FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Reasoning in First-Order Theories with Extensionality

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Computational Intelligence

eingereicht von

Bernhard Kragl
Matrikelnummer 0952989

an der Fakultät für Informatik
der Technischen Universität Wien

Betreuerin: Priv.-Doz. Dr. Laura Kovács

Wien, 20.03.2014

(Unterschrift Verfasser) (Unterschrift Betreuerin)

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Reasoning in First-Order Theories with Extensionality

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Computational Intelligence

by

Bernhard Kragl
Registration Number 0952989

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Priv.-Doz. Dr. Laura Kovács

Vienna, 20.03.2014

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum)	(Unterschrift Verfasser)

Acknowledgements

First and foremost I want to thank my advisor, Laura Kovács. Her continuous support during my studies was invaluable and went way beyond the supervision of this thesis. I am especially thankful for the opportunity to present our work at the 19th International Conferences on Logic for Programming, Artificial Intelligence and Reasoning (LPAR) in Stellenbosch, South Africa.

I like to thank Ashutosh Gupta for his guidance and the numerous hours he spent with me developing ideas, improving my work, and explaining the natural laws of the scientific world. Our collaboration was supported by Tom Henzinger, who provided a working environment in his group at IST Austria.

I was lucky to meet Andrei Voronkov when I started to work on this thesis. His interest in my work and his incredible experience in the field of automated reasoning where of great help. Every question about first-order theorem proving and VAMPIRE was only one Skype session away from learning what makes the world's fastest theorem prover.

I am very grateful to my parents, who always supported me and prized my needs over theirs. Finally, my deepest gratitude goes to Sandra for her infinite love and patience.

The research in this thesis was partly supported by the Austrian National Research Network RiSE (FWF grants S11402-N23 and S11410-N23).

Abstract

Extensionality axioms are common when reasoning about collections, such as sets in mathematics, or arrays and functions in program analysis. An extensionality axiom asserts that two collections are equal if they consist of the same elements (at the same indices).

Using extensionality is often required to show that two collections are equal, a typical example is the set theory theorem $(\forall x)(\forall y)x \cup y = y \cup x$. Interestingly, while humans have no problem with proving such set identities using extensionality, they are very hard for superposition theorem provers because of the calculi they use.

In this thesis we show how addition of a new inference rule, called extensionality resolution, allows first-order theorem provers easily solve problems no modern first-order theorem prover can solve. We illustrate this by running the world-leading theorem prover VAMPIRE with extensionality resolution on a number of set theory and array problems.

Kurzfassung

Extensionalitätsaxiome sind zentral für das Schließen über Containerobjekte, wie etwa Mengen in der Mathematik, oder Arrays und Funktionen in der Programmanalyse. Ein Extensionalitätsaxiom besagt, dass zwei Container genau dann gleich sind, wenn sie dieselben Elemente (an denselben Indizes) beinhalten.

Die Anwendung eines Extensionalitätsaxioms ist oft erforderlich um zu zeigen, dass zwei Container gleich sind – ein typisches Beispiel ist das Mengenlehrentheorem $(\forall x)(\forall y)x \cup y = y \cup x$. Obwohl das Beweisen solcher Mengenidentitäten mittels Extensionalität für Menschen kein Problem darstellt, sind diese für automatische Beweiser aufgrund der verwendeten Kalküle problematisch.

In dieser Diplomarbeit zeigen wir, wie das Hinzufügen einer neuen Inferenzregel, genannt Extensionalitätsresolution, automatischen Beweisern erlaubt, mühelos Probleme zu lösen, welche kein moderner automatischer Beweiser erster Stufe lösen kann. Wir veranschaulichen dies anhand von zahlreichen Problemen über Mengen und Arrays durch Erweiterung des weltweit führenden automatischen Beweisers VAMPIRE um Extensionalitätsresolution.

Contents

CHAPTER 1

Introduction

1.1 Problem Description

Logic is among the most effective formal tools in computer science [33]. As such, automated reasoning in logic-based formalisms has a longstanding tradition and led to a rich theory as well as sophisticated and practical tools [47, 18]. The most important problems targeted by automated reasoners are the following.

SAT Satisfiability of propositional formulas (NP-complete)

SMT Satisfiability of ground formulas with respect to a theory (usually decidable)

First-order Validity/unsatisfiability of first-order formulas (semi-decidable)

Applications of automated reasoning include, for example, analysis/testing/verification/synthesis of hardware and software, protocol verification, query answering over a knowledge base and theorem proving in mathematics.

Satisfiability Modulo Theories (SMT) solvers have become fundamental to many program analysis tools. They implement efficient decision procedures for theories which in turn capture domain knowledge to support the modeling of complex systems. However, ground formulas cannot express proof obligations with quantifiers. On the contrary, automated first-order theorem provers are refutation complete for arbitrarily quantified formulas but weak in theory reasoning. There are theoretical results showing undecidability of reasoning with both theories and quantifiers.[1]

Verification of software libraries involves reasoning about data collections, such as arrays, sets, and functions. Most of the interesting properties about these data types are expressed using both quantifiers and theory specific predicates/functions. Hence, verification of such properties requires specialized reasoning support in the existing reasoning engines, such as SMT solvers,

[1] Note that there is no complete first-order axiomatization for certain theories, e.g. true arithmetic.

and automated and interactive theorem provers. There are essentially two approaches to solving this problem of automated reasoning:

1. Analyze the properties involved in a particular verification problem and, if possible, capture the properties by a restricted fragment of the logic which still admits a decision procedure.

2. Extend "general purpose" theorem provers with theory specific techniques.

In this thesis we follow the second approach by improving theory reasoning in first-order theorem provers. In our recent work on *tree interpolation* [20] we extended the first-order theorem prover VAMPIRE [37] to derive tree interpolants from arbitrary local proofs. Tree interpolation is closely related to, and can be used for, solving constrained Horn-clauses, a popular intermediate representation for verification problems [31, 29, 17, 19, 50, 49, 38, 34]. Our results highlight the advantage of our technique on quantified problems. However, on problems from bounded model checking of device drivers, expressed in the quantifier-free theory of linear integer arithmetic and integer-indexed arrays, SMT-based tools perform better. By analyzing the array reasoning part of VAMPIRE on these and other problems from the SMT community, we identified a general problem with reasoning about collection types using superposition-based first-order theorem provers, as follows.

For proving properties about collections one needs to use *extensionality axioms* asserting that two collections are equal if and only if they consist of the same elements (at the same indices). A typical example is the set theory theorem $(\forall x)(\forall y)x \cup y = y \cup x$, asserting that set union is commutative and therefore the union of two sets x and y is the same as the union of y and x. Proving the theorem requires set equality reasoning and cannot be proved using only standard axiomatizations of the equality predicate. For proving set equality, one needs to prove that equal sets contain the same elements, a property that is asserted by the extensionality axiom of set theory.

Interestingly, while humans have no problem with proving such set identities using extensionality, they are very hard for superposition-based theorem provers because of the calculi they use. To overcome this limitation, we need specialized methods of reasoning with extensionality, preferably those not requiring radical changes in the underlining inference mechanism and implementations of superposition.

In this thesis we present a new inference rule, called *extensionality resolution*, which allows first-order theorem provers easily solve problems no modern first-order theorem prover can solve. Our approach makes no changes on the underlying inference mechanism of superposition, and introduces no additional constraints on the orderings used by the theorem prover. Building extensionality resolution in a theorem prover needs efficient recognition and treatment of extensionality axioms. We therefore analyzed various forms of extensionality axioms and described various choices made, and corresponding options, for extensionality resolution. We implemented our approach in the first-order theorem prover VAMPIRE [37] and evaluated our method on a number of challenging examples from set theory and reasoning about arrays. Our experiments show significant improvements on problems containing extensionality axioms: for example, many problems proved by the new implementation in essentially no time could not be proved by any of the existing first-order provers.

1.2 Structure of the Thesis

The thesis is structured as follows. We start with an introduction of the relevant notions of first-order logic and theorem proving in Chapter 2. Then, in Chapter 3, we present examples of reasoning with extensionality axioms and the technical details of why it is hard for superposition-based theorem provers. Our new inference rule extensionality resolution and its integration into saturation algorithms is presented in Chapter 4. In Chapter 5 we show different options for recognizing extensionality axioms derived from the analysis of a large collection of first-order benchmark problems. In Chapter 6 we evaluate our implementation of extensionality resolution in VAMPIRE and compare it to other state-of-the-art theorem provers. In Chapter 7 we review existing approaches to reasoning with both theories and quantifiers. In Chapter 8 we conclude the thesis and suggest directions for further research.

CHAPTER 2

Automated First-Order Theorem Proving

In this chapter we review saturation-based theorem proving in the superposition calculus. After fixing our notation, we introduce the notion of inference systems and the key ingredients of efficient first-order theorem proving. The material of this section is based on [8, 40, 37].

2.1 Basic notions

We consider the standard first-order predicate logic with equality. Let $\mathcal{V}$ be a set of variables and let $\Sigma = \langle \mathcal{P}, \mathcal{F} \rangle$ be a first-order signature, where $\mathcal{P}$ is a set of predicate symbols and $\mathcal{F}$ is a set of function symbols. Terms over $\mathcal{F}$ and $\mathcal{V}$ are defined as usual. Formulas over terms and $\mathcal{P}$ are built using all Boolean connectives, quantifiers over $\mathcal{V}$, and $\top$ and $\bot$ for always true respectively false formulas. The equality predicate is denoted by =. We write $s \neq t$ to mean $\neg(s = t)$, and analogous for every binary predicate written in infix notation. Throughout this thesis we denote variables by x, y, z, constant symbols by a, b, c, function symbols by f, g, h, predicate symbols by p, q, terms by l, r, s, t, u and formulas by F, possibly with indices.

An *atomic formula* A is of the form $p(t_1, \ldots, t_n)$. A *literal* L is either an atomic formula A or its negation $\neg A$. The former is called a *positive literal*, the latter a *negative literal*. A *clause* C is a finite multiset of literals, identified with their disjunction $L_1 \vee \cdots \vee L_n$. The *empty* clause is denoted by $\Box$. All variables in a clause are implicitly universally quantified. We call an *expression* E a term, atom, literal, or clause. An expression is *ground* if it contains no variables.

A *substitution* θ is a finite mapping from variables to terms, written as $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$. The application of this substitution to an expression E, denoted by $E\theta$, is the expression obtained from E by the simultaneous replacements of each x_i by t_i. An expression E_1 is called an *instance* of expression E_2, if $E_1 = E_2\theta$ for some substitution θ. A *unifier* of two expressions E_1 and E_2 is a substitution θ such that $E_1\theta = E_2\theta$. If there exists a unifier, then there exists a *most general unifier* (mgu) which is unique up to variable renaming.

We write $E[s]$ to mean an expression E with a particular occurrence of a term s. When we use the notation $E[s]$ and then write $E[t]$, the latter means the expression obtained from $E[s]$ by replacing the distinguished occurrence of s by the term t.

We interpret terms and formulas using standard first-order *structures* $\mathcal{M}$. We write $t^{\mathcal{M}}$ for the domain element t denotes in $\mathcal{M}$ and $F^{\mathcal{M}} = 1$ to denote that F is true in $\mathcal{M}$; then F is called *satisfiable* and $\mathcal{M}$ is called a *model* of F, otherwise $F^{\mathcal{M}} = 0$ and $\mathcal{M}$ is called a *counterexample* for F. If F has no models at all, F is called *unsatisfiable*. We generalize these notions to sets of formulas Γ as usual. E.g. Γ is called satisfiable if there is some structure $\mathcal{M}$ such that $F^{\mathcal{M}} = 1$ for all $F \in \Gamma$. A formula F is a *logical consequence* of a set of formulas Γ if F holds in all models of Γ, symbolically $\Gamma \models F$.

We have the following equivalence between the notions of logical consequence and unsatisfiability:

$$\Gamma \models F \quad \text{if and only if} \quad \Gamma \cup \{\neg F\} \text{ unsatisfiable} \tag{2.1}$$

We say a set of formulas Γ *defines* a set of structures $\mathcal{S}$, if it holds for all structures $\mathcal{M}$ that $\mathcal{M}$ is a model of Γ if and only if $\mathcal{M} \in \mathcal{S}$.

2.2 Proofs and Refutations

The problem a theorem prover aims to solve is to compute the membership predicate of the logical consequence relation $\models$. That is, given formulas $F_1, \ldots, F_n$ (axioms and assumptions) and a formula F (conjecture), determine whether $\{F_1, \ldots, F_n\} \models F$ holds or not. In general, the problem is only semi-decidable.

Reasoning semantically about formulas in terms of models is not always desired. Especially for automated reasoning by a computer program we need symbolic techniques. This is the realm of proof theory, which is concerned with the construction and analysis of proofs as formal syntactic objects. The strong correspondence between syntax and semantics in classical first-order logic, established by the soundness and completeness of proof systems, justifies proof search as a suitable method to determine logical consequence. Furthermore, a proof is a witness for the answer of a theorem prover and can be verified.

There are several different proof systems for classical first-order logic, for example the numerous variants of sequent calculus, natural deduction and Hilbert-type systems. However, not all of them are well suited for automation. On of the most successful proof systems in automated reasoning is resolution, introduced in 1965 [46]. Even today the fastest theorem provers are based on the ideas of resolution. We will use superposition as an extension of resolution to support equality, and instantiation based methods are guided by the resolution principle (see Chapter 7).

We now give an abstract framework for proof systems.[1] An *inference rule* is a n-ary relation

[1] This framework captures e.g. resolution, superposition and Hilbert-type systems. In sequent calculus derivation trees the nodes are sequents, not formulas. In natural deduction derivation trees there are non local dependencies for inferences.

between formulas. The elements of an inference rule are called *inferences* and are written as

$$\frac{F_1 \quad \dots \quad F_n}{F} \ . \tag{2.2}$$

$F_1, \dots, F_n$ are called the *premises* and F the *conclusion* of the inference. Inferences with no premises, i.e. $n = 0$, are called *axioms*. An *inference system* $\mathbb{I}$ is a set of inference rules.

A *derivation* in $\mathbb{I}$ is a finite tree constructed from inferences (2.2) in $\mathbb{I}$. If the root node is F, we call the derivation a *derivation of* F. A *derivation from* $F_1, \dots, F_n$ is a derivation where all leaves are axioms or in $\{F_1, \dots, F_n\}$. A *proof* is a derivation where all leaves are axioms. A derivation of $\bot$ is a *refutation*. We write $F_1, \dots, F_n \vdash_{\mathbb{I}} F$ to denote that there exists a derivation of F from $F_1, \dots, F_n$ in $\mathbb{I}$.

The abstract formulations of soundness and completeness of an inference system $\mathbb{I}$ are as follows:

- **Soundness:** When $F_1, \dots, F_n \vdash_{\mathbb{I}} F$ then $\{F_1, \dots, F_n\} \models F$.
- **Completeness:** When $\{F_1, \dots, F_n\} \models F$ then $F_1, \dots, F_n \vdash_{\mathbb{I}} F$.

In fact this properties can be strengthened as follows. A derivation of F from $\{F_1, \dots, F_n\}$ can be considered a derivation from any superset $\Gamma \supseteq \{F_1, \dots, F_n\}$. If we have $\Gamma \models F$, then as a consequence of compactness there is a finite $\{F_1, \dots, F_n\} \subseteq \Gamma$ such that $\{F_1, \dots, F_n\} \models F$. Then by the above completeness $F_1, \dots, F_n \vdash_{\mathbb{I}} F$ and we can write $\Gamma \vdash_{\mathbb{I}} F$. Hence we can state soundness and completeness concisely as $\vdash_{\mathbb{I}} \subseteq \models$ and $\models \subseteq \vdash_{\mathbb{I}}$, respectively.

2.3 Resolution

Resolution and superposition are based on the principle of proof by contradiction. According to (2.1) we add the negation of the conjecture to the assumptions and prove the unsatisfiability of the resulting set of formulas. Hence, instead of a derivation of F from Γ we construct a refutation of $\Gamma \cup \{\neg F\}$. Furthermore, resolution and superposition work on formulas in *conjunctive normal form* (CNF). That is, $\Gamma \cup \{\neg F\}$ is converted into an equisatisfiable set of clauses. In the rest of this section we denote the set S_0. See [4, 41] for surveys on normal form translations.

The classic *resolution inference system* consists of the following rules:

Resolution:

$$\frac{A \vee C_1 \quad \neg A' \vee C_2}{(C_1 \vee C_2)\theta} \ ,$$

where θ is a mgu of A and A'.

Factoring:

$$\frac{A \vee A' \vee C}{(A \vee C)\theta} \ ,$$

where θ is a mgu or A and A'.

One subtle point to note here is that we always assume the premises of resolution to have disjoint variables. That is, we apply a variable renaming before unification.

Example 1. Consider the inference

$$\frac{p(x) \vee q(z) \quad \neg p(y) \vee p'(z)}{q(z) \vee q'(z)} \; \{x \mapsto y\}.$$

The conclusion merges the z variables from both premises, weakening the result. Subscripting 1 to variables in the first premise and 2 to variables in the second premise gives the desired resolution inference

$$\frac{p(x_1) \vee q(z_1) \quad \neg p(y_2) \vee p'(z_2)}{q(z_1) \vee q'(z_2)} \; \{x_1 \mapsto y_2\}.$$

Resolution is *refutation complete*, that is, exhaustively applying resolution and factoring to an unsatisfiable set of clauses S_0 will eventually produce $\square$. This is the underlying principle of *saturation-based* theorem proving: keep a set S of clauses (initially S_0) called the *search space* and saturate it by gradually adding conclusions of inferences with premises in S, until all conclusions of possible inferences are already in S. However, blindly applying inferences quickly blows up the search space and works for non but the most trivial problems. Before presenting all ingredients to make saturation-based theorem proving work in practice, we investigate how equality reasoning can be integrated into resolution, leading to the superposition inference system.

2.4 Equality

The equality predicate is essential to formulate problems in first-order logic and hence has to be supported by automated theorem provers. There is a philosophical principle attributed to Leibniz[2] which states that equality of two objects means that they have all properties in common. In other words, two objects are considered equal if they behave the same in all contexts and hence are indiscernible.

The standard approach in first-order logic is to add equality as primitive to the logic and assign it the following fixed interpretation:

$$(s = t)^{\mathcal{M}} = 1 \quad \text{if and only if} \quad s^{\mathcal{M}} = t^{M} \tag{2.3}$$

Note that here all three occurrences of $=$ are different: the symbol part of the logic, equality among truth values and equality among domain elements. Hence equality is interpreted as the identity in the domain of the respective structure. A structure which interprets equality as in (2.3) is called *equality structure*. First-order logic with primitive equality and interpretation restricted to equality structures is called *first-order logic with equality*.

First-order logic with equality is more expressive than first-order logic without equality. For example it allows to define finite models (with certain size restrictions); the clause $x = a_1 \vee \cdots \vee x = a_n$ defines the models with at most n elements. Nevertheless, the set of axioms from Figure 2.1, denoted by $\mathcal{E}$, allows the reduction of reasoning with equality to reasoning

[2] Known as Leibniz's law, "the principle of substitutivity", "the indiscernibility of identicals", or "the replacement property".

$x = x$	(reflexivity)
$x = y \to y = x$	(symmetry)
$x = y \wedge y = z \to x = z$	(transitivity)
$x_1 = y_1 \wedge \cdots \wedge x_n = y_n \to f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n)$	($\mathcal{F}$-monotonicity)
$x_1 = y_1 \wedge \cdots \wedge x_n = y_n \wedge p(x_1, \ldots, x_n) \to p(y_1, \ldots, y_n)$	($\mathcal{P}$-monotonicity)

Figure 2.1: Equality Axiomatization. All variables are universally quantified. The monotonicity axioms are actually axiom schemes and have to be added for all $f \in \mathcal{F}$ and $p \in \mathcal{P}$.

without equality as follows: S is satisfiable in first-order logic with equality if and only if $S \cup \mathcal{E}$ is satisfiable in first-order logic without equality [2].[3]

There are at least two problems with this solution. First, the number of monotonicity clauses depends on the signature. Second, the clauses in $\mathcal{E}$ are extremely prolific. Any equality literal can be unified and resolved with numerous clauses in $\mathcal{E}$, generating mostly useless clauses. The solution of superposition is to add special inference rule for handling equality reasoning to the calculus. In addition, the concepts of orderings and selection allow to impose restrictions on the inference rule to minimize the number of necessary inferences.

In the following chapters we show how extensionality axioms give additional meaning to the equality predicate, causing problems for automated reasoners. Our solution presented in Chapter 4 is similar to the idea of superposition, namely special treatment of extensionality in the calculus.

It is interesting to note that Leibniz equality can be naturally defined in higher-order logic, e.g. as

$$=_\alpha \quad \stackrel{\text{def}}{=} \quad \lambda x^\alpha y^\alpha . \forall p^{\alpha \to o} \leftrightarrow (p\ x)\ (p\ y)$$

in classic type theory. But also here, despite other issues, automated reasoning methods require built in support for equality and extensionality [16, 14, 13, 15].

2.5 Superposition

The most common inference system for first-order logic with equality is the superposition inference system, developed in the early 1990s [5, 6, 7]. The superposition inference system is, in fact, a family of systems, parameterized by a simplification ordering and a selection function.

A *simplification ordering* $\succ$ is a partial ordering on terms with the following properties:

- $\succ$ is well-founded: there is no infinite decreasing chain $t_1 \succ t_2 \succ \ldots$;
- $\succ$ is monotonic: if $l \succ r$, then $s[l] \succ s[r]$;
- $\succ$ is stable under substitutions: if $l \succ r$, then $l\theta \succ r\theta$;

[3] However, there is no axiomatization which defines the equality structures.

- $\succ$ has the subterm property: if s is a proper subterm of $t[s]$, then $t[s] \succ s$.

If $s \succ t$ we say that s (t) is *bigger (smaller)* then t (s). We lift $\succ$ to literals by considering predicate symbols and $\neg$ as function symbols, and further to clauses by multiset extension.

A *selection function* σ is a function from clauses to clauses, such that $\sigma(C) \subseteq C$ and $\sigma(C) \neq \emptyset$. A literal $L \in \sigma(C)$ is called *selected*. When working with a selection function, we will underline selected literals: if we write a clause in the form $\underline{L} \vee C$, it means that L (and maybe some other literals) are selected in $L \vee C$. A *well-behaved selection function* selects in every clause either some negative literal or all maximal literals with respect to the simplification ordering $\succ$.

In the sequel, we assume that a simplification ordering $\succ$ and a selection function σ are fixed. The *superposition inference system*, denoted by $\mathrm{Sup}^{\succ}_{\sigma}$, consists of the following rules:

Resolution:

$$\frac{\underline{A} \vee C_1 \quad \underline{\neg A'} \vee C_2}{(C_1 \vee C_2)\theta} \ ,$$

where θ is a mgu of A and A'.

Factoring:

$$\frac{\underline{A} \vee \underline{A'} \vee C}{(A \vee C)\theta} \ ,$$

where θ is a mgu or A and A'.

Superposition:

$$\frac{\underline{l = r} \vee C_1 \quad \underline{L[s]} \vee C_2}{(L[r] \vee C_1 \vee C_2)\theta} \quad \frac{\underline{l = r} \vee C_1 \quad \underline{t[s] = t'} \vee C_2}{(t[r] = t' \vee C_1 \vee C_2)\theta} \quad \frac{\underline{l = r} \vee C_1 \quad \underline{t[s] \neq t'} \vee C_2}{(t[r] \neq t' \vee C_1 \vee C_2)\theta} \ ,$$

where θ is a mgu of the terms l and s, s is not a variable, $r\theta \not\succeq l\theta$, (first rule only) $L[s]$ is not an equality literal, and (second and third rules only) $t'\theta \not\succeq t[s]\theta$.

Equality Resolution:

$$\frac{\underline{s \neq t} \vee C}{C\theta} \ ,$$

where θ is a mgu of the terms s and t.

Equality Factoring:

$$\frac{\underline{s = t} \vee \underline{s' = t'} \vee C}{(s = t \vee t \neq t' \vee C)\theta} \ ,$$

where θ is an mgu of s and s', $t\theta \not\succeq s\theta$, and $t'\theta \not\succeq t\theta$.

Note the general pattern that inferences are only performed with selected literals. The superposition rule can be described as rewriting terms by equal terms. Let $\underline{l = r} \vee C_1$ be the left premise of a superposition inference. If an instance of l occurs in some other selected literal, this occurrence can be replaced by r. We only rewrite with smaller terms and if we rewrite an equality literal, then only the bigger side of the equality literal. Furthermore, we do not have to rewrite variables.

2.6 Saturation and Redundancy

Superposition is a powerful inference system, and orderings and selection already discard many inferences. However, we are still missing one essential concept of modern resolution and superposition theorem proving, the theory of *redundancy*.

The above idea of saturation was to start with the initial set of clauses S_0 and saturate it with all possible inferences. Formally, a set of clauses S is *saturated* with respect to an inference system $\mathbb{I}$, if for all inferences in $\mathbb{I}$ with premises in S, the conclusion is also in S. If $\mathbb{I}$ is refutation complete and S is the smallest saturated set with respect to $\mathbb{I}$ containing S_0, then S_0 is unsatisfiable if and only if $\square \in S$. To build a saturated set in practice, we need an algorithm which gradually selects inferences and adds their conclusions to the search space. But, as it turns out, it is even possible to *delete* certain clauses from the search space without affecting completeness.

A clause C is called *redundant* in S if there are $S' \subseteq S$ such that $S' \models C$ and $C \succ C'$ for all $C' \in S'$. An $\mathbb{I}$ *inference process* is a sequence $S_0, S_1, \ldots$ of clauses such that for each *inference step* S_i, S_{i+1} either

- $S_{i+1} = S_i \cup \{C\}$ and there is an inference

$$\frac{C_1 \quad \ldots \quad C_n}{C}$$

  in $\mathbb{I}$ with $\{C_1, \ldots, C_n\} \subseteq S_i$; or

- $S_{i+1} = S_i - \{C\}$ and C is redundant in S_i.

Obviously, even without deleting redundant clauses, completeness alone does not guarantee to eventually find $\square$ starting from an unsatisfiable set S_0. The clauses in the limit $S_\infty = \bigcup_{i \geq 0} \bigcap_{k > i} S_k$ of an inference process are called *persistent*. An inference process is *fair*, if the conclusion of every inference with persistent premises is added at some inference step. In other words, an inference enabled at some point is not allowed to be postponed indefinitely.

We are now ready to formulate the strong completeness property of superposition.

Theorem 1 (Completeness of Superposition)**.** *Let $\succ$ be a simplification ordering, σ a well-behaved selection function and $S_0, S_1, \ldots$ a fair $\mathrm{Sup}^{\succ}_{\sigma}$ inference process. Then S_0 is unsatisfiable if and only if $\square \in S_i$ for some $i \geq 0$.*

2.7 Theorem Proving in Practice

Theorem 1 not only gives us conditions for the existence of refutations, but also conditions under which we are guaranteed to find them. An algorithm implementing inference processes is called a *saturation algorithm*. Modern first-order theorem provers implement different versions of saturation algorithms. The VAMPIRE theorem prover, for example, implements 3 saturation algorithms from the family of *given clause algorithms*. For an overview of saturation algorithms we refer to [45, 37].

```
input: init: set of clauses;
var active, passive, unprocessed: set of clauses;
var given, new: clause;
active := ∅;
unprocessed := init;
loop
  while unprocessed ≠ ∅
    new:=pop(unprocessed);
    if new = □ then return unsatisfiable;
    if retained(new) then                                   (* retention test *)
      simplify new by clauses in active ∪ passive ;         (* forward simplification *)
      if new = □ then return unsatisfiable;
      if retained(new) then                                 (* another retention test *)
        delete and simplify clauses in active and           (* backward simplification *)
                                        passive using new;
        move the simplified clauses to unprocessed;
        add new to passive;
  if passive = ∅ then return satisfiable or unknown;
  given := select(passive);                                 (* clause selection *)
  move given from passive to active;
  unprocessed:=forward_infer(given, active);                (* forward generating inferences *)
  add backward_infer(given, active) to unprocessed;         (* backward generating inferences *)
```

Figure 2.2: Otter Saturation Algorithm.

In what follows, we illustrate the essential concepts of given-clause saturation algorithms on the Otter saturation algorithm, implemented also in Vampire. A simplified description of the Otter saturation algorithm is shown in Figure 2.2. It uses three kinds of inferences: *generating*, which add new clauses to the search space; *simplifying*, which replace existing clauses by new ones, and *deletion*, which delete clauses from the search space.

Generating inferences are the standard inferences like resolution or superposition. The distinction of forward and backward generating inferences is explained in Chapter 4 using the example of our new inference rule. Simplifying inferences are in principle also generating inferences as they add a new clauses. However, they are guaranteed to make another clause in the search space redundant. The redundant clause can be deleted and hence adding the new clause comes at no cost. Deletion rules remove redundant clauses, e.g. tautologies, as part of the *retention test*.

The algorithm maintains three sets of clauses:

1. $active$: the set of clauses to which generating inferences have already been applied;

2. $passive$: clauses that are retained by the prover (that is, not deleted);

3. $unprocessed$: clauses that are in a queue for a retention test.

At each step, the algorithm either processes a clause new, picked from $unprocessed$, or performs generating inferences with the so-called *given clause* $given$, which is the clause most recently added to $active$. The algorithm maintains two loop invariants:

1. All generating inferences with premises in $active$ have been performed.

2. $active \cup passive$ is maximally simplified with respect to the used simplification rules.

Fairness of the induced inference process is achieved through clause selection. That is, if no clause remains passive indefinitely, we are guaranteed to perform every inference with persistent premises.

All operations performed by the saturation algorithm that may take considerable time to execute are normally implemented using *term indexing*, that is, building a special purpose index data structure that makes the operation faster [44]. For example, all theorem provers with built-in equality reasoning have an index for forward demodulation. The challenging task is to index complicated data structures (e.g. sets of trees) subject to a high number of update operations within reasonable memory expenses.

CHAPTER 3

Motivating Examples

In this chapter we explain why theories with extensionality axioms require special treatment in superposition theorem provers.

3.1 Set Theory

We start with an axiomatization of the set theory and will refer to this axiomatization in the rest of the thesis. The set theory will use the membership predicate $\in$ and the subset predicate $\subseteq$, the constant $\varnothing$ denoting the empty set, and operations $\cup$ (union), $\cap$ (intersection), $-$ (difference), $\triangle$ (symmetric difference), and complement, denoted by over-lining the expression it is applied to, that is, the complement of a set x is denoted by $\overline{x}$. An axiomatization of set theory with these predicates and operations is shown in Figure 3.1.

Note that our set theory is different from axiomatic set theory à la Zermelo-Fraenkel. In particular, we have a two sorted logic which distinguishes between sets and elements.

$(\forall x)(\forall y)((\forall e)(e \in x \leftrightarrow e \in y) \rightarrow x = y)$	(extensionality)
$(\forall x)(\forall y)(x \subseteq y \leftrightarrow (\forall e)(e \in x \rightarrow e \in y))$	(definition of subset)
$(\forall e)(e \notin \varnothing)$	(definition of the empty set)
$(\forall x)(\forall y)(\forall e)(e \in x \cup y \leftrightarrow e \in x \vee e \in y)$	(definition of union)
$(\forall x)(\forall y)(\forall e)(e \in x \cap y \leftrightarrow e \in x \wedge e \in y)$	(definition of intersection)
$(\forall x)(\forall y)(\forall e)(e \in x - y \leftrightarrow e \in x \wedge e \notin y)$	(definition of set difference)
$(\forall x)(\forall y)(\forall e)(e \in x \triangle y \leftrightarrow (e \in x \leftrightarrow e \notin y))$	(definition of symmetric difference)
$(\forall x)(\forall e)(e \in \overline{x} \leftrightarrow e \notin x)$	(definition of complement)

Figure 3.1: Set theory axiomatization. Here we denote set variables by x, y and set elements by e.

Example 2. The commutativity of union is a valid property of sets and a logical consequence of the set theory axiomatization:

$$(\forall x)(\forall y)\ x \cup y = y \cup x. \tag{3.1}$$

This identity is problem 2 in our problem suite of Chapter 6. Proving such properties poses no problem to humans. We present an example of a human proof.

(1) Take two arbitrary sets a and b. We have to prove $a \cup b = b \cup a$.

(2) By extensionality, to prove (1) we should take an arbitrary element e and prove that $e \in a \cup b$ if and only if $e \in b \cup a$.

(3) We will prove that $e \in a \cup b$ implies $e \in b \cup a$, the reverse direction is obvious.

(4) To this end, assume $e \in a \cup b$. Then, by the definition of union, $e \in a$ or $e \in b$. Again, by the definition of union, both $e \in a$ implies $e \in b \cup a$ and $e \in b$ implies $e \in b \cup a$. In both cases we have $e \in b \cup a$, so we are done.

The given proof is almost trivial. Apart from the application of extensionality (step 2) and Skolemization (introduction of constant a, b, e), it uses the definition of union and propositional inferences.

What is interesting is that this problem is hard for first-order theorem provers. If we use our full axiomatization of set theory, none of the top three first-order provers according to the CASC-24 theorem proving competition of last year [53], that is VAMPIRE [37], E [51] and IPROVER [35], can solve it. If we only use the relevant axioms, that is extensionality and the definition of union, these three provers can prove the problem, however not immediately, with runtimes ranging from 0.24 to 27.18 seconds.

If we take slightly more complex set identities, the best first-order theorem provers cannot solve them within reasonable time. We next give such an example.

Example 3. Consider the following conditional identity:

$$(\forall x)(\forall y)(\forall z)(x \cap y \subseteq z \wedge z \subseteq x \cup y \rightarrow (x \cup y) \cap (\overline{x} \cup z) = y \cup z) \tag{3.2}$$

The above formula cannot be proved by any existing theorem prover within a 1 hour time limit. This formula is problem 25 in our problem suite of Chapter 6. Our problem suite contains four other problems that no prover could solve.

It is not hard to analyze the reason for the failure of superposition provers for examples requiring extensionality, such as Example 3: it is the treatment of the extensionality axioms. Suppose that we use a superposition theorem prover and use the standard Skolemization and CNF transformation algorithms. Then one of the clauses derived from the extensionality axiom of Figure 3.1 is:

$$f(x, y) \notin x \vee f(x, y) \notin y \vee x = y. \tag{3.3}$$

$(\forall x)(\forall i)(\forall e)\ select(store(x,i,e),i) = e$ (select-over-store 1)
$(\forall x)(\forall i)(\forall j)(\forall e)\ (i \neq j \rightarrow select(store(x,i,e),j) = select(x,j))$ (select-over-store 2)
$(\forall x)(\forall y)\ (((\forall i)\ select(x,i) = select(y,i)) \rightarrow x = y),$ (extensionality)

Figure 3.2: Axiomatization of the theory of arrays. Here the variables x, y denote arrays, i, j array indices and e an array element; $select$ is the standard select/read function, $store$ the standard store/write function over arrays.

Here f is a Skolem function. This clause is also required for a computer proof, since without it the resulting set of clauses is satisfiable.

Independently of the ordering used by a theorem prover, $x = y$ will be the smallest literal in clause (3.3). Since it is also positive, no superposition prover will select this literal. Thus, the way the clause will be used by superposition provers is to use already proved membership literals $s \in t$ to derive a new set identity obtained by instantiating $x = y$. Note that it will be used in the same way independently of whether the goal is $a \cup b = b \cup a$ or any other set identity. This essentially means that the only way to prove $a \cup b = b \cup a$ is to saturate the rest of the clauses until $x \cup y = y \cup x$ is derived, and likewise for all other set identities! This explains why theorem provers are very inefficient when an application of extensionality is required to prove a set identity.

3.2 Arrays

We now give an example of extensionality reasoning over arrays. The standard axiomatization of the theory of arrays, given in Figure 3.2, also contains an extensionality axiom. The clause derived from the array extensionality axiom is:

$$select(x, g(x,y)) \neq select(y, g(x,y)) \vee x = y, \tag{3.4}$$

where g is a Skolem function. Note that this axiom is different from that of sets because arrays are essentially maps and two maps are equal if they contain the same elements at the same indices.

Example 4. Consider the following formula expressing the valid property that the result of updating an array at two different indices does not depend on the order of updates:

$$i_1 \neq i_2 \rightarrow store(store(a, i_1, v_1), i_2, v_2) = store(store(a, i_2, v_2), i_1, v_1). \tag{3.5}$$

Again, this problem (and similar problems for a larger number of updates) is very hard for theorem provers, see Chapter 6. The explanation of why it is hard is the same as for sets: the extensionality axiom is used by theorem provers in "the wrong direction" because the literal $x = y$ in the clause (3.4) is never selected.

3.3 Solutions?

Though extensionality is important for reasoning about collections, and collection types are first-class in nearly all modern programming languages, reasoning with extensionality is hard for theorem provers because of the (otherwise very efficient) superposition calculus implementation. It is interesting to note that the MUSCADET theorem prover [42] proves set theory identities better than superposition provers, since it does not have full equality reasoning and always treats implications as rules for reducing the right-hand side of the implication to its left-hand side.

The above discussion may suggests that one simple solution would be to select $x = y$ in clauses derived from an extensionality axiom. Note that selecting *only* $x = y$ will result in a loss of completeness, so we can assume that it is selected *in addition* to the literals a theorem prover normally selects. It is not hard to see that this solution effectively makes provers fail on most problems. The reason is that superposition from a variable, resulting from selecting $x = y$, can be done in *every non-variable term*. For example, consider the clause

$$e \in x - y \vee e \in x \vee e \notin y, \tag{3.6}$$

obtained by converting the set difference axiom of Figure 3.1 into CNF and suppose that the first literal is selected in it. A superposition step from the extensionality clause (3.3) into this clause gives

$$f(x - y, z) \notin x - y \vee f(x - y, z) \notin z \vee e \in z \vee e \in x \vee e \notin y. \tag{3.7}$$

Note the size of the new clause and also that it contains new occurrences of $x - y$, to which we can apply extensionality again.

From the above example it is easy to see that selecting $x = y$ in the extensionality clause (3.3) will result in a rapid blow-up of the search space by large clauses. One can think of other radical solutions, as in MUSCADET, whose drawback is that equality reasoning may become very inefficient, with a possible loss of completeness.

The solution we propose and defend in this thesis is to add a special generating inference rule for treating extensionality, called *extensionality resolution*, which requires relatively simple changes in the architecture of a superposition theorem prover.

CHAPTER 4

Reasoning in First-Order Theories with Extensionality Axioms

In this chapter we explain our solution to problems arising in reasoning with extensionality axioms. For doing so, we introduce the new inference rule *extensionality resolution* and show how to integrate it into a superposition theorem prover.

4.1 The Generic Extensionality Axiom

In Chapter 3 we showed two particular examples of extensionality axioms, the set extensionality axiom in Figure 3.1 and the array extensionality axiom in Figure 3.2. Although this gives some intuition, our use of the term "extensionality axiom" is informal. Up to now we do not have a precise way to tell if a certain formula is an extensionality axiom or not.

The *extensionality principle* we want to capture is the following refined notion of equality, which abstracts from the internal definitions of objects: two objects of a certain type are equal, if and only if they are indiscernible by *a particular* property. This can be considered as restricted Leibniz equality, which requires indiscernibility by *every* property. For example, equal arrays are indiscernible with respect to selection at arbitrary indices. The array extensionality axiom is especially interesting, because it uses equality on array elements. This equality can be again extensionally defined, e.g. for arrays of arrays.

According to the above principle, an extensionality axiom defines equality for a particular type. And indeed, our familiar examples consist of an implication with an equality among variables on the right-hand side. But what about the reverse implication, e.g.

$$(\forall x)(\forall y)(x = y \rightarrow (\forall e)(e \in x \leftrightarrow e \in y))$$

in the set extensionality axiom? This implication is already a logical consequence of the standard axioms of equality and a tautology in first-order logic with equality. Therefore, the reverse implications with the equality among variables on the left-hand side are redundant.

We conjecture that our informal extensionality principle can be expressed by formulas of the form

$$(\forall x)(\forall y)(F \to x = y) \tag{4.1}$$

and hence restrict our considerations to extensionality axioms of this form. However, as we will argue in Chapter 5, not every formula of the form (4.1) is intended to be an extensionality axiom. Furthermore, it is a priori not clear which formulas F on the left-hand side of the implication constitute an extensionality axiom.

4.2 Extensionality Resolution

Since extensionality axioms are not uniquely defined, our new inference rule is parameterized by a function for recognizing extensionality axioms.

Definition 1 (Extensionality Recognizer)**.** An *extensionality recognizer* is a partial function ext_rec, such that for every clause C, $ext\_rec(C)$ is either undefined, or returns a positive equality among variables $x = y$ from C.

Note that every clause derived from an extensionality axiom of the form (4.1) contains a positive equality among variables, but in general not every clause containing such an equality corresponds to an extensionality axiom. Chapter 5 is devoted to the design of extensionality recognizers.

We will also sometimes use an extensionality recognizer as Boolean function, meaning that it is true if and only if it is defined. For the rest of this chapter, let ext_rec be a fixed extensionality recognizer.

Definition 2 (Extensionality Clause)**.** We call an *extensionality clause* any clause C for which $ext\_rec(C)$ holds.

Definition 3 (Extensionality Resolution)**.** The *extensionality resolution* rule is the following inference rule:

$$\frac{x = y \vee C \quad \underline{s \neq t} \vee D}{C\theta \vee D}\ , \tag{4.2}$$

where

1. $ext\_rec(x = y \vee C) = (x = y)$, hence, $x = y \vee C$ is an extensionality clause;
2. θ is the substitution $\{x \mapsto s, y \mapsto t\}$.

Note that, since equality is symmetric, there are two inferences between the premises of (4.2); one is given above and the other one is with the substitution $\{x \mapsto t, y \mapsto s\}$.

Example 5. Consider two clauses: clause (3.3) and the unit clause $a \cup b \not\approx b \cup a$. Suppose that the former clause is recognized as an extensionality clause. Then the following inference is an instance of extensionality resolution:

$$\frac{f(x,y) \not\in x \vee f(x,y) \not\in y \vee x = y \quad a \cup b \not\approx b \cup a}{f(a \cup b, b \cup a) \not\in a \cup b \vee f(a \cup b, b \cup a) \not\in b \cup a} \ .$$

Given a clause with a selected literal $s \not\approx t$, which can be considered as a request to prove $s = t$, extensionality resolution replaces it by an instance of the premises of extensionality. This example shows that an application of extensionality resolution achieves the same effect as the use of extensionality in the "human" proof of Example 2.

Example 6. Similarly, consider the array extensionality clause (3.4) and the clause

$$\underbrace{store(store(a, i_1, v_1), i_2, v_2)}_{t_1} \not\approx \underbrace{store(store(a, i_2, v_2), i_1, v_1)}_{t_2}$$

derived from negating the array theorem (3.5). Then the extensionality resolution inference

$$\frac{select(x, g(x,y)) \not\approx select(y, g(x,y)) \vee x = y \quad t_1 \not\approx t_2}{select(t_1, g(t_1,t_2)) \not\approx select(t_2, g(t_1,t_2))} \ .$$

performs the goal directed reduction of the array inequality to the inequality of some array cell.

4.3 Integration into Saturation

Let us now explain how extensionality resolution can be integrated in a saturation algorithm of a superposition theorem prover. The key questions to consider is when the rule is applied and whether this rule requires term indexing or other algorithms to be performed. The implementation is similar for all saturation algorithms; for ease of presentation we will describe it only for the Otter saturation algorithm presented in Chapter 2. In fact, the architecture of VAMPIRE allowed the complete integration in the base class of all saturation algorithms.

Extensionality resolution is a generating inference rule, so the relevant lines of the saturation algorithm from Figure 2.2 are the ones at the bottom, referring to generating inferences. The same saturation algorithm with extensionality resolution related parts marked by ✓ is shown in Figure 4.1.

As one can see from the algorithm in Figure 4.1, extensionality resolution is easy to integrate into superposition theorem provers. The reason is that it requires no sophisticated indexing to find candidates for inferences: extensionality resolution applies to *every* extensionality clause and *every* clause with a negative selected equality literal. Therefore, we only have to maintain two collections: neg_equal of active clauses having a negative selected equality literal and ext of extensionality clauses as recognized by the function ext_rec.

Extensionality resolution has two premises and hence the implementation of the inference is split into a forward and a backward part, depending on the role of the given clause.

```
  input: init: set of clauses;
  var active, passive, unprocessed: set of clauses;
  var given, new: clause;
✓ var neg_equal, ext: set of clauses;
  active := ∅;
  unprocessed := init;
  loop
    while unprocessed ≠ ∅
      new :=pop(unprocessed);
      if new = □ then return unsatisfiable;
      if retained(new) then                                  (* retention test *)
        simplify new by clauses in active ∪ passive ;        (* forward simplification *)
        if new = □ then return unsatisfiable;
        if retained(new) then                                (* another retention test *)
          delete and simplify clauses in active and          (* backward simplification *)
                                              passive using new;
          move the simplified clauses to unprocessed;
          add new to passive;
    if passive = ∅ then return satisfiable or unknown;
    given := select(passive);                                (* clause selection *)
    move given from passive to active;
    unprocessed :=forward_infer(given, active);              (* forward generating inferences *)
✓   if given has a negative selected equality then
✓     add given to neg_equal;
✓     add to unprocessed all conclusions of extensionality resolution inferences
✓                                       between clauses in ext and given;
    add backward_infer(given, active) to unprocessed;  (* backward generating inferences *)
✓   if ext_rec(given) then
✓     add new to ext;
✓     add to unprocessed all conclusions of extensionality resolution inferences
✓                                       between given and clauses in neg_equal;
```

Figure 4.1: Otter Saturation Algorithm with Extensionality Resolution parts marked by ✓.

Forward Extensionality Resolution $given$ is the right premise and for every selected negative equality literal we retrieve matching active extensionality clauses from ext.

Backward Extensionality Resolution $given$ is the left premise, i.e. an extensionality clause, and we retrieve all active clauses with matching selected negative equality literals from neg_equal.

In general, every implementation of an inference with more than one premise has a forward and a backward part. Furthermore, the two version usually require very different indexing tech-

niques to retrieve candidates for the inference. Note that a clause can act as both premises of an inference. A typical example is self superposition. Hence $given$ is moved to $active$ before performing inferences.

Another addition to the saturation algorithm, not shown in Figure 4.1, is that deleted or simplified clauses belonging to any of the collections neg_equal or ext should be deleted from the collections too. An easy way to implement this is to ignore such deletions when they occur and instead check the storage class of a clause (that is active, passive, unprocessed or deleted) when we iterate through the collection during generating inferences. If during such an iteration we discover a clause that is no more active, we remove it from the collection and perform no generating inferences with it.

CHAPTER 5

Recognizing Extensionality Axioms

One of the key questions for building in extensionality reasoning in a theorem prover is the recognition of extensionality clauses, i.e. the concrete choice of ext_rec. Every clause containing a positive equality between two different variables $x = y$ is a potential extensionality clause. However, extensionality resolution should perform only few but essential inferences to facilitate a goal directed proof search for problems which need extensionality. Therefore, recognizing "too much" or "wrong" clauses as extensionality clauses will result in a quick blow up of the search space.

To understand the use and treatment of extensionality axioms, we analyzed first-order problems from the TPTP library [52] and derived according filter criteria for recognizing extensionality clauses, as reported below.

5.1 TPTP Library Analysis

It turned out that the TPTP library contains about 6,000 different axioms (mainly formulas, not clauses) that can result in a clause containing a positive equality among variables. By different here we mean up to variable renaming. One can consider other equivalence relations among axioms, such as using commutativity and associativity of $\wedge$ or $\vee$, or closure under renaming of predicate and function symbols, for which the number of different axioms will be smaller. Anyhow, having 6,000 different axioms in about 14,000 problems shows that such axioms are very common.

The most commonly used examples of extensionality axioms are the already discussed set and array extensionality axioms. In addition to them, set theory axiomatizations often contain the subset-based extensionality axiom $x \subseteq y \wedge y \subseteq x \rightarrow x = y$.

Contrary to these intended extensionality axioms, there is one kind of axioms which is dangerous to consider as extensionality: constructor axioms, describing that some function symbol is a constructor. Constructor axioms are central in theories of algebraic data types. For example,

consider an axiom describing a property of pairs

$$pair(x_1, x_2) = pair(y_1, y_2) \rightarrow x_1 = y_1,$$

or a similar axiom for the successor function

$$succ(x) = succ(y) \rightarrow x = y.$$

If we regard the latter as an extensionality axiom, extensionality resolution allows one to successively derive from any inequality $s \neq t$ the inequalities:

$$succ(s) \neq succ(t)$$
$$succ(succ(s)) \neq succ(succ(t))$$
$$\ldots$$

This will clutter the search space with bigger and bigger clauses. Hence, clauses derived from constructor axioms must not be recognized as extensionality clauses.

Another common formula is the definition of a non-strict order: $x \leq y \leftrightarrow x < y \vee x = y$. We did not yet investigate how considering this axiom as an extensionality axiom affects the search space, and consider such an investigation an interesting task for future work.

In addition to the above mentioned potential extensionality axioms, there is a large variety of such axioms in the TPTP library, including very long ones. One example, coming from the Mizar library, is:

$$\begin{array}{l}(\forall x_0)(\forall x_1)(\forall x_2)(\forall x_3)(\forall x_4)\\ \quad((v1\_funct\_1(x_1) \wedge v1\_funct\_2(x_1, k2\_zfmisc\_1(x_0, x_0), x_0) \wedge\\ \quad\quad m1\_relset\_1(x_1, k2\_zfmisc\_1(x_0, x_0), x_0) \wedge v1\_funct\_1(x_2) \wedge\\ \quad\quad v1\_funct\_2(x_2, k2\_zfmisc\_1(x_0, x_0), x_0) \wedge m1\_relset\_1(x_2, k2\_zfmisc\_1(x_0, x_0), x_0) \wedge\\ \quad\quad m1_s ubset_1(x_3, x_0) \wedge m1_s ubset_1(x_4, x_0)) \rightarrow\\ \quad\quad(\forall x_4)(\forall x_6)(\forall x_7)(\forall x_8)(\forall x_9)(\\ \quad\quad\quad g3\_vectsp\_1(x_0, x_1, x_2, x_3, x_4) = g3\_vectsp\_1(x_5, x_6, x_7, x_8, x_9) \rightarrow\\ \quad\quad\quad\quad(x_0 = x_5 \wedge x_1 = x_6 \wedge x_2 = x_7 \wedge x_3 = x_8 \wedge x_4 = x_9))).\end{array}$$

Another example comes from problems generated automatically by parsing natural language sentences:

$$\begin{array}{l}x_4 = x_6 \vee ssSkC0 \vee \neg in(x_6, x_7) \vee \neg front(x_7) \vee \neg furniture(x_7) \vee \neg seat(x_7) \vee\\ \neg fellow(x_6) \vee \neg man(x_6) \vee \neg young(x_6) \vee \neg seat(x_5) \vee \neg furniture(x_5) \vee \neg front(x_5) \vee\\ \neg in(x_4, x_5) \vee \neg young(x_4) \vee \neg man(x_4) \vee \neg fellow(x_4) \vee \neg in(x_2, x_3) \vee \neg city(x_3) \vee\\ \neg hollywood(x_3) \vee \neg event(x_2) \vee \neg barrel(x_2, x_1) \vee \neg down(x_2, x_0) \vee \neg old(x_1) \vee\\ \neg dirty(x_1) \vee \neg white(x_1) \vee \neg car(x_1) \vee \neg chevy(x_1) \vee \neg street(x_0) \vee \neg way(x_0) \vee\\ \neg lonely(x_0).\end{array}$$

5.2 Extensionality Recognizer Choices

Based on our analysis of potential extensionality axioms, we decided to generally exclude:

F1 clauses having more than one equality among variables, and

F2 clauses having a negative equality of the same sort as $x = y$.

The second criterion eliminates constructor axioms. However, in unsorted problems, i.e. every term has the same sort, we would for example also lose the array extensionality axiom.

In addition, we designed the following increasingly restrictive filter criteria.

F3 Exclude clauses having a positive equality other than $x = y$.

Consider for example the contrapositive of the select-over-store 2 axiom from Figure 3.2 and the clause

$$i = j \vee select(store(x, i, e), j) = select(x, j)$$

obtained by converting the axiom to CNF. The axiom is certainly not intended to be an extensionality axiom and recognizing the clause as extensionality clause would only distract the proof search.

F4 Exclude clauses exceeding some maximal length n, i.e. clauses having more than n literals.

Showing the actual effectiveness of a new automated reasoning technique is subject to experimental evaluation. Especially in first-order theorem proving, due to the high complexity, experiments are central to judge the usefulness of new techniques. To this end we implemented the filter criteria **F1**-**F4** as options and evaluated how different combinations and parameterizations influence the proof search, as described in Chapter 6.

Further ideas for filter criteria in extensionality recognizers, which we did not yet implement or evaluate, are the following.

- Only recognize clauses from the input as extensionality clauses.
- Only recognize a limited number of extensionality clauses. Similarly, the ratio of extensionality inferences compared to other inferences could be limited.
- Recognize extensionality axioms in input formulas before CNF transformation.
- Allow the user to declare axioms as extensionality or expose a pattern match language to specify extensionality patterns.

CHAPTER 6

Experimental Results

We implemented extensionality resolution in VAMPIRE. Our implementation required about 1,000 lines of C++ code on top of the existing VAMPIRE code. The extended VAMPIRE is available as binary at [1]. The extended implementation will be referred to as VAMPIRE^{EX} and will be merged in the next official release of VAMPIRE.

In this chapter we evaluate extensionality resolution by running it on the relevant part of the TPTP library [52], a number of handcrafted set theory problems, and SMT-LIB array problems [11], see Tables 6.1–6.4. In the TPTP library, VAMPIRE with extensionality resolution solves many problems not solved by VAMPIRE without it. On the set theory problems our prover significantly outperforms all theorem provers that were competing in the CASC-24 system competition of last year [53]. The extended VAMPIRE is the only prover that solves all the problems, while 17 out of 36 problems cannot be solved by any other prover, including the VAMPIRE without extensionality resolution. When evaluating VAMPIRE^{EX} on array problems taken from the SMT-LIB library, we observed that it solved 70% of the problems while VAMPIRE without extensionality resolution could only solve 40% of the problems.

The rest of this chapter describes in detail our experiments. Our results on analyzing extensionality axioms from the TPTP library were obtained using a cluster of dedicated servers at the University of Manchester with 2.3GHz quad core. We ran at most three problems on a server at a time, each run with 3GB memory limit and 60 seconds time limit, while each server has 16GB RAM. Further, our experiments on set and array examples were obtained on a GridEngine managed cluster system at IST Austria. Each run of a prover on a problem was assigned a dedicated 2.3 GHz core and 10 GB RAM with the time limit of 60 seconds.

6.1 TPTP Library Experiments

Based on the discussion of Chapter 5, we introduced three options to control the recognition of extensionality clauses in VAMPIRE^{EX}, namely `off`, `known` and `filter`. The option `off` does not recognize any extensionality clauses. The option `known` only recognizes clauses obtained from the set and array extensionality axioms and the subset-based set extensionality. The

Table 6.1: Experiments on TPTP problems with various options for recognizing extensionality clauses in VAMPIRE^{EX}.

options	solved	extensionality clauses		extensionality resolvents	
		max	avg	max	avg
`off`	2,909	0	0.0	0	0
`known`	2,914	5	0.7	13,090	90
`filter`(2,-)	2,695	1,120	5.8	280,000	1,428
`filter`(2,+)	2,691	1,120	6.1	280,000	1,515
`filter`(3,-)	2,623	1,123	8.6	280,750	1,765
`filter`(3,+)	2,623	1,123	8.7	280,750	1,806
`filter`(inf,-)	2,485	3,531	23.0	280,750	2,936
`filter`(inf,+)	2,475	3,843	23.2	280,750	3,001

option `filter` applies the criteria **F1** and **F2** given in Chapter 5 and allows further filtering by two other options corresponding to the criteria **F3** and **F4**. For the maximal length of the extensionality clause we used the values 2, 3 and ∞ for experiments.

We ran experiments on all 7033 TPTP problems that may contain an equality between variables. Our results are summarized in Table 6.1, where the first argument of `filter` is the limit on the length of extensionality clauses and the second argument is + if extensionality clauses are allowed to contain positive equalities other than $x = y$, - otherwise. All problems were run using the default strategy of VAMPIRE^{EX}, that is the default strategy of VAMPIRE in conjunction with extensionality resolution.

As one can see from Table 6.1, apart from the value `known`, any increase in the set of recognized extensionality clauses results in a decrease of the number of solved problems. This means nothing per se, since some very useful options (such as set of support), can drastically decrease the number of solved problems, but solve many problems that cannot be solved by any other strategy. The value of a new option is mainly in whether it can solve problems not solvable without it. Indeed, modern theorem provers treat hard problems with a cocktail of strategies. What we observed is that all problems solvable with the value `filter` were also solvable with other values. The use of the value `known` solves some very hard problems in nearly 0 seconds. The conclusion from our experiments is that it is best to have an option recognizing known extensionality axioms, which correlates with the original motivation of solving problems over collection types.

6.2 Set Theory Experiments

We handcrafted 36 set identity problems given in Figure 6.1, which also include the problems presented in Chapter 3. For proving the problems, we created TPTP files containing the set theory axioms from Figure 3.1 as TPTP axioms and the problem to be proven as a TPTP conjecture. As not all the axioms are needed in proving each of the problems, we also created TPTP files containing only the respectively relevant axioms. As a result, in our set theory experiments we used a total of 72 problems.

1. $x \cap y = y \cap x$
2. $x \cup y = y \cup x$
3. $x \cap y = ((x \cup y) - (x - y)) - (y - x)$
4. $\overline{(\overline{x})} = x$
5. $x = x \cap (x \cup y)$
6. $x = x \cup (x \cap y)$
7. $(x \cap y) - z = (x - z) \cap (y - z)$
8. $\overline{x \cup y} = \overline{x} \cap \overline{y}$
9. $\overline{x \cap y} = \overline{x} \cup \overline{y}$
10. $x \cup (y \cap z) = (x \cup y) \cap (x \cup z)$
11. $x \cap (y \cup z) = (x \cap y) \cup (x \cap z)$
12. $x \subseteq y \rightarrow x \cup y = y$
13. $x \subseteq y \rightarrow x \cap y = x$
14. $x \subseteq y \rightarrow x - y = \emptyset$
15. $x \subseteq y \rightarrow y - x = y - (x \cap y)$
16. $x \cup y \subseteq z \rightarrow z - (x \triangle y) = (x \cap y) \cup (z - (x \cup y))$
17. $x \triangle y = \emptyset \rightarrow x = y$
18. $z - (x \triangle y) = (x \cap (y \cap z)) \cup (z - (x \cup y))$
19. $(x - y) \cap (x \triangle y) = x \cap \overline{y}$
20. $x \triangle y = (x - y) \cup (y - x)$
21. $(x \triangle y) \triangle z = x \triangle (y \triangle z)$
22. $(x \triangle y) \triangle z = ((x - (y \cup z)) \cup (y - (x \cup z))) \cup ((z - (x \cup y)) \cup (x \cap (y \cap z)))$
23. $((x \cup y) \cap (\overline{x} \cup z)) = (y - x) \cup (x \cap z)$
24. $(\exists x)(((x \cup y) \cap (\overline{x} \cup z)) = y \cup z)$
25. $(x \cap y) \subseteq z \subseteq (x \cup y) \rightarrow ((x \cup y) \cap (\overline{x} \cup z)) = y \cup z$
26. $x \subseteq y \rightarrow (z - x) - y = z - y$
27. $x \subseteq y \rightarrow (z - y) - x = z - y$
28. $x \subseteq y \rightarrow z - (y \cup x) = z - y$
29. $x \subseteq y \rightarrow z - (y \cap x) = z - x$
30. $x \subseteq y \rightarrow (z - y) \cap x = \emptyset$
31. $x \subseteq y \rightarrow (z - x) \cap y = z \cap (y - x)$
32. $x \subseteq y \subseteq z \rightarrow (z - x) \cap y = y - x$
33. $x - y = x \cap \overline{y}$
34. $x \cap \emptyset = \emptyset$
35. $x \cup \emptyset = x$
36. $x \subseteq y \rightarrow (\exists z)(y - z = x)$

Figure 6.1: Collection of 36 handcrafted set theory problems. All variables without explicit quantification are universally quantified.

Tables 6.2 and 6.3 show the runtimes and the number of problems solved by VAMPIRE^{EX} compared to all but two provers participating in the first-order theorems (FOF) and typed first-order theorems (TFA) divisions of the CASC-24 competition.[1] The only provers whom we did not compare VAMPIRE^{EX} with were PROVER9 and SPASS+T, for the following reasons: PROVER9 depends on the directory structure of the CASC system and the TPTP library, thus it did not run on our test system; SPASS+T only accepts problems containing arithmetic. Since not all provers participating in CASC-24 support typed formulas, we have also generated untyped versions of the problems. As a result, theorem provers supporting typed formulas were then evaluated on both typed and untyped problems. In Table 6.2 we present our experiments obtained by evaluating theorem provers on the set theory axioms from Figure 3.1, whereas Table 6.3 reports on our results obtained by using only the respectively relevant axioms for each problem.

Our results show that only VAMPIRE^{EX} could solve all problems, and 22 problems could not be solved by any other prover. Moreover, VAMPIRE^{EX} is very fast: out of the 72 typed problems,

[1] We used the exact programs and command calls as in the competition, up to adaptions of the absolute file paths to our test system [1].

Table 6.2: Runtimes in seconds of provers on the set theory problems from Figure 6.1. Each problem includes the complete set theory axiomatization from Figure 3.1. The last row counts the number of solved problems.

	Untyped TPTP formulas										Typed TPTP formulas			
#	BEAGLE	CVC4	E-KR-HYPER	E	iPROVER	MUSCADET	PRINCESS	VAMPIRE	VAMPIRE^{EX}	ZIPPER-POSITION	BEAGLE	PRINCESS	VAMPIRE	VAMPIRE^{EX}
1					13.70	0.10	7.61		0.08			7.78		0.02
2		41.54					8.22		0.02			7.92		0.01
3									0.29					0.06
4		30.24		1.38	1.47	0.65	9.45	0.24	0.07			9.36	0.21	0.02
5		56.05		33.98	0.89	0.10	14.64		0.25			17.19	1.92	0.02
6		54.40			0.29		10.97		0.25			15.41		0.02
7									0.03					0.03
8									0.08					0.02
9									0.09					0.02
10									0.09					0.04
11									0.27					0.04
12		50.52			0.58		14.66	0.40	0.25			15.36	0.39	0.02
13		30.34		0.35	1.10	0.09	15.13	0.17	0.02			15.23	0.14	0.02
14	7.88			0.07	2.44	0.09	8.09	0.03	0.07	10.59	6.85	7.80	0.02	0.02
15		32.15		1.55	13.80		8.04	0.15	0.03			8.55	0.12	0.02
16									4.14					3.41
17		30.94		24.31				0.02	0.09	0.44			0.02	0.01
18									1.08					0.94
19									0.04					0.03
20									0.25					0.02
21									0.25					0.03
22									1.76					1.73
23									0.50					0.24
24								0.26	0.42				0.43	0.15
25									0.05					0.05
26									0.10					0.05
27				52.47	11.80		20.97		0.08			25.80		0.03
28		34.32			11.80		37.05	0.72	0.31			33.73	0.80	0.06
29		31.33		1.64	38.63			0.26	0.04				0.22	0.03
30				27.54	3.32	0.11	11.53	0.07	0.08	23.30		12.36	0.06	0.02
31									0.27					0.03
32									0.09					0.04
33					23.28		21.00		0.01			20.92		0.02
34	2.21	30.29		0.03	0.50	0.08	6.71	0.02	0.01	0.59	2.22	6.71	0.02	0.02
35		30.34		30.23	8.23		7.24	0.25	0.02			6.87	0.23	0.02
36		44.77			1.50			21.01	0.03				20.86	0.02
	2	13	0	11	16	7	15	13	36	4	2	15	14	36

only 9 took more than 0.01 seconds and only 4 took more than 1 second.[2] The total runtime of VAMPIRE^{EX} on all typed problems was 37.6 seconds. Among the problems also solved by VAMPIRE, VAMPIRE^{EX} is only slower on problem 24 of Table 6.3 (besides problems 12 and 27, on which the runtime difference is negligible). This is due to the selected proving strategy, which is not effective in this particular case.

[2] In our experiments with typed formulas, type information reduces the number of well-formed formulas and therefore the search space. Hence VAMPIRE^{EX} is generally faster on typed problems, in our experiments 4.1 seconds in total.

Table 6.3: Runtimes in seconds of provers on the set theory problems from Figure 6.1. Each problem includes only the respectively relevant axioms from Figure 3.1. The last row counts the number of solved problems.

	Untyped TPTP formulas										Typed TPTP formulas			
#	BEAGLE	CVC4	E-KR-HYPER	E	iPROVER	MUSCADET	PRINCESS	VAMPIRE	VAMPIRE^{EX}	ZIPPER-POSITION	BEAGLE	PRINCESS	VAMPIRE	VAMPIRE^{EX}
1		37.07	0.06	0.06	0.22	0.07	7.10	0.05	0.02	0.97		3.39	0.04	0.02
2		55.30	0.11	27.18	0.30	0.11	7.25	0.60	0.02			7.23	0.60	0.02
3						0.10			0.03					0.03
4	1.32	0.05	0.21	0.03	0.09	0.33	8.92	0.02	0.01	0.05	1.44	11.86	0.02	0.02
5		0.07		6.05	0.09	0.08	11.58	0.14	0.02	1.00		9.18	0.13	0.02
6		0.06		2.55	0.10	0.08	8.81	2.11	0.01			10.87	2.10	0.01
7						0.09			0.02					0.02
8									0.01					0.02
9									0.01					0.02
10						0.11			0.02					0.02
11						0.08			0.02					0.02
12		0.10	12.10	0.11	0.07	0.07	3.08	0.02	0.02	46.18		9.59	0.02	0.05
13		0.09		0.06	0.10	0.07	9.92	0.03	0.02	0.37		9.96	0.03	0.02
14	2.68	0.63	0.25	0.02	0.11	0.12	7.50	0.02	0.01	0.11	2.60	8.41	0.02	0.02
15		30.20		0.06	0.18	0.08	8.41	0.02	0.02			7.71	0.03	0.02
16						0.16			0.46					0.44
17	1.76	0.07	32.94	0.03	2.89	0.08	5.62	0.02	0.01	0.07	1.87	42.00	0.02	0.01
18						0.15			0.03					0.04
19									0.01					0.01
20					34.90	0.08			0.02					0.02
21						0.16			0.02					0.02
22						0.26			1.29					1.26
23									0.28					0.28
24				9.03				1.70	27.54				1.69	27.55
25									0.04					0.03
26					4.67	0.08			0.01					0.01
27				0.39	0.16	0.08	13.85	0.03	0.02	1.50		12.21	0.03	0.04
28		31.16		1.77	0.30	0.08	24.63	0.06	0.03			18.78	0.06	0.04
29		30.25		0.07	0.27	0.08	21.61	0.03	0.02	2.80		16.33	0.03	0.02
30				1.81	0.15	0.14	12.20	0.03	0.02	0.30		12.60	0.03	0.02
31					1.45	0.10	35.64		0.02			23.94		0.02
32		32.67			2.83	0.09	37.21		0.02			26.61		0.03
33		12.39					14.61		0.01			15.47		0.02
34	1.35	0.06	0.05	0.03	0.08	0.07	1.89	0.02	0.01	0.02	1.35	1.69	0.02	0.02
35	7.39	0.07	9.76	0.04	0.13	0.07	2.45	0.01	0.03	0.06	3.54	2.70	0.02	0.02
36		1.37		0.10	7.84			0.08	0.04	33.12			0.06	0.04
	5	17	8	18	21	28	19	18	36	13	5	19	18	36

6.3 Array Experiments

For evaluating VAMPIRE^{EX} on array problems, we used all the 278 unsatisfiable problems from the QF_AX category of quantifier-free formulas over the theory of arrays with extensionality of SMT-LIB. We translated the problems into the TPTP syntax. These problems belong to three problem classes from [3], namely they are instances of the parametric constructions *storecomm*, *swap* and *storeinv*. Table 6.4 reports on the results of VAMPIRE^{EX} on these problems and compares them to the results obtained by VAMPIRE. VAMPIRE^{EX} solves almost twice as much *storecomm* and *swap* problems in only half (respectively two-thirds) of the time. On *storeinv* problems only the runtime decreased slightly, but the same number of problems was solved.

Table 6.4: Evaluation of extensionality resolution on array problems. Runtimes are in seconds.

Problem class	**Number of problems**	VAMPIRE		VAMPIRE^{EX}	
		solved	runtime	solved	runtime
storecomm	105	47	5,161.90	87	2,466.52
swap	153	48	6,850.72	91	4,573.31
storeinv	20	15	339.55	15	315.23
Total	278	110	12,352.17	193	7,355.06

CHAPTER 7

Related Work

Reasoning with both theories and quantifiers is considered as a major challenge in the theorem proving and SMT communities. There is a large body of work on extending SMT-based methods to quantified formulas, as well as extending methods for full first-order logic with theory specific techniques. In this chapter we present some of this approaches and their relation to our work.

7.1 Satisfiability Modulo Theories

Modern SMT solvers are based on the DPLL(T) framework [39], a tight integration of a propositional SAT solver with satisfiability procedures for ground conjunctions of theory literals. The SAT solver is responsible for case splitting on the propositional part of a formula and considers knowledge of theory conflicts for further decisions.

SMT solvers can process very large formulas in ground decidable theories [24, 10]. Quantifier reasoning in SMT solvers is implemented using trigger-based E-matching. To this end, quantified formulas are annotated with patterns/triggers containing the quantified variables. Whenever a generated term matches the pattern modulo the current set of established equalities, the quantified formula is instantiated using the corresponding substitution. While this heuristic can be effective in certain situations, in general, it is incomplete, sensitive to the syntax of formulas and the state of the solver, and requires user guidance in the form of patterns. Hence, E-matching is not as powerful as the use of unification in superposition calculi. The work of [23] shows a tight integration of a SMT solver with superposition-based saturation, and [48] shows how to combine E-matching with free variable approaches.

Our work is dedicated to first-order reasoning about collections, such as sets and arrays. It is partially motivated by program analysis, since collection types are first-class types in many programming languages and nearly every programming languages has collection libraries. In [3] superposition with restricted orderings is shown to terminate on ground array formulas, i.e. constitutes a decision procedure. However, this was not extended to more general cases and is less efficient than reduction-based SMT solving [25]. The work of [22, 32] explores the limits

of decidability in the theory of arrays. The array property fragment of [22] can express, for example, that an array is sorted, or that two arrays are equal in a given range of indices. The satisfiability problem reduces to formulas in the combined theory of equality with uninterpreted functions (EUF), linear integer arithmetic, and the theory of array elements. The fragment in [32] allows relating consecutive array elements or specifying periodic facts. However, both fragments impose strict syntactic restrictions. E.g. no nested array reads, like $select(select(a, i), j)$, are allowed. The model based quantifier instantiation approach of [28] decides a class of formulas which subsumes, for example, the array property fragment. Our work is different since we consider collections in full first-order logic. Unlike [3], we impose no additional constraints on the used simplification ordering and can deal with arbitrary axioms on top of array axioms.

7.2 First-Order Theorem Proving

Unlike SMT solvers, first-order theorem provers are very efficient in handling quantifiers but weak in theory reasoning. Notoriously bad examples are theories with numeric domains, such as integer, rational, or real arithmetic. However, arithmetic is one of the most important theories in applications of program analysis and verification.

In the hierarchic theorem proving framework, a general-purpose "foreground" prover for full first-order logic cooperates with a specialized "background" theory prover to derive a refutation from a set of clauses. The work of [9] introduces the hierarchical superposition calculus by combining the superposition calculus with black-box style theory reasoning. The implementation in [43] combines the superposition theorem prover SPASS with SMT solvers for arithmetic. Since general completeness is impossible, a major goal in the hierarchic theorem proving approach is to have sufficient conditions for completeness. The internal process of clause abstraction, separating foreground and background symbols in formulas, may destroy one of the conditions given in [9]. To this end, in [12], a new form of clause abstraction is introduced, along with a completeness result for the fragment where all background-sorted terms are ground.

A different approach to first-order reasoning is the instantiation-based theorem proving method of [26, 36]. The idea is to generate quantifier-free instances of the first-order problem. These ground instances are passed to the reasoning engine of the background theory for proving unsatisfiability of the original quantified problem on the ground level. In case of satisfiability, the ground abstraction is refined based on the generated model and new instances are next generated. Similar to the theory of resolution, this framework provides methods for proving completeness of instantiation calculi, redundancy elimination criteria and saturation strategies. The current implementation in iPROVER [35] delegates ground reasoning to a propositional SAT solver and is complete for first-order logic. The work in [27] shows how to integrate an answer-complete theory solver into the instantiation framework. However, the practical impact of this approach is not yet well-understood.

Hierarchic superposition and instantiation-based reasoning separate the theory-specific and quantifier reasoning. This is not the case with our work, we introduce a new inference rule for extensionality reasoning and apply superposition reasoning in conjunction with this rule to prove quantified properties in full first-order theories. As many others, we are trying to bridge the gap between quantifier and theory reasoning, but in a way that is friendly to existing architectures

of first-order theorem provers. In a way, our approach is similar to the one of [21], where it is proposed to extend the resolution calculus by theory-specific rules, which do not change the underlying inference mechanisms. Indeed, our implementation of extensionality resolution requires relatively simple changes in saturation algorithms.

CHAPTER 8

Conclusion

We examined why reasoning with extensionality axioms is hard for superposition-based theorem provers and proposed a new inference rule, called *extensionality resolution*, to improve their performance on problems containing such axioms. Our experimental results show that first-order provers with extensionality resolution can easily solve problems in reasoning with sets and arrays that were unsolvable by all existing theorem provers and, also much harder versions of these problems. Our results contribute to one of the main problems in modern theorem proving: efficiently solving problems using both quantifiers and theories.

A possible direction for further research is to extend the applicability of extensionality resolution by combining it with other theory specific techniques. For example, in the theory of arrays, proving certain problems, including the ones in our evaluation, amounts to a huge case analysis according to the select-over-store axioms. Hence it would be interesting to incorporate SAT or SMT techniques to handle this combinatorial problem. Another possible direction would be to add extensionality resolution to the hierarchic superposition calculus. This is especially promising when arithmetical reasoning over array indices or array content is required. Finally, further rules for recognizing extensionality axioms, as suggested in Chapter 5, can be implemented and evaluated.

Preliminary results of this thesis have been submitted for publication [30].

Bibliography

[1] Experimental results of this thesis. `http://vprover.org/extres`.

[2] P. B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof*, volume 27 of *Applied Logic Series*. Springer, 2 edition, 2002.

[3] A. Armando, M. P. Bonacina, S. Ranise, and S. Schulz. New Results on Rewrite-Based Satisfiability Procedures. *ACM Transactions on Computational Logic*, 10(1), 2009.

[4] M. Baaz, U. Egly, and A. Leitsch. Normal Form Transformations. In *Handbook of Automated Reasoning*, volume I, chapter 5, pages 273–333. Elsevier and MIT Press, 2001.

[5] L. Bachmair and H. Ganzinger. Completion of First-Order Clauses with Equality by Strict Superposition. In *Proc. of CTRS*, pages 162–180, 1990.

[6] L. Bachmair and H. Ganzinger. On Restrictions of Ordered Paramodulation with Simplification. In *Proc. of CADE*, pages 427–441, 1990.

[7] L. Bachmair and H. Ganzinger. Rewrite-Based Equational Theorem Proving with Selection and Simplification. *Journal of Logic and Computation*, 4(3):217–247, 1994.

[8] L. Bachmair and H. Ganzinger. Resolution Theorem Proving. In *Handbook of Automated Reasoning*, volume I, chapter 2, pages 19–99. Elsevier and MIT Press, 2001.

[9] L. Bachmair, H. Ganzinger, and U. Waldmann. Refutational Theorem Proving for Hierarchic First-Order Theories. *Applicable Algebra in Engineering, Communication and Computing*, 5:193–212, 1994.

[10] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli. CVC4. In *Proc. of CAV*, pages 171–177, 2011.

[11] C. Barrett, A. Stump, and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2010.

[12] P. Baumgartner and U. Waldmann. Hierarchic Superposition with Weak Abstraction. In *Proc. of CADE*, pages 39–57, 2013.

[13] C. Benzmüller. *Equality and Extensionality in Higher-Order Theorem Proving*. PhD thesis, Naturwissenschaftlich-Technische Fakultät I, Saarland University, Saarbrücken, Germany, 1999.

[14] C. Benzmüller. Extensional Higher-Order Paramodulation and RUE-Resolution. In *Proc. of CADE*, pages 399–413, 1999.

[15] C. Benzmüller, C. Brown, and M. Kohlhase. Higher-Order Semantics and Extensionality. *Journal of Symbolic Logic*, 69(4):1027–1088, 2004.

[16] C. Benzmüller and M. Kohlhase. Extensional Higher-Order Resolution. In *Proc. of CADE*, pages 56–71, 1998.

[17] T. A. Beyene, C. Popeea, and A. Rybalchenko. Solving Existentially Quantified Horn Clauses. In *Proc. of CAV*, pages 869–882, 2013.

[18] A. Biere, M. J. H. Heule, H. van Maaren, and T. Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.

[19] N. Bjørner, K. L. McMillan, and A. Rybalchenko. On Solving Universally Quantified Horn Clauses. In *Proc. of SAS*, pages 105–125, 2013.

[20] R. Blanc, A. Gupta, L. Kovács, and B. Kragl. Tree Interpolation in Vampire. In *Proc. of LPAR*, pages 173–181, 2013.

[21] W. W. Bledsoe and R. S. Boyer. Computer Proofs of Limit Theorems. In *Proc. of IJCAI*, pages 586–600, 1971.

[22] A. Bradley, Z. Manna, and H. Sipma. What's Decidable About Arrays? In *Proc. of VMCAI*, pages 427–442, 2006.

[23] L. de Moura and N. Bjørner. Engineering DPLL(T) + Saturation. In *Proc. of IJCAR*, pages 475–490, 2008.

[24] L. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Proc. of TACAS*, pages 337–340, 2008.

[25] L. de Moura and N. Bjørner. Generalized, Efficient Array Decision Procedures. In *Proc. of FMCAD*, pages 45–52, 2009.

[26] H. Ganzinger and K. Korovin. New Directions in Instantiation-Based Theorem Proving. In *Proc. of LICS*, pages 55–64, 2003.

[27] H. Ganzinger and K. Korovin. Theory Instantiation. In *Proc. of LPAR*, pages 497–511, 2006.

[28] Y. Ge and L. de Moura. Complete Instantiation for Quantified Formulas in Satisfiabiliby Modulo Theories. In *Proc. of CAV*, pages 306–320, 2009.

[29] S. Grebenshchikov, N. P. Lopes, C. Popeea, and A. Rybalchenko. Synthesizing software verifiers from proof rules. In *Proc. of PLDI*, pages 405–416, 2012.

[30] A. Gupta, L. Kovács, B. Kragl, and A. Voronkov. Extensional Crisis and Proving Identity. Under submission, 2014.

[31] A. Gupta, C. Popeea, and A. Rybalchenko. Predicate Abstraction and Refinement for Verifying Multi-Threaded Programs. In *Proc. of POPL*, pages 331–344, 2011.

[32] P. Habermehl, R. Iosif, and T. Vojnar. What Else Is Decidable about Integer Arrays? In *Proc. of FoSSaCS*, pages 474–489, 2008.

[33] J. Y. Halpern, R. Harper, N. Immerman, P. G. Kolaitis, M. Y. Vardi, and V. Vianu. On the Unusual Effectiveness of Logic in Computer Science. *Bulletin of Symbolic Logic*, 7(2):213–236, 2001.

[34] K. Hoder and N. Bjørner. Generalized Property Directed Reachability. In *Proc. of SAT*, pages 157–171, 2012.

[35] K. Korovin. iProver - An Instantiation-Based Theorem Prover for First-Order Logic (System Description). In *Proc. of IJCAR*, pages 292–298, 2008.

[36] K. Korovin. Inst-Gen – A Modular Approach to Instantiation-Based Automated Reasoning. In *Programming Logics – Essays in Memory of Harald Ganzinger*, pages 239–270, 2013.

[37] L. Kovács and A. Voronkov. First-Order Theorem Proving and Vampire. In *Proc. of CAV*, pages 1–35, 2013.

[38] K. L. McMillan and A. Rybalchenko. Solving Constrained Horn Clauses using Interpolation. Technical Report MSR-TR-2013-6, Microsoft Research, 2013.

[39] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *Journal of the ACM*, 53(6):937–977, 2006.

[40] R. Nieuwenhuis and A. Rubio. Paramodulation-Based Theorem Proving. In *Handbook of Automated Reasoning*, volume I, chapter 7, pages 371–443. Elsevier and MIT Press, 2001.

[41] A. Nonnengart and C. Weidenbach. Computing Small Clause Normal Forms. In *Handbook of Automated Reasoning*, volume I, chapter 6, pages 335–367. Elsevier and MIT Press, 2001.

[42] D. Pastre. MUSCADET: An Automatic Theorem Proving System Using Knowledge and Metaknowledge in Mathematics. *Artificial Intelligence*, 38(3):257–318, 1989.

[43] V. Prevosto and U. Waldmann. SPASS+T. In *Proc. of ESCoR*, pages 18–33, 2006.

[44] I. V. Ramakrishnan, R. C. Sekar, and A. Voronkov. Term Indexing. In *Handbook of Automated Reasoning*, volume II, chapter 26, pages 1853–1964. Elsevier and MIT Press, 2001.

[45] A. Riazanov and A. Voronkov. Limited Resource Strategy in Resolution Theorem Proving. *Journal of Symbolic Computation*, 36(1-2):101–115, 2003.

[46] J. A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the ACM*, 12(1):23–41, 1965.

[47] J. A. Robinson and A. Voronkov, editors. *Handbook of Automated Reasoning (in 2 volumes)*. Elsevier and MIT Press, 2001.

[48] P. Rümmer. E-Matching with Free Variables. In *Proc. of LPAR*, pages 359–374, 2012.

[49] P. Rümmer, H. Hojjat, and V. Kuncak. Classifying and Solving Horn Clauses for Verification. In *Proc. of VSTTE*, pages 1–21, 2013.

[50] P. Rümmer, H. Hojjat, and V. Kuncak. Disjunctive Interpolants for Horn-Clause Verification. In *Proc. of CAV*, pages 347–363, 2013.

[51] S. Schulz. System Description: E 1.8. In *Proc. of LPAR*, pages 735–743, 2013.

[52] G. Sutcliffe. The TPTP Problem Library and Associated Infrastructure. *Journal of Automated Reasoning*, 43(4):337–362, 2009.

[53] G. Sutcliffe and C. Suttner. The State of CASC. *AI Communications*, 19(1):35–48, 2006.