results are satisfactory. However, we want to explore another way for the second property by re-using a bag datatype. More precisely, we try to prove the same algorithm but with a property 2 saying that the initial array and the resulting array have the same content.

```
type ibag;

// empty bag
logic ibag empty_bag();

// singleton(n)
logic ibag singleton(integer n);

// remove element n from bag b
logic ibag remove(integer n, ibag b);

// union b1 and b2
logic ibag union(ibag b1, ibag b2);
```

Figure 8.6: Signature for bags

```
axiom union_assoc:
 \forall ibag b1 b2 b3;
   union(union(b1,b2),b3) == union(b1,union(b2,b3));
axiom union_comm:
 \forall ibag b1 b2; union(b1,b2) == union(b2,b1);
axiom union_empty_id_left:
 \forall ibag b; union(empty_bag(),b) == b;
axiom union_empty_id_right:
 \forall ibag b; union(b,empty_bag()) == b;

axiom remove_union:
 \forall ibag b, integer x;
   remove(x,union(singleton(x),b)) == b;
```

Figure 8.7: Algebraic specification of bags

8.2.3 Specifying the sorting algorithm by selection with a bag

A bag (or multiset) is a collection without order. We want to associate with each array the bag of its elements, and to express that the output array is a permutation of the input array by writing that the corresponding bags are the same. It is a new way to prove property 2.

The type of bags is described by the following functions on Figure 8.6 and the following set of first-order axioms on Figure 8.7 that present some properties of bags. The first four axioms tell that `union` is associative, commutative and that the `empty_bag` is a neutral element for the union of bags. The last axiom establishes a relation between `union` and `remove`. When an element is removed from the union of a bag `b` and the bag containing only this element, the result is the bag `b`.

```
logic ibag boundContent{L1}(int[] a,
                            integer i, integer j) reads a[i..j];

axiom emptyContent{L1}:
  \forall int[] a; \forall integer i j;
      (i > j ==> boundContent{L4}(a,i,j) == empty_bag());

axiom nonemptyContent{L1}:
  \forall int[] a, integer i j;
      i <= j ==> boundContent{L4}(a,i,j) ==
            union(boundContent{L4}(a,i+1,j),singleton(a[i]));
```

Figure 8.8: Hybrid function for array content

```
boundContent{Old}(a,0,a.length-1) == boundContent{Here}(a,0,a.length-1);
```

Figure 8.9: Postcondition for `selectionSort` and `swap` methods

Figure 8.8 declares a hybrid function named `boundContent` which takes an array, a lower and an upper bound as parameters and returns a bag. The KML `reads` keyword says that `boundContent` just reads the array between `i` and `j`, it does not modify it. The first axiom says that `boundContent` returns the empty bag if the lower bound is greater than the upper bound. The second axiom says that, otherwise, the resulting bag is the union of the singleton bag containing the first array element and the content of the remaining part of the array.

It should be proved that the `swap` and `selectionSort` methods do not change the content of the slice as shown in Figure 8.9.

This postcondition is proved for the `selectionSort` method, but is not proved for the `swap` method. The `selectionSort` method depends on the `swap` method, therefore, it is easy to prove, but proving the `swap` method requires induction because `boundContent` is inductively defined. To prove the `swap` method we must guide provers step by step with the following assertions which are presented in Figure 8.11.

The first assertion tells that the new content is obtained from the old content by replacing the old value of `a[i]` by the value of `a[j]` in the old content. The second assertion tells that the value of `a[j]` has not changed. Then the memory state between states `Old` and `Here` is labelled `Middle`. The last assertion tells that the new content is obtained from the previous content by removing the previous value of `a[j]` and adding the value of the local variable `tmp` (which is the old value of `a[i]`).

Moreover, the following lemma presented in Figure 8.12 is added. It says that whenever the elements of an array are the same at two states, except in some position `k`, then the array content at the second state can be obtained from its content at the first state by removing the element at position `k` in the first state and adding the element at position `k` in the second state.

	Proof obligations	Alt-Ergo 0.8	Simplify 1.5.4	Yices 1.0.21	Z3 2.2	CVC3 2.1
	Lemma	−	−	−	−	−
POs for the selectionSort method	Loop invariants	+	+	−	+	−
	Postcondition presented in Figure 8.9	+	+	+	+	−
	Postcondition using the `Sorted` predicate	−	+	−	−	−
	Pointer dereferencing	+	+	−	+	−
POs for the swap method	Postcondition presented in Figure 8.9	−	+	−	−	−
	PO for the `assigns` clause	−	+	−	+	−
	Postcondition using the `Swap` predicate	+	+	−	+	+
	Pointer dereferencing	+	+	−	+	+
	Number of proved POs	5	8	1	6	2

Figure 8.10: Results

Figure 8.10 presents proof obligations (POs) proved by five provers: Alt-Ergo 0.8, Simplify 1.5.4, Yices 1.0.21, Z3 2.2 and CVC3 2.1. There are nine POs: one PO for the `lemma`, four POs for the `selectionSort` method and four POs for the `swap` method. Since the lemma itself is not provable without induction it is proved by none of these provers. All POs are proved by the Simplify prover. Unlike Simplify the SMT solvers Alt-Ergo,Yices, Z3 and CVC3 fail to prove some of the POs.

As a conclusion, the sorting algorithm is proved for array elements with the `int` Java type. Nevertheless, we would like to prove this algorithm for every Java type, that is as a generic sorting algorithm. It is the matter of the next section.

```
  void swap(int a[], int i, int j) {
    int tmp = a[i];
    a[i] = a[j];
    /*@ for cont: assert
      @   boundContent{Here}(a,0,a.length−1) ==
      @     union(remove(\at(a[i],Pre),
      @            boundContent{Pre}(a,0,a.length−1)),
      @      singleton(\at(a[j],Pre)));
      @*/
    /*@ for cont: assert
      @    a[j] == \at(a[j],Pre);
      @*/
      Middle: {
    a[j] = tmp;
    /*@ for cont: assert
      @   boundContent{Here}(a,0,a.length−1) ==
      @     union(remove(\at(a[j],Middle),
      @            boundContent{Middle}(a,0,a.length−1)),
      @      singleton(tmp));
      @*/
    }
  }
```

Figure 8.11: Assertions to guide provers step by step

```
lemma UpdateContent{L1,L2}:
  \forall int[] a, integer i j k;
     // update of a[k]
     i <= k <= j &&
     (\forall integer l;
         i <= l <= j && k != l ==>
           \at(a[l],L1) == \at(a[l],L2))
     ==> boundContent{L2}(a,i,j) ==
           union(
             remove(\at(a[k],L1),
                    boundContent{L1}(a,i,j)),
             singleton(\at(a[k],L2)));
```

Figure 8.12: New lemma

```
class Main {
  public static void main(String[] args) {
    IntLtComparator intc = new IntLtComparator();
    Integer[] b = {new Integer(2),new Integer(1),new Integer(3)};
    java.util.Arrays.sort(b,intc);
    //@ assert b[0].value <= b[1].value;
  }
}
```

Figure 8.13: A sample client code calling the generic sorting method

```
class IntLtComparator implements Comparator<Integer> {
   public int compare(Integer x, Integer y) {
      if (x.intValue() < y.intValue()) return −1;
      if (x.intValue() == y.intValue()) return 0;
      return 1;
   }
}
```

Figure 8.14: The usual "less-than" comparator on integers

8.3 Generic sorting

Java generics are a language feature that allows definition and use of generic types and methods. Generics are needed for implementing a generic class that can be instantiated for a variety of types.

We want to specify a generic method for sorting arrays, where the array elements are of any type T. A significant challenge is to specify the ordering relation which is given as a parameter, under the form of a comparison function on T. As we will see, it is also important to study how this generic specification can be used by *client* code, because it has to be instantiated.

8.3.1 Generic sorting in Java

The class `java.util.Arrays` defines a generic sorting method with the profile:

```
public static <T> void sort(T[] a, Comparator<? super T> c)
```

In this method `<T>` is a type parameter and the syntax `<? super T>` denotes an unknown type that is a supertype of `T` (or `T` itself). The `java.util.Comparator<T>` interface imposes a total ordering on some collection of objects.

```
interface Comparator<T> {
  public int compare(T x, T y);
}
```

`T` is the type of objects that may be compared by this comparator. The method `compare` is expected to return a negative integer, zero, or a positive integer when the first argument

```
predicate Swap<T>{L1,L2}(T a[], integer i, integer j) =
 \at(a[i],L1) == \at(a[j],L2) &&
 \at(a[j],L1) == \at(a[i],L2) &&
 \forall integer k; k != i && k != j ==>
   \at(a[k],L1) == \at(a[k],L2);

inductive Permut<T>{L1,L2}(T a[], integer l, integer h){
 case Permut_refl{L}: \forall T a[], integer l h;
   Permut<T>{L,L}(a, l, h);
 case Permut_sym{L1,L2}: \forall T a[], integer l h;
   Permut<T>{L1,L2}(a, l, h) ==>
     Permut<T>{L2,L1}(a, l, h);
 case Permut_trans{L1,L2,L3}:
   \forall T a[], integer l h;
     Permut<T>{L1,L2}(a, l, h) &&
     Permut<T>{L2,L3}(a, l, h) ==>
       Permut<T>{L1,L3}(a, l, h);
 case Permut_swap{L1,L2}:
   \forall T a[], integer l h i j;
     l <= i <= h && l <= j <= h &&
     Swap<T>{L1,L2}(a, i, j) ==>
       Permut<T>{L1,L2}(a, l, h);
}
```

Figure 8.15: The permutation predicate

is respectively less than, equal to, or greater than the second one, for the desired ordering relation.

The sample code given in Figure 8.13 illustrates an instance of use of this `sort` method. It ends with a simple assertion which we expect to be able to prove, as a consequence of the generic specification we will provide. The validity of this assertion indeed depends on the comparator we choose. Here it is an instance of the class given in Figure 8.14, which implements the usual "less-than" ordering on integers. Changing this comparator, say to the "greater-than" ordering, would of course sort the array in decreasing order instead, violating the assertion.

8.3.2 Type parameters: the permutation property

The first extension we propose to KML is to allow type parameters in algebraic specifications, as follows

```
/*@ predicate id⟨T_1, ..., T_l⟩{L_1, ..., L_k}(t_1 x_1, ..., t_n x_n) = p;
  @*/
```

and similarly for functions, inductive predicates, and such.

For the sorting example, and following [Mar09], we define a `Permut`$\langle T\rangle\{L_1, L_2\}(a, l, h)$ predicate which means that the part of array a between indexes l and h, in some program

state L_1 is a permutation of the same array part in state L_2. It is defined inductively in Figure 8.15. The first postcondition of the `sort` method is then specified below.

```
/*@ ensures Permut<V>{Old,Here}(a,0,a.length-1);
  @*/
public static <V> void sort(V[] a, Comparator<? super V> cmp);
```

8.3.3 Theory parameters: the sorting property

The challenge is to specify the behavior of the comparator given as argument. What we propose is to allow to *pass theories as parameters*. On the sorting example, the first step is to define a general theory for types equipped with an ordering relation. Figure 8.16 shows a theory named `ComparatorTheory` which defines two predicates `eq` for equality and `lt` for an arbitrary strict total order. Equality is reflexive, symmetric and transitive. The strict total order satisfies four properties: irreflexivity, antisymmetry, totality and transitivity.

The `Comparator` interface should take some comparison theory as a parameter. This is shown in Figure 8.17. The `Comparator` interface thus has two parameters: a Java type `U` and a theory `Th`. The syntax `Th instantiating ComparatorTheory<U>` says that `Th` is an instance of the general theory defined in Figure 8.16. One may wonder why we require an *instance* of the comparison theory `ComparatorTheory`: this will be discussed in Section 8.3.4.

Figure 8.18 specifies the sorting property of the method `sort` with a second postcondition. The sorting method is not only parameterized by the type `V` but also by the type `W` which denotes the super type of `V` on which the comparator operates, and by a theory `th` which can be any instance of the general `ComparatorTheory` on `W`. Notice the new `as` keyword added to relate the anonymous Java type denoted by `?` and the explicit name `W` we need to introduce for it in the specification.

The comparator type is itself instantiated with the Java type `W` and the theory `th`. In the method postcondition the predicate `sorted` is then qualified with this theory.

8.3.4 Theory instantiation

To deal with our client program, we need more annotations. First we add specifications to the `java.lang.Integer` class: Figure 8.19 shows an excerpt of it annotated in KML[3].

Then, we need to specify the `IntLtComparator` class of Figure 8.14. For that, we first provide a theory which instantiates the general comparison theory `ComparatorTheory`, in Figure 8.20. The goal is to formalize that we decided to compare with the "less-than" ordering. The `instantiates` declaration generates verification conditions: this is discussed in Section 8.3.5. The class `IntLtComparator` shown in Figure 8.21 implements the instantiation of the `Comparator` interface where the Java type is the `Integer` class and the theory is the comparison theory for this class, defined in Figure 8.20.

[3]In KML, the private field is visible in the annotations of the public methods, whereas in JML, the field should be annotated with modifier `spec_public`.

```
theory ComparatorTheory<T> {
  predicate eq{L}(T x, T y);
  axiom eq_ref{L}: \forall T a; eq{L}(a,a);
  axiom eq_sym{L}: \forall T a b; eq{L}(a,b)
    ==> eq{L}(b,a);
  axiom eq_trans{L}: \forall T a1 a2 a3;
    eq{L}(a1, a2) && eq{L}(a2,a3)
      ==> eq{L}(a1,a3);

  predicate lt{L}(T x, T y);
  axiom lt_irref{L}: \forall T a; ! lt{L}(a,a);
  axiom lt_antisym{L}: \forall T a1 a2;
    !(lt{L}(a1,a2) && lt{L}(a2,a1))
  axiom lt_trans{L}:
    \forall T a1 a2 a3;
      lt{L}(a1,a2) && lt{L}(a2,a3)
        ==> lt{L}(a1,a3);
  axiom lt_totality{L}: \forall T a1 a2;
    eq{L}(a1,a2) || lt{L}(a1,a2)
      || lt{L}(a2,a1);

  predicate leq{L}(T x, T y) =
    eq{L}(x,y) || lt{L}(x,y);

  predicate sorted{L}(T[] a,
                      integer l, integer h) =
    \forall integer i; l <= i <h
      ==> leq{L}(\at(a[i],L),\at(a[i+1],L));
}
```

Figure 8.16: General theory for `Comparators`

Finally, notice that when checking the specific call to the method `sort` in the client program, the "implicit" parameters `V`, `W` and also the theory `Th` must be guessed. The value of `V` comes as usual with the Java typing, the value of `W` comes from the type of the `Comparator`. Then the theory `th` must be inferred from the theory argument of `cmp`, and it must be checked whether it really instantiates a convenient comparison theory.

8.3.5 Verification conditions for soundness

The soundness conditions to generate are as follows. First, the `IntLtComparatorTheory` should be a valid instantiation of the `ComparatorTheory<Integer>`. This amounts to check that the definitions of the predicates `eq` and `lt` given in `IntLtComparatorTheory` satisfy the axioms given in `ComparatorTheory<T>` when the type variable `T` is instantiated with `Integer`. This condition is easily discharged by SMT solvers. Second, the class `IntLtComparator` should correctly implement the interface `Comparator<Integer>`,

```
interface Comparator<U> /*@ <Th instantiating ComparatorTheory<U> > */ {

  /*@ ensures (Th.lt(x,y) <==> \result < 0) &&
    @         (Th.eq(x,y) <==> \result == 0) &&
    @         (Th.lt(y,x) <==> \result > 0);
    @*/
  public int compare(U x, U y);
}
```

Figure 8.17: Specification of the `Comparator` interface

```
/*@ ensures th.sorted(a,0,a.length-1);
  @*/
public static <V> /*@ <W> <th instantiating ComparatorTheory<W> > */
  void sort(V[] a, Comparator<? /*@ as W */ super V> /*@ <th> */ cmp);
```

Figure 8.18: Specification of the generic sorting method

```
public final class Integer extends Number implements Comparable {

    private int value;

    /*@ assigns this.value;
      @ ensures this.value == v;
      @*/
    public Integer(int v) { this.value = v; }

    /*@ assigns \nothing;
      @ ensures \result == this.value;
      @*/
    public int intValue() { return this.value; }
}
```

Figure 8.19: Annotated `Integer` class

```
theory IntLtComparatorTheory instantiates ComparatorTheory<Integer> {
  predicate eq{L}(Integer x, Integer y) = \at(x.value == y.value, L);
  predicate lt{L}(Integer x, Integer y) = \at(x.value < y.value,L);
}
```

Figure 8.20: Theory for "less-than" comparison

```
class IntLtComparator /*@ <IntLtComparatorTheory> */ {
    public int compare(Integer x, Integer y) { ... }
}
```

Figure 8.21: Specification of the `IntLtComparator` class of Figure 8.14

```
class Fib {
    HashMap<Integer,Long> memo;

    Fib() { memo = new HashMap<Integer,Long>(); }

    public long fib(int n) {
        if (n <= 1) return n;
        Integer n_obj = new Integer(n);
        Long x = memo.get(n_obj);
        if (x == null) {
            x = new Long(fib(n−1)+fib(n−2));
            memo.put(n_obj,x);
        }
        return x.longValue();
    }
}
```

Figure 8.22: Java source for Fib class

which requires that the method `compare` in the `IntLtComparator` class should satisfy the specification of the method `compare` declared in the interface `Comparator<U>`, when the type parameter `U` is instantiated with `Integer` and the theory parameter `Th` is instantiated with `IntLtComparatorTheory`. This condition is again easily proved by SMT solvers. Finally, checking the assertion in our client code can be done, by instantiating the generic postcondition `th.sorted`(...) of `sort` with the `IntLtComparatorTheory`. Substituting the `th.lt` predicate with its actual definition does the job. It is important to notice here that this final substitution is necessary, so it justifies why we initially parameterized the `sort` method with a theory parameter: otherwise, we would know that the array is sorted w.r.t some order, without knowing precisely which one.

8.4 Generic hashmaps

Finding the value associated with a given index is made efficient by use of classical hashing techniques. We present additional constructs needed when specifying generic *hash maps*. These are data types which build finite mappings from indexes of some type *key* to values of some other type *data*.

A simple but illustrating example of use of hash maps is a method for computing *Fibonacci numbers*[4]: $F(0) = 0$, $F(1) = 1$, and $F(n+2) = F(n+1) + F(n)$ for $n \geq 0$. To avoid the exponential complexity of the naive recursive algorithm, we apply the general technique of *memoization*. A Java `Fib` class with a `fib` method computing Fibonacci numbers with memoization is shown in Figure 8.22.

[4]From CeProMi collection of challenging examples, `http://www.lri.fr/cepromi`. This is only for illustration, since there exist other efficient ways to compute Fibonacci numbers.

```
theory HashableTheory<T> {
  ... // equality theory as in Figure 4
  logic integer hash{L}(T x);
  axiom hash_eq{L}: \forall T x,y;
    eq{L}(x,y) ==> hash{L}(x) == hash{L}(y);
}
```

Figure 8.23: Theory of hashable objects

8.4.1 Specification of the Fibonacci sequence

A mathematical definition of the Fibonacci sequence as a theory is given below.

```
theory Fibonacci {
  logic integer math_fib(integer n);
  axiom fib0: math_fib(0) == 0;
  axiom fib1: math_fib(1) == 1;
  axiom fibn: \forall integer n; n >= 2 ==>
       math_fib(n) == math_fib(n-1) + math_fib(n-2);
}
```

The expected behavior of the `fib` method is specified as follows.

```
/*@ requires n >= 0;
  @ assigns \nothing;
  @ ensures \result == math_fib(n);
  @*/
public long fib(int n);
```

Notice that issues related to arithmetic overflow are ignored. We just assume for simplicity that computations are made on unbounded integers.

8.4.2 Theories for hashable objects and hash maps

The first step is to define a theory which provides a predicate for testing equality, and a hash function. This theory is given in Figure 8.23. The dots are for the same axiomatization of the `eq` predicate as in Figure 8.16. The important part of this theory is the axiom `hash_eq` specifying the expected property for the hash function: two equal objects must have the same hash code.

Then, we provide a theory for maps, as shown in Figure 8.24. This theory is parameterized by both a type `K` for the keys, and a theory for equality and hashing of `K` objects. The type of data is not given as a parameter to the theory itself, but as a parameter `V` of the type of maps. This allows using the same theory of maps for several instances of `V`. This theory is indeed the classical *theory of arrays* which is a typical theory supported by SMT solvers. It is defined by a function `acc` to access the element indexed by some key, and a function `upd` which provides a so-called *functional update* of a map, returning a new map in which the element associated with some key is changed. The behavior of these two

```
theory Map<K><Th instantiating HashableTheory<K> > {
  type t<V>;
  logic <V> V acc{L}(t<V> m, K key);
  logic <V> t<V> upd{L}(t<V> m, K key, V value);

  axiom <V> acc_upd_eq{L}:
    \forall t<V> m, K key1 key2, V value;
      Th.eq{L}(key1,key2) ==>
        \at(acc(upd(m,key1,value),key2) == value,L);

  axiom <V> acc_upd_neq{L}:
    \forall t<V> m, K key1 key2, V value;
      ! Th.eq{L}(key1,key2) ==>
        \at(acc(upd(m,key1,value),key2) ==
          acc(m,key2),L);
}
```

Figure 8.24: Theory of maps

```
class HashMap<K,V>
    /*@ <Th instantiating HashableTheory<K> >
        constraint: K extends Object<K><Th> */
{
 //@ theory M = Map<K><Th>;
 //@ model M.t<V> m;

  /*@ requires x instanceof K;
    @ assigns \nothing;
    @ ensures \result != null ==>
    @         \result == M.acc(m,(K)x);
    @*/
  V get(Object x);

  /*@ requires k != null;
    @ assigns m;
    @ ensures m == M.upd(\old(m),k,v);
    @*/
  void put(K k, V v);
}
```

Figure 8.25: Specification of the `HashMap` class

functions is axiomatized by the two axioms in Figure 8.24, which makes use of the equality predicate on keys. It has to be noticed that specifying the proper equality relation on keys is one of the issues in this specification, and our proposal to use parameterized theories addresses this issue.

The resulting specification of the generic `java.util.HashMap` class is shown in Fig-

```
public class Object
    /*@ <T><H instantiating HashableTheory<T> > */ {

 /*@ requires this instanceof T && o instanceof T;
   @ ensures \result == true <==>
   @           H.eq((T)this,(T)o);
   @*/
 public boolean equals(Object o) { ... }

 /*@ requires this instanceof T;
   @ ensures \result == H.hash((T)this);
   @*/
 public int hashCode(){ ... }
}
```

Figure 8.26: Specification of two methods in the `Object` class

```
theory HashableInteger
          instantiates HashableTheory<Integer> {
  predicate eq{L}(Integer x, Integer y) =
    \at(x.value == y.value, L);
  logic integer hash{L}(Integer x) =
    \at(x.value, L);
}
```

Figure 8.27: Theory of equality and hashing of `Integer`s

ure 8.25. Since the type variable `K` of keys in `HashMap<K><V>` implicitly extends `Object`, it inherits the `equals` and `hashCode` methods defined in the `java.lang.Object` class. These two methods should be specified in the `Object` class with some theory instantiating `HashableTheory<K>`, as shown in Figure 8.26. A new issue arises here from dynamic dispatch: the instance of `Object` satisfying the `HashableTheory` is not known yet. Our proposal is to introduce another type parameter T in the specification, to be bound later. To ensure that the theory given as argument to `HashMap` class is an `HashableTheory` on the right type, a *constraint* is posed (Figure 8.25) on the type `K`. Notice also the use of local naming of a particular instance of a theory: the name `M` is given to the theory of `Map`s instantiated on the type of keys and on its theory of equality and hashing.

8.4.3 Instantiating generic hash maps

The generic `HashMap` class being specified, we can use it in the `Fib` class. The first step is to provide an instance of the theory of equality and hashing on `Integer`s. This is done in Figure 8.27. A proper implementation of the `Integer` class is then given in Figure 8.28.

In order to prove the `fib` postcondition, it is mandatory to provide a class invariant which, informally, states that for any pair (x, y) stored in the `memo` map, $y = F(x)$. The class invariant for the `Fib` class can be written as in Figure 8.29.

```
class Integer extends Object
        /*@ <Integer><HashableInteger> */ {

    boolean equals(Object o) {
        if (o instanceof Integer)
           return this.value == ((Integer)o).value;
        return false;
    }

    int hashCode() { return this.value; }
}
```

Figure 8.28: Implementation of hashable `Integer`s

```
class Fib {

  HashMap<Integer,Long> /*@ <HashableInteger> */ memo;

  /*@ invariant memo_fib: memo != null &&
    @   \forall Integer x, Long y;
    @   x != null && y == memo.M.acc(memo,x) &&
    @   y != null ==>
    @      y.value == math_fib(x.value);
    @*/

  ... // fib as specified in Section 4.1
}
```

Figure 8.29: Class invariant of the `Fib` class

8.4.4 Verification conditions for soundness

Verification conditions come first from the `instantiates` declaration of Figure 8.27. It should be proved that the given definitions satisfy the axioms given in `HashableTheory<T>` (Figure 8.23), when the type variable `T` is instantiated with `Integer`, which is straightforward. The constraint of Figure 8.25 comes immediately from the declaration of `Integer` class.

Other verification conditions come classically: invariant should be established by the `Fib()` constructor, which comes from a simple specification of `HashMap()` left to the reader; it should be preserved by any call to `fib()`, which comes from a simple reasoning by case distinction. The preconditions to calls to `get()` and `put()` are straightforward, and the postcondition of `fib()` follows from the invariant. However, one verification condition cannot be discharged, the one from the ***assigns*** clause: the contract says that there are no side-effects at all, whereas in reality the private `memo` hash map can be modified.

8.5 Summary

We have described the specification language KML for the Java programming language. The sorting algorithm by selection is proved by using a hybrid function which takes an array as a parameter and returns a bag. A bag is a collection without order. Given an array, this function returns the bag of its elements. We have expressed that the output array is a permutation of the input array by writing that the corresponding bags are the same. The works in [FM99] and [Mar09] suggest to define an inductive predicate to axiomatize the property that the output array is a permutation of the input array. Specifying with bags is another way to prove this property. But this new way of specifying a sorting algorithm leads to some difficulties. For instance, the `swap` method cannot be proved automatically without additional assertions and one lemma which itself is unprovable without induction. These assertions are required by provers to succeed their proofs.

Chapter 9

Conclusion and Perspectives

Contents

9.1 Conclusion

In this thesis we have addressed problems related to the development of tools used in the verification of software-based systems. We have been mostly interested in the design of decision procedures specified as inference systems. We have used a rewriting-based approach that allows building satisfiability procedures in a flexible way by using a general calculus for automated deduction, namely the paramodulation calculus. This calculus provides a decision procedure for the theory of interest if one can show that it terminates on every input composed of the finitely many axioms and any set of ground literals. Of course this process is error-prone. To reason on the derivations computed by paramodulation, the schematic paramodulation [LM02] calculus has been designed. It is an automated method of proving that the standard paramodulation calculus can be used to decide the satisfiability problem in the considered theories. In this thesis we have presented a rule-based system implementing a complete many-sorted schematic paramodulation calculus for non-unit theories. This environment, based on the theoretical studies in [LM02, LRRT11], is the first implementation of schematic paramodulation calculus. It has been implemented from scratch on the firm basis provided by Maude. Some automated deduction tools are already implemented in Maude, for instance a Church-Rosser checker [DM10a], a coherence checker [DM10b], etc. Our tool is a new contribution to this collection of tools. Thanks to this implementation and our experiments we have improved the schematic paramodulation calculus that takes now in a more satisfactory way the constants into account.

The authors of [NRR09c] has presented the paramodulation calculus for theories sharing Integer Offsets. Our environment has helped us to design a schematic calculus integrating the axioms of the Integer Offsets theory into a framework based on schematic

paramodulation. This calculus allows us to automatically decide the satisfiability problem for some theory sharing Integer Offsets. Since the theory of Integer Offsets allows us to build arithmetic expressions of the form $\mathsf{s}^n(t)$ for $n > 0$, we have proposed to schematize this exponent by the term of the form $\mathsf{s}^+(t)$. In this context, introducing the s^+ operator together with rewriting rules for terms containing s^+ fits well with automatic verification needs. Indeed, similar abstractions have been successfully used to verify cryptographic protocols with algebraic properties [BHK06], and to prove properties of Java Bytecode programs [BGJLR07]. Moreover, like in [BGJLR07], our schematization can be used for fine-tuning the precision of the analysis. We have presented the first extension of the notion of schematic paramodulation dedicated to a paramodulation calculus modulo a built-in theory. This study has led to new automatic proof techniques that are different from those performed manually in [NRR09c]. The assumptions we have used to apply our proof techniques are easy to satisfy for equational theories of practical interest.

The correctness of our tool is validated by checking the decidability of classical theories such as the theory of lists, the theory of records, the theory of possibly empty lists, the theory of array and the theory of recursive data structures. For all these theories paramodulation is known to terminate, and thanks to our tool we have proved decidability automatically. We have also automatically proved the termination of paramodulation modulo Integer Offsets for data structures equipped with counting operators, such as the theory of lists with length, the theory of lists with integer elements and the theory of records with increment.

Regarding the question of modular specification of generic Java classes and methods, we have proposed extensions to the Krakatoa Modeling Language, a part of the Why platform for proving that a Java or C program is a correct implementation of some specification. We have explained that in order to formally specify generic methods and classes, it is necessary to extend the notion of theory existing in specification languages like KML. It is not only mandatory to add type parametricity in theories, but also to provide a notion of parametricity of theories and a corresponding notion of theory instantiation. The extensions we have proposed are essentially inspired by existing notions of languages implementing higher-order logic, namely the notion of modules and functors as they exist for example in Coq, or the notion of type classes [SO08]. During this work the support of theories and its definition have been implemented within the Why platform. Since I stopped working on this question, our proposed extensions have not been integrated in Krakatoa implementation. However, the concept of theories and substitution/instantiation appeared in the logic language of Why3, and its notion of "cloning" of a theory.

9.2 Perspectives

The work carried out during the thesis opens several interesting research and development directions. All these directions are divided into three parts.

Short-term work. In Chapter 5 we have designed a new schematic paramodulation calculus that deals with non-disjoint extensions of Integer Offsets. This calculus provides automated proofs of termination and modular termination of decision procedures. Other

properties of schematic saturations can be checked, for instance deduction completeness and stably infiniteness which are key properties for the combination of decision procedures by the Nelson-Oppen method. This is a short term work that requires a bit of time.

More challenging issue is to design a general paramodulation calculus modulo Integer Offsets for *arbitrary* theories. It is still an open problem. When this problem is solved, we could study a related schematic calculus. These calculi would be very useful to study some extensions of the theory of arrays which plays a very important role in computer science and in verification.

Decision procedures for some extensions of the theory of arrays already exist (see, e.g. [BMS06, GNRZ07]) but our approach would additionally provide automated proofs of decidability. In this direction, we would have to find a less restrictive assumption to guarantee termination, possibly via a criterion involving the simplification ordering on terms extended to clauses.

When extending the schematic paramodulation calculus to the non-unit theories, the question of modular termination of arbitrary theories sharing Integer Offsets comes by itself. Other theories of counter arithmetic such as the theory of Increment can be considered in this framework [NRR10].

Mid-term work. Another interesting research direction is to consider theories that extend the theory of Abelian Groups (AG). The theory AG is clearly more expressive than the theory of Integer Offsets, and can provide more interesting examples. The paramodulation calculus with built-in Abelian groups has been developed in [GN04, NRR09a]. This calculus is more complicated than the standard one, since a semantic AG-unification is used instead of syntactic one. Following this direction, we are interested in developing a new schematic paramodulation calculus for AG that will allow us to automatically prove termination.

Also it will be interesting to study the problem of modular termination for theories sharing AG in a way similar to [RS11] for theories sharing counter arithmetic.

A non-disjoint combination method for theories modeling data structures with a counting operator and fragments of arithmetic has been proposed in [NRR09b]. At that point, a natural question arises: is it possible a non-disjoint combination method for data structures modulo AG plus some theories of arithmetic.

Long-term work. In [LM02], the authors use the schematic paramodulation to determine a bound on the number of clauses generated in a saturation by the paramodulation calculus. We also plan to apply our schematic paramodulation calculi to the complexity analysis.

Another research direction consists in studying other interesting fragments of arithmetic to consider as built-in theories in the paramodulation calculus. Moreover, since the schematic paramodulation calculus can be defined for particular cases, such as Integer Offsets, Abelian Groups, could we imagine a more general way to define schematic calculi?

Currently, the modular termination is considered for some specific shared theories. The modular termination for a more general setting for a class of shared theories is still

an open problem.

Another future work consists in integrating paramodulation-based procedures into SMT solvers, and moreover their non-disjoint combinations. Also, our decision procedures can be applied to Model-Checking Modulo Theories [GR10], a fully declarative and deductive symbolic model checker for safety properties of infinite state systems whose state variables are arrays.

Notice that all presented perspectives require further developments within our rule-based system.

Résumé étendu

Les ordinateurs font désormais partie de notre vie quotidienne. Qu'il s'agisse des transports, de la finance, de l'industrie, de l'aéronautique ou des appareils électroniques que nous utilisons au quotidien. Les pannes et les plantages peuvent engendrer des dommages matériels et financiers considérables, mais également des morts. Entre 1985 et 1987, un logiciel défectueux qui contrôlait la machine de radiothérapie Thérac-25 envoyait des doses bien trop importantes aux patients. Ce dysfonctionnement provoqua au moins 3 décès connus. En 1993, un plantage du logiciel de contrôle en vol d'un avion de chasse suédois engendra des problèmes de navigation qui conduisirent au crash de l'appareil. Un sous-traitant de la Nasa qui travaillait sur le projet Mars Orbiter a confondu le système métrique avec les unités de mesures anglo-saxonnes. La sonde s'est écrasée dès son arrivée sur Mars en 1999, réduisant à néant des années de travail et un projet qui avait couté plus de 327 millions de dollars au gouvernement américain. Certains plantages causent des problèmes d'importance secondaire. Mais pour d'autres, l'erreur n'est tout simplement pas envisageable. Ces systèmes doivent donc opérer de façon parfaitement fiable. Il existe plusieurs méthodes pour déterminer la fiabilité d'un logiciel.

De nombreux bugs peuvent être éliminés grâce aux tests. Mais ce genre de méthodes ne peut pas prouver que le système, l'algorithme ou le programme soit exempt d'erreurs ou de défauts, ou encore qu'il satisfasse une certaine propriété. Le nombre de situations possibles est si important que seul un petit nombre d'entre elles peut être testé. Une autre méthode consiste à procéder à une vérification formelle qui donne une preuve mathématique que le programme est correct et répond à une certaine propriété. Un programme vérifié de façon formelle fonctionnera correctement pour chaque donnée.

La vérification déductive constitue une approche de la vérification formelle. Durant la dernière décennie, des progrès considérables ont été enregistrés dans le domaine de la vérification des programmes. Désormais, les langages de programmation les plus populaires comme Java, C# et C ont leur propre langage de spécification formelle. Par exemple, Java Modelling Language [LKP07] a été créé pour les programmes Java. Spec# [BDF+05] a été créé pour C# et ACSL [BFM+09] a été créé pour C. Dans cette configuration, les fonctions et les méthodes ont des pré-conditions et des post-conditions inscrites dans le langage de spécification. Si la pré-condition est remplie et si la méthode se termine, alors elle établit la post-condition. Why [FM07] est une plateforme pour la vérification déductive des codes sources. Elle extrait les conditions de vérification d'un programme source annoté par les spécifications et les transmet aux prouveurs de théorèmes comme les solveurs SMT (Simplify, Z3, CVC3, Yices, Alt-Ergo, etc) ou les assistants de preuves (Coq, Isabelle/HOL, PVS, etc).

La Satisfiabilité Modulo une Théorie (SMT) consiste a décider de la satisfaisabilité d'une formule du premier ordre par rapport à une théorie. Une procédure de satisfaisabilité est une procédure de décision pour le problème de satisfaisabilité, c'est-à-dire un algorithme qui se termine toujours par une réponse "oui" si la formule est satisfaite ou par la réponse "non" dans le cas contraire. Dans le contexte de cette thèse, lorsque nous considérons une procédure de décision, nous considérons en réalité une procédure de satisfaisabilité. Nous nous concentrerons sur les procédures de décision spécifiées comme des systèmes d'inférence, qui sont des ensembles de règles d'inférence. Une règle d'inférence relie des prémisses à une conclusion. Par exemple, si $u = c$ et $u' = c$, alors nous pouvons conclure que $u = u'$. La conception et l'implémentation de ces procédures de décision demeurent des tâches extrêmement complexes. Pour faire gagner du temps au chercheur, une approche importante fondée sur la réécriture a été étudiée durant la dernière décennie [ARR01, ARR03, ABRS09]. Cette approche permet d'élaborer des procédures de satisfaisabilité de façon flexible en utilisant un calcul général pour le raisonnement équationnel, nommé Paramodulation ($\mathcal{PC}$) [NR01] (appelé aussi superposition). En général, une application exhaustive des règles de $\mathcal{PC}$ conduit à une procédure de semi-décision qui s'arrête si les données sont contradictoires (une clause vide est alors générée). Cependant, elle peut diverger sur des données non contradictoires. Heureusement, elles peuvent aussi se terminer pour certaines théories, et de ce fait devenir une procédure de décision. Dans [ARR01, ARR03, ABRS09], il est montré de quelle manière un système d'inférence basé sur la paramodulation peut être utilisé comme une procédure de décision pour différentes théories, notamment des listes, des tableaux, des enregistrements ou de leurs combinaisons.

Par souci d'efficacité, il est utile d'avoir des axiomes intégrés dans les calculs et de pouvoir créer des calculs de paramodulation modulo une theorie. Cela est particulièrement important pour les fragments arithmétiques, en raison de l'ubiquité de l'arithmétique dans l'application des méthodes formelles. Par exemple, le calcul de paramodulation standard a été étendu dans [NRR09c] de façon à prendre en compte les axiomes de la théorie des *Integer Offsets*, qui est une forme d'arithmétique de comptage. De nouvelles méthodes de combinaison à la Nelson-Oppen ont été développées pour considérer l'union de ces théories qui partagent des fragments d'arithmétique. Cela ouvre la voie à l'utilisation de méthodes de combinaison de théories non-disjointes au sein de solveurs SMT.

Pour raisonner sur le calcul de paramodulation standard, un calcul de paramodulation schématique ($\mathcal{SPC}$) [LM02] a été étudié pour prouver automatiquement la terminaison. L'avantage de $\mathcal{SPC}$ est que s'il s'arrête pour une donnée abstraite, alors $\mathcal{PC}$ s'arrête pour toutes les données concrètes correspondantes. Plus généralement, $\mathcal{SPC}$ est un outil automatique qui vérifie les propriétés de $\mathcal{PC}$ comme la terminaison, la *stable infinité* et la complétude vis-à-vis de la déduction. Des améliorations de la première présentation de $\mathcal{SPC}$ ont été proposées dans [LT07, LRRT11]. Les auteurs de ces articles ont étudié la décidabilité automatique mais également la combinabilité automatique.

Jusqu'à présent, aucune schématisation de la paramodulation modulo la théorie des *Integer Offsets* n'a été créé. De ce fait, il existe un réel besoin pour une méthode capable de prouver qu'une théorie donnée admet une procédure de décision basée sur la paramodulation modulo les *Integer Offsets*.

Les contributions de cette thèse concernent des résultats à la fois pratiques et théoriques:

1. La contribution principale de cette thèse est un système de règles qui met en œuvre un calcul de la paramodulation schématique complet et multi-sorté pour des théories arbitraires. Cet outil nous permet de vérifier automatiquement si le calcul de la paramodulation termine pour les théories définies par des clauses arbitraires et si les procédures de décision qui y sont associées sont combinables. L'outil peut aussi être utilisé pour vérifier la terminaison modulaire quand on considère une combinaison de théories à signatures disjointes ou des théories dont les signatures ne sont pas disjointes. De plus, cette mise en œuvre de paramodulation schématique fournit une trace pour chacune des règles appliquées, ce qui est très utile pour valider ou invalider les preuves de saturation décrites précédemment dans la littérature. La validité de cet outil est démontrée en vérifiant la décidabilité de théories classiques comme la théorie des listes (avec ou sans extensionnalité), la théorie des enregistrements sans extensionnalité, la théorie des listes possiblement vides, la théorie des tableaux et la théorie des structures de données récursives pour lesquels on sait que la paramodulation termine [ABRS09, NRR09c, LRRT11].

2. Le calcul de la paramodulation schématique a été amélioré grâce à notre expérimentation. En fait, le calcul de la paramodulation schématique proposé dans [LRRT11] ne prend pas en compte les constantes dans la signature de la théorie. Par exemple, la signature de la théorie des listes possiblement vides contient une constante "nil". Le calcul de la paramodulation schématique présenté dans cette thèse prend en compte ces constantes.

3. Un nouveau calcul de la paramodulation schématique pour décrire les saturations modulo les Integer Offsets est présenté dans cette thèse. Ce calcul est validé grâce à notre outil. Notre approche requiert une nouvelle forme de schématisation de façon à prendre en compte les expressions arithmétiques. L'intérêt de la paramodulation schématique réside dans la correspondance entre une dérivation utilisant la paramodulation concrète et une dérivation utilisant la paramodulation schématique : de façon simple, l'ensemble de dérivations obtenu grâce à la paramodulation schématique vient sur-approximer l'ensemble de dérivations obtenu grâce à la paramodulation concrète. Nous montrons sous quelles conditions la terminaison de la paramodulation schématique implique la terminaison de la paramodulation concrète. A nouveau, le fait de considérer les Integer Offsets requiert des arguments de preuve spécifiques. Grâce à ce calcul schématique, nous pouvons automatiquement vérifier si la paramodulation modulo les Integer Offsets est une procédure de décision pour l'union de deux théories non disjointes partageant la théorie des Integer Offsets.

4. Au début de cette thèse, nous avons participé à une étude sur la spécification modulaire de classes et de méthodes Java génériques, sous la supervision de Claude Marché, Alain Giorgetti et Olga Kouchnarenko. Nous avons proposé des extensions du Krakatoa Modelling Language, une partie de la plateforme Why pour prouver qu'un programme Java ou C est une implémentation correcte de certaines spécifications. Les nouvelles constructions que nous proposons pour la spécification des

programmes Java génériques sont présentées à travers deux exemples particulièrement significatifs: la spécification de la méthode générique pour trier des tableaux provenant de la classe `java.util.Arrays` dans l'API Java, et la spécification de la classe `java.util.Hashmap` définissant une **hash map** générique. Les principales caractéristiques sont l'introduction de la paramétricité à la fois pour les types et pour les théories, ainsi qu'une relation d'instantiation entre les théories. Nous discuterons des conditions de correction et de leur vérification.

Dans cette thèse, nous nous sommes intéressés à des problèmes liés au développement d'outils utilisés dans la vérification des systèmes basés sur des logiciels. Nous nous sommes tout particulièrement intéressés à la création de procédures de décision spécifiées comme des systèmes d'inférence. Nous avons utilisé une approche fondée sur la réécriture qui permet de construire des procédures de satisfaisabilité de façon flexible, en utilisant un calcul général pour le raisonnement équationnel, nommé paramodulation. Ce calcul fournit une procédure de décision pour la théorie étudiée si l'on peut démontrer qu'il se termine pour chaque donnée composée d'un ensemble fini d'axiomes et d'un ensemble de littéraux plats. Le calcul de paramodulation schématique [LM02] a été créé pour raisonner sur les dérivations générées par la paramodulation. Il s'agit d'une méthode qui prouve automatiquement que le calcul de la paramodulation standard peut être utilisé pour décider les problèmes de satisfaisabilité dans les théories étudiées.

Dans cette thèse, nous avons présenté un système fondé sur des règles qui met en œuvre un calcul de la paramodulation schématique complet et ordonné pour les théories non-unitaires. Cet environnement, basé sur les études théoriques de [LM02, LRRT11], constitue la première mise en œuvre du calcul de paramodulation schématique. Il a été créé *from scratch*, sur les bases fournies par Maude. Certains outils de déduction automatique sont déjà mis en œuvre dans Maude, par exemple un vérificateur de la propriété de Church-Rosser [DM10a], un vérificateur de cohérence [DM10b], etc. Notre outil offre une contribution supplémentaire à la panoplie existante. Grâce à cette mise en œuvre et à nos expérimentations, nous avons grandement amélioré le calcul de la paramodulation schématique qui prend désormais en compte les constantes de façon bien plus satisfaisante.

Les auteurs de [NRR09c] ont présenté un calcul de paramodulation pour les extensions de la théorie des Integer Offsets. Notre environnement nous a aidé à créer un calcul schématique intégrant les axiomes de la théorie des Integer Offsets dans un cadre basé sur la paramodulation schématique. Ce calcul nous permet de décider automatiquement le problème de satisfaisabilité de certaines extensions de la théorie des Integer Offsets. Dans la mesure ou la théorie des Integer Offsets nous permet de construire des expressions arithmétiques de la forme $s^(t)$ pour $n > 0$, nous avons proposé de schématiser cet exposant sous la forme $s^+(t)$. Dans ce contexte, introduire l'opérateur s^+ avec des règles de réécriture pour les termes contenant s^+ s'accorde bien avec les besoins de la vérification automatique. En effet, des abstractions similaires avaient déjà été utilisées avec succès pour vérifier des protocoles cryptograhiques avec des propriétés algébriques [BHK06] et pour prouver des propriétés de programmes Java Bytecode [BGJLR07]. De plus, comme dans [BGJLR07], notre schématisation peut être utilisée pour améliorer la précision de l'analyse. Nous avons présenté la première extension de la notion de paramodulation sché-

matique dédiée à un calcul de paramodulation modulo une théorie. Cette étude a conduit a de nouvelles techniques de preuves automatiques qui sont différentes de celles opérées manuellement dans [NRR09c]. Les hypothèses que nous avons utilisées pour appliquer nos techniques de preuve sont faciles à satisfaire pour des théories équationnelles d'intérêt pratique.

La validité de notre instrument est attesté par les vérifications de décidabilité de théories classiques comme la théorie des listes, la théorie des enregistrements, la théorie des listes possiblement vides, la théorie des tableaux ou encore la théorie des structures de données récursives. Pour toutes ces théories, la paramodulation a la bonne propriété de terminer et grâce à notre outil nous l'avons prouvé de façon automatique. Nous avons également prouvé la terminaison de la paramodulation modulo les Integer Offsets pour les structures de données équipées avec des opérateurs de comptage, comme la théorie des listes avec longueur, la théorie des listes avec éléments entiers et la théorie des enregistrements avec incrément.

Pour la question de la spécification modulaire des classes et des méthodes Java génériques, nous avons proposé l'extension du Krakatoa Modelling Language, une partie de la plateforme Why, pour prouver qu'un programme Java ou C est une implémentation correcte de certaines spécifications. Nous avons montré que pour spécifier formellement les méthodes et les classes génériques, il est nécessaire d'étendre la notion de théorie existant dans les langages de spécification comme KML. Il est non seulement obligatoire d'ajouter une paramétricité du type dans les théories, mais il faut également fournir une notion de paramétricité des théories et une notion correspondante de l'instantiation de la théorie. Les extensions que nous avons proposées sont essentiellement inspirées de notions existantes, à savoir la notion de modules et de foncteurs tels qu'ils existent par exemple dans Coq, ou encore les notions de *type classes* [SO08]. Depuis que j'ai arrêté de travailler sur cette question, les extensions que nous avions proposées n'ont pas été intégrées dans la mise en œuvre de Krakatoa. Néanmoins, le concept des théories et de la substitution/instantiation apparait dans la logique du language de Why3, et sa notion de "clonage" d'une théorie.

Le travail effectué dans cette thèse ouvre plusieurs directions de recherche et de développement particulièrement intéressantes. Ces directions se divisent en trois parties :

- Travail à court terme.

  Nous avons créé un nouveau calcul de paramodulation schématique qui traite des extensions non-disjointes des Integer Offsets. Ce calcul fournit des preuves automatisées de terminaison et de terminaison modulaire des procédures de décision. D'autres propriétés peuvent être vérifiées par saturation schématique , par exemple la complétude vis-à-vis de la déduction ou la *stable infinité* qui sont des propriétés clés pour la combinaison de procédures de décision par la méthode de Nelson-Oppen. Il s'agit d'un travail sur le court terme qui ne nécessite que peu de temps.

  Un objectif plus complexe consiste à créer un calcul général de paramodulation modulo les Integer Offsets pour des théories arbitraires. Cela reste un problème ouvert. Quand le problème sera réglé, nous pourrions étudier un calcul schématique correspondant. Ces calculs seraient très utiles pour étudier quelques extensions de

la théorie des tableaux qui une structure de données importante, notamment en vérification pour représenter la mémoire.

Des procédures de décision pour certaines extensions de la théorie des tableaux existent déjà (voir e.g. [BMS06, GNRZ07]), mais notre approche va fournir de nouvelles preuves automatiques de décidabilité. A cette fin, nous devrions trouver des hypothèses moins restrictives pour garantir la termination, possiblement à travers un critère impliquant un ordre de simplification sur les termes étendus aux clauses.

En étendant le calcul de la paramodulation schématique aux théories non-unitaires, la question de la terminaison modulaire des théories arbitraires partageant les Integer Offsets se pose d'elle-même. D'autres théories arithmétiques, comme la théorie de l'Incrément, peuvent également être considérées dans ce cadre [NRR10].

- Travail à moyen terme.

  Une autre direction de recherche particulièrement intéressante est de considérer les théories qui étendent la théorie des Groupes Abéliens (AG). La théorie AG est clairement plus expressive que la théorie des Integer Offsets, et elle peut fournir des exemples nettement plus intéressants. Un calcul de paramodulation modulo AG a été utilisé dans [GN04, NRR09a]. Ce calcul est plus compliqué que le calcul standard, dans la mesure ou une unificatio modulo AG est utilisée au lieu de l'unification syntaxique. En suivant cette direction, nous aimerions développer un nouveau calcul de paramodulation schématique pour AG qui nous permettrait de prouver automatiquement la terminaison.

  Il serait également intéressant d'étudier le problème de la terminaison modulaire pour les théories partageant AG de la même façon que dans [RS11] pour les théories partageant l'arithmétique de comptage.

  Une méthode de combinaison non-disjointe pour les théories modélisant les structures de données avec un opérateur de comptage et des fragments d'arithmétique a été proposée dans [NRR09b]. A ce stade, une question se pose alors tout naturellement: Est-il possible d'avoir une méthode de combinaison non-disjointe pour les structures de données modulo AG avec quelques théories d'arithmétique ?

- Travail à long terme.

  Dans [LM02], les auteurs utilisent la paramodulation schématique pour déterminer le nombre de clauses générées dans une saturation par le calcul de la paramodulation. Nous comptons également appliquer notre calcul de la paramodulation schématique à l'analyse de complexité.

  Une autre direction de recherche consiste à intégrer d'autres fragments intéressants d'arithmétique dans le calcul de paramodulation. De plus, dans la mesure ou des calculs de paramodulation schématique peuvent être définis pour des cas particuliers comme les Integer Offsets ou les Groupes Abéliens, peut-être pourrions-nous imaginer une méthode plus générale pour définir les calculs schématiques ?

  A l'heure actuelle, la terminaison modulaire est utilisée pour certaines théories spécifiques partagées. La terminaison modulaire pour une classe de théories partagées

est encore un problème ouvert.

Un autre travail futur consistera à intégrer les procédures basées sur la paramodulation dans les solveurs SMT ainsi que leurs combinaisons non-disjointes. Ainsi, nos procédures de décision peuvent être utilisées dans le cadre du *Model Checking Modulo Theories* [GR10], un vérificateur symbolique de modèle entièrement déclaratif et déductif pour les propriétés de sûreté des systèmes d'états infinis dont les variables sont des tableaux.

Veuillez remarquer que toutes les perspectives présentées ici demandent davantage de développements de notre système basée sur les règles.

Bibliography

[ABK+01] E. Astesiano, M. Bidoit, H. Kirchner, B. Krieg-Bruckner, P.D. Mosses, D. Sannella, and A. Tarlecki. Casl: The common algebraic specification language, 2001.

[Abr05] J.-R. Abrial. *The B-book - assigning programs to meanings.* Cambridge University Press, 2005.

[ABRS05] A. Armando, M.P. Bonacina, S. Ranise, and S. Schulz. On a rewriting approach to satisfiability procedures: Extension, combination of theories and an experimental appraisal. In B. Gramlich, editor, *Proc. of the 5th Int. Workshop on Frontiers of Combining Systems (FroCoS'2005)*, volume 3717 of *LNCS*, pages 65–80, Vienna, Austria, 2005. Springer.

[ABRS09] A. Armando, M. P. Bonacina, S. Ranise, and S. Schulz. New results on rewrite-based satisfiability procedures. *ACM Trans. Comput. Logic*, 10(1):1 – 51, 2009.

[ARR01] Alessandro Armando, Silvio Ranise, and Michaël Rusinowitch. Uniform derivation of decision procedures by superposition. In *Proceedings of the 15th International Workshop on Computer Science Logic*, CSL '01, pages 513–527, London, UK, UK, 2001. Springer-Verlag.

[ARR03] A. Armando, S. Ranise, and M. Rusinowitch. A rewriting approach to satisfiability procedures. *J. Inf. Comput*, 183(2):140 – 164, 2003.

[BCC+03] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G.T. Leavens, K.R.M. Leino, and E. Poll. An Overview of JML Tools and Applications. In *FMICS 03*, volume 80 of *ENTCS*, pages 73–89. Elsevier, 2003.

[BDF+05] Michael Barnett, Robert DeLine, Manuel Fähndrich, Bart Jacobs, K. Rustan M. Leino, Wolfram Schulte, and Herman Venter. The spec# programming system: Challenges and directions. In Bertrand Meyer and Jim Woodcock, editors, *VSTTE*, volume 4171 of *Lecture Notes in Computer Science*, pages 144–152. Springer, 2005.

[BFM+09] P. Baudin, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto. *ACSL: ANSI/ISO C Specification Language, version 1.4*, 2009. `http://frama-c.cea.fr/acsl.html`.

[BG80] R. M. Burstall and J. A. Goguen. The semantics of clear, a specification language. In *Proceedings of the Abstract Software Specifications, 1979 Copenhagen Winter School*, pages 292–332, London, UK, UK, 1980. Springer-Verlag.

[BG90] Leo Bachmair and Harald Ganzinger. On restrictions of ordered paramodulation with simplification. In Mark E. Stickel, editor, *CADE*, volume 449 of *Lecture Notes in Computer Science*, pages 427–441. Springer, 1990.

[BGJLR07] Y. Boichut, T. Genet, T. P. Jensen, and L. Le Roux. Rewriting approximations for fast prototyping of static analyzers. In F. Baader, editor, *RTA*, volume 4533 of *LNCS*, pages 48–62. Springer, 2007.

[BGLS95] L. Bachmair, H. Ganzinger, C. Lynch, and W. Snyder. Basic paramodulation. *Inf. Comput.*, 121(2):172–192, 1995.

[BHK06] Y. Boichut, P.-C. Héam, and O. Kouchnarenko. Handling algebraic properties in automatic analysis of security protocols. In K. Barkaoui, A. Cavalcanti, and A. Cerone, editors, *ICTAC*, volume 4281 of *LNCS*, pages 153–167. Springer, 2006.

[BMS06] A. R. Bradley, Z. Manna, and H. B. Sipma. What's decidable about arrays? In E. A. Emerson and K. S. Namjoshi, editors, *VMCAI*, volume 3855 of *LNCS*, pages 427–442. Springer, 2006.

[BN98] F. Baader and T. Nipkow. *Term rewriting and all that.* Cambridge University Press, New York, NY, USA, 1998.

[Bra75] D. Brand. Proving theorems with the modification method. *SIAM J. Comput.*, 4(4):412–430, 1975.

[CDE+03] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. The Maude 2.0 system. In R. Nieuwenhuis, editor, *RTA*, volume 2706 of *LNCS*, pages 76–87. Springer, 2003.

[CDE+09] M. Clavel, F. Durán, S. Eker, S. Escobar, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott. Unification and narrowing in Maude 2.4. In R. Treinen, editor, *RTA*, volume 5595 of *LNCS*, pages 380–390. Springer, 2009.

[CM96a] M. Clavel and J. Meseguer. Axiomatizing reflective logics and languages. In *Proceedings of Reflection'96*, pages 263–288, 1996.

[CM96b] M. Clavel and J. Meseguer. Reflection and strategies in rewriting logic. *Electr. Notes Theor. Comput. Sci.*, 4:126–148, 1996.

[CMP07] M. Clavel, J. Meseguer, and M. Palomino. Reflection in membership equational logic, many-sorted equational logic, horn logic with equality, and rewriting logic. *Theor. Comput. Sci.*, 373(1-2):70–91, 2007.

[Der82] N. Dershowitz. Orderings for term-rewriting systems. *Theor. Comput. Sci.*, 17:279–301, 1982.

[DGS93] R. Diaconescu, J. Goguen, and P. Stefaneas. Logical support for modularisation. In *Papers presented at the second annual Workshop on Logical environments*, pages 83–130, New York, NY, USA, 1993. Cambridge University Press.

[DJ90] N. Dershowitz and J.-P. Jouannaud. Rewrite Systems. In *Handbook of Theoretical Computer Science*, volume B, pages 243–320. Elsevier and MIT Press, 1990.

[DM10a] F. Durán and J. Meseguer. A Church-Rosser checker tool for conditional order-sorted equational Maude specifications. In P.C. Ölveczky, editor, *Proc. of 8th Int. Workshop on Rewriting Logic and Its Applications (WRLA'10)*, volume 6381 of *LNCS*, pages 69–85, Paphos, Cyprus, March 2010. Springer.

[DM10b] F. Durán and J. Meseguer. A Maude coherence checker tool for conditional order-sorted rewrite theories. In P.C. Ölveczky, editor, *Proc. of 8th Int. Workshop on Rewriting Logic and Its Applications (WRLA'10)*, volume 6381 of *LNCS*, pages 86–103, Paphos, Cyprus, March 2010. Springer.

[EMOMV07] St. Eker, N. Martí-Oliet, J. Meseguer, and A. Verdejo. Deduction, strategies, and rewriting. *Electr. Notes Theor. Comput. Sci.*, 174(11):3–25, 2007.

[FM99] J.-C. Filliâtre and N. Magaud. Certification of sorting algorithms in the Coq system. In *Theorem Proving in Higher Order Logics: Emerging Trends*, 1999.

[FM07] J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In *CAV'07*, volume 4590 of *Lecture Notes in Computer Science*, pages 173–177. Springer, July 2007.

[GN04] G. Godoy and R. Nieuwenhuis. Superposition with completely built-in abelian groups. *Journal of Symbolic Computation*, 37(1):1–33, 2004.

[GNRZ07] S. Ghilardi, E. Nicolini, S. Ranise, and D. Zucchelli. Decision procedures for extensions of the theory of arrays. *Ann. Math. Artif. Intell.*, 50(3-4):231–254, 2007.

[GR10] S. Ghilardi and S. Ranise. Mcmt: A model checker modulo theories. In J. Giesl and R. Hähnle, editors, *IJCAR*, volume 6173 of *Lecture Notes in Computer Science*, pages 22–29. Springer, 2010.

[HR91] J. Hsiang and M. Rusinowitch. Proving refutational completeness of theorem-proving strategies: the transfinite semantic tree method. *J. ACM*, 38(3):558–586, July 1991.

[Kap92] D. Kapur, editor. *Automated Deduction - CADE-11, 11th International Conference on Automated Deduction, Saratoga Springs, NY, USA, June 15-18, 1992, Proceedings*, volume 607 of *Lecture Notes in Computer Science*. Springer, 1992.

[KK06] C Kirchner and H. Kirchner. *Rewriting Solving Proving*. LORIA, INRIA and CNRS, 2006.

[KL80] S. Kamin and J.-J. Lévy. Attempts for generalizing the recursive path ordering. Unpublished manuscript, 1980.

[LC06] G. T. Leavens and Y. Cheon. Design by Contract with JML. Available from `http://www.jmlspecs.org`, 2006.

[LKP07] Gary T. Leavens, Joseph R. Kiniry, and Erik Poll. A jml tutorial: Modular specification and verification of functional behavior for java. In Werner Damm and Holger Hermanns, editors, *CAV*, volume 4590 of *Lecture Notes in Computer Science*, page 37. Springer, 2007.

[LM02] C. Lynch and B. Morawska. Automatic decidability. In *LICS*, pages 7–16, Copenhagen, Denmark, July 2002. IEEE Computer Society.

[LP09] K.R.M. Leino and Müller P. Using the spec# language, methodology, and tools to write bug-free programs, 2009.

[LRRT11] C. Lynch, S. Ranise, C. Ringeissen, and D.-K. Tran. Automatic decidability and combinability. *J. Inf. Comput*, 209(7):1026–1047, 2011.

[LT07] C. Lynch and D.K. Tran. Automatic decidability and combinability revisited. In Frank Pfenning, editor, *CADE*, volume 4603 of *Lecture Notes in Computer Science*, pages 328–344. Springer, 2007.

[Mar07] C. Marché. Towards modular algebraic specifications for pointer programs: a case study. In *Rewriting, Computation and Proof*, volume 4600 of *Lecture Notes in Computer Science*, pages 235–258. Springer, 2007.

[Mar09] C. Marché. The Krakatoa tool for deductive verification of Java programs. Winter School on Object-Oriented Verification, Viinistu, Estonia, January 2009. `http://krakatoa.lri.fr/ws/`.

[MOMV05] N. Martí-Oliet, J. Meseguer, and A. Verdejo. Towards a strategy language for Maude. *Electr. Notes Theor. Comput. Sci.*, 117:417–441, January 2005.

[MZ02] Z. Manna and C.G. Zarba. Combining decision procedures. In B.K. Aichernig and T. S. E. Maibaum, editors, *10th Anniversary Colloquium of UNU/IIST*, volume 2757 of *Lecture Notes in Computer Science*, pages 381–422. Springer, 2002.

[NO79] N. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1:245–257, 1979.

[NR92] R Nieuwenhuis and A. Rubio. Theorem proving with ordering constrained clauses. In Kapur [Kap92], pages 477–491.

[NR01] R. Nieuwenhuis and A. Rubio. Paramodulation-based theorem proving. In J. A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, pages 371–443. Elsevier and MIT Press, 2001.

[NRR09a] E. Nicolini, C. Ringeissen, and M. Rusinowitch. Combinable extensions of Abelian groups. In R. Schmidt, editor, *CADE*, volume 5663 of *LNCS*, pages 51–66. Springer, 2009.

[NRR09b] E. Nicolini, C. Ringeissen, and M. Rusinowitch. Data structures with arithmetic constraints: A non-disjoint combination. In S. Ghilardi and R. Sebastiani, editors, *FroCoS*, volume 5749 of *Lecture Notes in Computer Science*, pages 319–334. Springer, 2009.

[NRR09c] E. Nicolini, C. Ringeissen, and M. Rusinowitch. Satisfiability procedures for combination of theories sharing integer offsets. In S. Kowalewski and A. Philippou, editors, *TACAS*, volume 5505 of *LNCS*, pages 428–442. Springer, 2009.

[NRR10] E. Nicolini, C. Ringeissen, and M. Rusinowitch. Combining satisfiability procedures for unions of theories with a shared counting operator. *Fundam. Inform.*, 105(1-2):163–187, 2010.

[Pet83] G.E. Peterson. A technique for establishing completeness results in theorem proving with equality. *SIAM J. Comput.*, 12(1):82–100, 1983.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965.

[RS11] C. Ringeissen and V. Senni. Modular termination and combinability for superposition modulo counter arithmetic. In C. Tinelli and V. Sofronie-Stokkermans, editors, *FroCoS*, volume 6989 of *Lecture Notes in Computer Science*, pages 211–226. Springer, 2011.

[Rus91] M. Rusinowitch. Theorem-proving with resolution and superposition. *J. Symb. Comput.*, 11(1/2):21–49, 1991.

[RV01] A. Riazanov and A. Voronkov. Splitting without backtracking. In B. Nebel, editor, *IJCAI*, pages 611–617. Morgan Kaufmann, 2001.

[RW69] G. Robinson and L. Wos. Paramodulation and theorem proving in first-order theories with equality. *Machine Intelligence*, (4):135–160, 1969.

[SO08] M. Sozeau and N. Oury. First-Class Type Classes. In Sofiène Tahar, Otmame Ait-Mohamed, and César Muñoz, editors, *21th International Conference on Theorem Proving in Higher Order Logics*, Lecture Notes in Computer Science. Springer, August 2008.

[Spi92] J.M. Spivey. *Z Notation - a reference manual (2. ed.)*. Prentice Hall International Series in Computer Science. Prentice Hall, 1992.

[TGRK12] E. Tushkanova, A. Giorgetti, C. Ringeissen, and O. Kouchnarenko. A rule-based framework for building superposition-based decision procedures. In F. Durán, editor, *WRLA*, volume 7571 of *LNCS*, pages 221–239. Springer, 2012.

[The06] The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.1*, July 2006. `http://coq.inria.fr`.

[Tra07] D.-K. Tran. *Conception de Procédure de Décision par Combinaison et Saturation.* PhD thesis, LORIA – Université Henri Poincaré, Nancy, France, 2007.

[TRRK10] D.-K Tran, C. Ringeissen, S. Ranise, and H. Kirchner. Combination of convex theories: Modularity, deduction completeness, and explanation. *J. Symb. Comput.*, 45(2):261–286, 2010.

Résumé

Dans cette thèse, on étudie des problèmes liés à la vérification de systèmes (logiciels). On s'intéresse plus particulièrement à la conception sûre de procédures de décision utilisées en vérification. De plus, on considère également un problème de modularité pour un langage de modélisation utilisé dans la plateforme de vérification Why.

De nombreux problèmes de vérification peuvent se réduire à un problème de satisfaisabilité modulo des théories (SMT). Pour construire des procédures de satisfaisabilité, Armando et *al.* ont proposé en 2001 une approche basée sur la réécriture. Cette approche utilise un calcul général pour le raisonnement équationnel appelé paramodulation. En général, une application équitable et exhaustive des règles du calcul de paramodulation (PC) conduit à une procédure de semi-décision qui termine sur les entrées insatisfaisables (la clause vide est alors engendrée), mais qui peut diverger sur les entrées satisfaisables. Mais ce calcul peut aussi terminer pour des théories intéressantes en vérification, et devient ainsi une procédure de décision. Pour raisonner sur ce calcul, un calcul de paramodulation schématique (SPC) a été étudié, en particulier pour prouver automatiquement la décidabilité de théories particulières et de leurs combinaisons. L'avantage de ce calcul SPC est que s'il termine sur une seule entrée abstraite, alors PC termine pour toutes les entrées concrètes correspondantes. Plus généralement, SPC est un outil automatique pour vérifier des propriétés de PC telles que la terminaison, la stable infinité et la complétude de déduction.

Une contribution majeure de cette thèse est un environnement de prototypage pour la conception et la vérification de procédures de décision. Cet environnement, basé sur des fondements théoriques, est la première implantation du calcul de paramodulation schématique. Il a été complètement implanté sur la base solide fournie par le système Maude mettant en œuvre la logique de réécriture. Nous montrons que ce prototype est très utile pour dériver la décidabilité et la combinabilité de théories intéressantes en pratique pour la vérification.

Cet environnement est appliqué à la conception d'un calcul de paramodulation schématique dédié à une arithmétique de comptage. Cette contribution est la première extension de la notion de paramodulation schématique à une théorie prédéfinie. Cette étude a conduit à de nouvelles techniques de preuve automatique qui sont différentes de celles utilisées manuellement dans la littérature. Les hypothèses permettant d'appliquer nos techniques de preuves sont faciles à satisfaire pour les théories équationnelles avec opérateurs de comptage. Nous illustrons notre contribution théorique sur des théories représentant des extensions de structures de données classiques comme les listes ou les enregistrements.

Nous avons également contribué au problème de la spécification modulaire pour les classes et méthodes Java génériques. Nous proposons des extensions du language de modélisation Krakatoa, faisant partie de la plateforme Why qui permet de prouver qu'un programme C ou Java est correct par rapport à sa spécification. Les caractéristiques essentielles de notre apport sont l'introduction de la paramétricité à la fois pour les types et les théories, ainsi qu'une relation d'instantiation entre les théories. Les extensions proposées sont illustrées sur deux exemples significatifs: tri de tableaux et fonctions de hachage.

Les deux problèmes traités dans cette thèse ont pour point commun les solveurs SMT. Les procédures de décision sont les moteurs des solveurs SMT, alors que la plateforme Why engendre des conditions de vérification dérivées d'une programme source annoté, et les transmet aux solveurs SMT (ou assistants de preuve) pour vérfier la correction du programme.

Mots-clés: Procédures de décision, Paramodulation, Saturation Schématique, Combinaison

Abstract

In this thesis we address problems related to the verification of software-based systems. We are mostly interested in the (safe) design of decision procedures used in verification. In addition, we also consider a modularity problem for a modeling language used in the Why verification platform.

Many verification problems can be reduced to a satisfiability problem modulo theories (SMT). In order to build satisfiability procedures Armando et *al.* have proposed in 2001 an approach based on rewriting. This approach uses a general calculus for equational reasoning named paramodulation. In general, a fair and exhaustive application of the rules of paramodulation calculus (PC) leads to a semi-decision procedure that halts on unsatisfiable inputs (the empty clause is then generated) but may diverge on satisfiable ones. Fortunately, it may also terminate for some theories of interest in verification, and thus it becomes a decision procedure. To reason on the paramodulation calculus, a schematic paramodulation calculus (SPC) has been studied, notably to automatically prove decidability of single theories and of their combinations. The advantage of SPC is that if it halts for one given abstract input, then PC halts for all the corresponding concrete inputs. More generally, SPC is an automated tool to check properties of PC like termination, stable infiniteness and deduction completeness.

A major contribution of this thesis is a prototyping environment for designing and verifying decision procedures. This environment, based on the theoretical studies, is the first implementation of the schematic paramodulation calculus. It has been implemented from scratch on the firm basis provided by the Maude system based on rewriting logic. We show that this prototype is very useful to derive decidability and combinability of theories of practical interest in verification. It helps testing new saturation strategies and experimenting new extensions of the original (schematic) paramodulation calculus.

This environment has been applied for the design of a schematic paramodulation calculus dedicated to the theory of Integer Offsets. This contribution is the first extension of the notion of schematic paramodulation to a built-in theory. This study has led to new automatic proof techniques that are different from those performed manually in the literature. The assumptions to apply our proof techniques are easy to satisfy for equational theories with counting operators. We illustrate our theoretical contribution on theories representing extensions of classical data structures such as lists and records.

We have also addressed the problem of modular specification of generic Java classes and methods. We propose extensions to the Krakatoa Modeling Language, a part of the Why platform for proving that a Java or C program is a correct implementation of some specification. The key features are the introduction of parametricity both for types and for theories and an instantiation relation between theories. The proposed extensions are illustrated on two significant examples: the specification of the generic method for sorting arrays and for generic hash map.

Both problems considered in this thesis are related to SMT solvers. Firstly, decision procedures are at the core of SMT solvers. Secondly, the Why platform extracts verification conditions from a source program annotated by specifications, and then transmits them to SMT solvers or proof assistants to check the program correctness.

Keywords: Decision procedures, Paramodulation, Schematic Saturation, Combination