Extensional Crisis and Proving Identity*

Ashutosh Gupta[1], Laura Kovács[2], Bernhard Kragl[1,3], and Andrei Voronkov[4]

[1] IST Austria
[2] Chalmers University of Technology
[3] Vienna University of Technology
[4] The University of Manchester

Abstract. Extensionality axioms are common when reasoning about data collections, such as arrays and functions in program analysis, or sets in mathematics. 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 paper we show how addition of a new inference rule, called extensionality resolution, allows first-order theorem provers to easily solve problems no modern first-order theorem prover can solve. We illustrate this by running the VAMPIRE theorem prover with extensionality resolution on a number of set theory and array problems. Extensionality resolution helps VAMPIRE to solve problems from the TPTP library of first-order problems that were never solved before by any prover.

1 Introduction

Software verification involves reasoning about data collections, such as arrays, sets, and functions. Many modern programming languages support native collection types or have standard libraries for collection types. Many interesting properties of collections are expressed using both quantifiers and theory specific predicates and functions. Unless these properties fall into a decidable theory supported by existing satisfiability modulo theories (SMT) solvers or theorem provers, verifying them requires a combination of reasoning with quantifiers and collection-specific reasoning.

For proving properties of collections one often 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. To prove this theorem, in addition to using the definition of the union operation (see Section 2), one needs to use the property that sets containing the same elements are equal. This property is asserted by the extensionality axiom of set theory.

* This research was supported in part by the European Research Council (ERC) under grant agreement 267989 (QUAREM), the Swedish VR grant D0497701, the Austrian National Research Network RiSE (FWF grants S11402-N23 and S11410-N23), and the WWTF PROSEED grant ICT C-050.

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. The technical details of why it is so are presented in the next section. To overcome this limitation, we need specialized methods of reasoning with extensionality, preferably those not requiring radical changes in the underlying inference mechanism and implementation of superposition.

In this paper we present a new inference rule, called *extensionality resolution*, which allows first-order theorem provers to easily solve problems no modern first-order theorem prover can solve (Section 3). Our approach requires no substantial changes in the implementation 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 analyze various forms of extensionality axioms and describe various choices made, and corresponding options, for extensionality resolution (Section 4).

We implemented our approach in the first-order theorem prover VAMPIRE [15] and evaluated our method on a number of challenging examples from set theory and reasoning about arrays (Section 5). 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, including VAMPIRE without extensionality resolution. In particular, we found 12 problems from the TPTP library of first-order problems [21] that were never proved before by any existing prover in any previous edition of the CASC world championship for automated theorem proving [22].

2 Motivating Examples

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

We assume some basic understanding of first-order theorem proving and the superposition calculus, see, e.g. [3, 16] or [15]. Throughout this paper we denote the equality predicate by $=$ and the empty clause by $\square$. We write $s \neq t$ to mean $\neg(s = t)$, and similarly for every binary predicate written in infix notation. Superposition calculi deal with selection functions: in every non-empty clause at least one literal is selected. Unlike [3], we impose no restrictions on literal selection.

Set Theory. We start with an axiomatization of set theory and will refer to this axiomatization in the rest of the paper. 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 2. We denote set variables by x, y, z and set elements by e.

Example 1. The commutativity of union is a valid property of sets and a logical consequence of the set theory axiomatization:

$$(\forall x)(\forall y)\ x \cup y = y \cup x. \tag{1}$$

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

Fig. 1. Set Theory Axiomatization.

This identity is problem 2 in our problem suite of Section 5. 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 [22], that is Vampire [15], E [20] and iProver [14], 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 2. Consider the following conditional identity:

$$(\forall x)(\forall y)(\forall z)(x \cap y \subseteq z \wedge z \subseteq x \cup y \rightarrow (x \cup y) \cap (\overline{x} \cup z) = y \cup z) \tag{2}$$

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 Section 5.

It is not hard to analyze the reason for the failure of superposition provers for examples requiring extensionality, such as Example 2: 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 2 is:

$$f(x, y) \notin x \vee f(x, y) \notin y \vee x = y. \tag{3}$$

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). 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 derive a new set identity from already proved membership literals $s \in t$ 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.

Arrays. We now give an example of extensionality reasoning over arrays. The standard axiomatization of the theory of arrays also contains an extensionality axiom of the form

$$(\forall x)(\forall y)((\forall i)\ select(x, i) = select(y, i)) \rightarrow x = y), \tag{4}$$

where x, y denote array variables, i is an array index and $select$ the standard select/read function over arrays. 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 3. 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:

$$i_1 \neq i_2 \rightarrow store(store(a, i_1, v_1), i_2, v_2) = store(store(a, i_2, v_2), i_1, v_1). \tag{5}$$

Here, $store$ is the standard store/write function over arrays.

Again, this problem (and similar problems for a larger number of updates) is very hard for theorem provers, see Section 5. The explanation of why it is hard is the same as for sets: the extensionality axiom is used in "the wrong direction" because the literal $x = y$ in axiom (4) is never selected.

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.

The above discussion may suggest 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

$$e \in x - y \vee e \in x \vee e \notin y, \tag{6}$$

obtained by converting the set difference axiom of Figure 2 into CNF and suppose that the first literal is selected in it. A superposition step from the extensionality clause (3) into this clause gives

$$f(x-y,z) \notin x-y \vee f(x-y,z) \notin z \vee e \in z \vee e \in x \vee e \notin y. \tag{7}$$

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) will result in a rapid blow-up of the search space by large clauses. The solution we propose and defend in this paper 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.

3 Reasoning in First-Order Theories with Extensionality Axioms

In this section 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.

Suppose that we have a partial function ext_rec, called *extensionality recognizer*, such that for every clause C, $ext\_rec(C)$ either is undefined, or returns the single positive equality among variables $x = y$ from C. We will also sometimes use ext_rec as a boolean function, meaning that it is true iff it is defined. We call an *extensionality clause* any clause C for which $ext\_rec(C)$ holds. Note that every clause derived from an extensionality axiom contains a single positive equality among variables, but in general not every clause containing such an equality corresponds to an extensionality axiom, see Section 4.

The *extensionality resolution* rule is the following inference rule:

$$\frac{x = y \vee C \quad s \neq t \vee D}{C\theta \vee D}\ , \tag{8}$$

where

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

Note that, since equality is symmetric, there are two inferences between the premises of (8); one is given above and the other one is with the substitution $\{x \mapsto t, y \mapsto s\}$.

Example 4. Consider two clauses: clause (3) and the unit clause $a \cup b \neq b \cup a$. Suppose that the former clause is recognized as an extensionality clause. Then the following inference is an instance of extensionality resolution:

$$\frac{f(x,y) \notin x \vee f(x,y) \notin y \vee x = y \quad a \cup b \neq b \cup a}{f(a \cup b, b \cup a) \notin a \cup b \vee f(a \cup b, b \cup a) \notin b \cup a}\ .$$

```
input: $init$: set of clauses;
var $active, passive, unprocessed := \varnothing$: set of clauses;
var $given, new$: clause;
$unprocessed\ := init$;
loop
  while $unprocessed \neq \varnothing$
    $new := pop(unprocessed)$;
    if $new = \Box$ then return $unsatisfiable$;
    if $retained(new)$ then                                   (* retention test *)
      simplify $new$ by clauses in $active \cup passive$ ;      (* forward simplification *)
      if $new = \Box$ 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 $passive = \varnothing$ 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 *)
```

Fig. 2. Otter Saturation Algorithm.

Given a clause with a selected literal $s \neq t$, 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 1.

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 [15]. For an overview of saturation algorithms we refer to [18, 15].

A simplified description of the Otter saturation algorithm is shown in Figure 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. The algorithms 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 $active$.

All operations performed by the saturation algorithm that may take considerable time to execute are normally implemented using *term indexing*, that is, by building a

```
  input: init: set of clauses;
  var active, passive, unprocessed := $\varnothing$: set of clauses;
  var given, new: clause;
✓var neg_equal, ext := $\varnothing$: set of clauses;
  unprocessed := init;
  loop
    while unprocessed $\neq \varnothing$
      new := pop(unprocessed);
      if new $= \square$ then return unsatisfiable;
      if retained(new) then                                          (* retention test *)
        simplify new by clauses in active $\cup$ passive ;               (* forward simplification *)
        if new $= \square$ 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 $= \varnothing$ 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 given to ext;
✓     add to unprocessed all conclusions of extensionality resolution inferences
✓                                          between given and clauses in neg_equal;
```

Fig. 3. Otter Saturation Algorithm with Extensionality Resolution parts marked by ✓.

special purpose index data structure that makes the operation faster. For example, all theorem provers with built-in equality reasoning have an index for forward demodulation.

Extensionality resolution is a generating inference rule, so the relevant lines of the saturation algorithm 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 3.

As one can see from the algorithm in Figure 3, 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. Another addition to the saturation algorithm, not shown in Figure 3, is that deleted or simplified clauses belonging to any of these collections 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.

4 Recognizing Extensionality Axioms

One of the key questions for building extensionality reasoning into a theorem prover is the recognition of extensionality clauses, i.e. the concrete choice of ext_rec. Every clause containing a single positive equality between two different variables $x = y$ is a potential extensionality clause.

To understand this, we analyzed problems in the TPTP library of about 14,000 first-order problems [21] . 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 derive from any inequality $s \neq t$ the inequality $succ(s) \neq succ(t)$, which, in turn, allows one to derive $succ(succ(s)) \neq succ(succ(t))$ and so on. This will clutter the search space with bigger and bigger clauses. Hence, clauses derived from constructor axioms must not be recognized as extensionality clauses. We achieve this by excluding clauses having a negative equality of the same sort as $x = y$. However, in unsorted problems, i.e. every term has the same sort, we would for example also lose the array extensionality axiom.

The clause $i = j \vee select(store(x, i, e), j) = select(x, j)$ from the axiomatization of arrays is also certainly not intended to be an extensionality axiom. From this example we derive the option to exclude clauses having a positive equality other than the one among variables.

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

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

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

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

These examples give rise to an option for limiting the number of literals in an extensionality clause.

Based on our analysis in this section, there are a number of options for recognizing extensionality clauses. In Section 5 we show two combinations of these options are useful for solving distinct problems.

5 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] and will be merged in the next official release of VAMPIRE. In the sequel, we refer to our extended VAMPIRE implementation as VAMPIRE$^{\text{EX}}$.

In this section we report on our experimental results obtained by evaluating extensionality resolution on three collections of benchmarks: (i) handcrafted hard set theory problems, (ii) array problems from the SMT-LIB library [6], and (iii) first-order problems of the TPTP library [21]. Our results are summarized in Tables 1–3, and detailed below.

On the set theory problems our implementation significantly outperforms all theorem provers that were competing in the last year's theorem proving system competition CASC-24 [22]. VAMPIRE$^{\text{EX}}$ efficiently solves all the set theory problems, while every other prover including the original VAMPIRE solves less than half of the problems (Table 1). We also tried the SMT solver Z3, which failed to prove any of our set theory problems.

When evaluating VAMPIRE$^{\text{EX}}$ on array problems taken from the SMT-LIB library, VAMPIRE$^{\text{EX}}$ solved more problems than all existing first-order theorem provers (Table 2). The SMT solver Z3 outperformed VAMPIRE$^{\text{EX}}$ if we encode these array problems as problems from the theory of arrays with extensionality, in which case Z3 can use its decision procedure for this theory.

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. $x \subseteq y \rightarrow (\exists z)(y - z = x)$

Fig. 4. Collection of 36 handcrafted set theory problems. All variables without explicit quantification are universally quantified.

On the TPTP library, VAMPIRE$^{\text{EX}}$ solved 84 problems not solved by the CASC version of VAMPIRE (Table 3). Even more, 12 of these problems have rating 1, which means that no existing prover, either first-order or SMT, can solve them.

The rest of this section describes in detail our experiments. All results 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.

Set Theory Experiments. We handcrafted 36 set identity problems given in Figure 4, which also include the problems presented in Section 2. For proving the problems, we created TPTP files containing the set theory axioms from Figure 2 as TPTP axioms and the problem to be proved as a TPTP conjecture.

Table 1 shows the runtimes and the number of problems solved by VAMPIRE$^{\text{EX}}$ 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 which we did not compare with VAMPIRE$^{\text{EX}}$ 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

[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.

Table 1. Runtimes in seconds of provers on the set theory problems from Figure 4. Empty entries mean timeout after 60 seconds. The first row indicates whether the prover was run on typed (TFF) or untyped (FOF) problems. The last row counts the number of solved problems.

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

a result, theorem provers supporting typed formulas were then evaluated on both typed and untyped problems.

Our results show that only Vampire$^{\text{EX}}$ could solve all problems, and 17 problems could not be solved by any other prover. Moreover, Vampire$^{\text{EX}}$ is very fast: out of the 36 typed problems, only 5 took more than 0.1 seconds and only 2 took more than 1 second.

In our experiments with typed formulas, type information reduces the number of well-formed formulas and therefore the search space. Hence Vampire$^{\text{EX}}$ is generally faster on typed problems, in our experiments by 4.18 seconds in total. The total run-

Table 2. Evaluation of extensionality resolution on array problems. Runtimes are in seconds.

Prover	solved	runtime
VAMPIRE$^{\text{EX}}$	154	1193.85
VAMPIRE	107	1020.76
E	81	600.01
BEAGLE	16	185.44
ZIPPERPOSITION	15	49.27
PRINCESS	10	35.02
iPROVER	9	47.13
CVC4	8	0.36
E-KRHYPER	8	1.26
MUSCADET	4	0.41
Z3	277	64.25

time of VAMPIRE$^{\text{EX}}$ on all typed problems was 7.33 seconds. Among the problems also solved by VAMPIRE, VAMPIRE$^{\text{EX}}$ is always faster.

Finally, Table 1 does not compare VAMPIRE$^{\text{EX}}$ with SMT solvers for the reason that these set theory problems use both quantifiers and theories. We however note that the use of quantifiers in the set theory axiomatization caused SMT solvers, in particular Z3, to fail on all these examples. Z3 provides a special encoding [11] for sets that allows some of the problems to be encoded as quantifier free and we believe that a comparison with this encoding is unfair.

Array Experiments. For evaluating VAMPIRE$^{\text{EX}}$ 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 these problems into the TPTP syntax. Table 2 reports on the results of VAMPIRE$^{\text{EX}}$ on these problems and compares them to the results obtained by the other first-order provers and the SMT solver Z3, which solves all of them but one using a decision procedure for the theory. However, we feel that arrays with extensionality are not very interesting for applications, since we failed to find natural examples of problems that require such extensionality, apart from those that state that the results of updating arrays at distinct indexes does not depend on the order of updates (for example, all problems in the QF_AX category of SMT-LIB are such problems).

For array experiments we were interested whether VAMPIRE$^{\text{EX}}$ can outperform first-order provers without extensionality resolution. Table 2 shows that the number of array problems it solves is significantly larger than that of all other first-order provers, thus confirming the power of our approach to extensionality reasoning.

The TPTP Library Experiments. VAMPIRE uses a collection of strategies to prove hard problems and our new inference rule adds new possible options in the repertoire of VAMPIRE. Based on the discussion of Section 4, we introduced two new options for the VAMPIRE strategies to control the recognition of extensionality clauses in VAMPIRE$^{\text{EX}}$, namely `known` and `all`. The option `known` only recognizes clauses

Table 3. Experiments with various options for recognizing extensionality clauses in Vampire$^{\text{EX}}$.

Strategies	solved	uniquely solved	
`original`	4015	156	
`original+known`	3870	8	84
`original+all`	3747	50	

obtained from the set and array extensionality axioms, as well as the subset-based set extensionality axiom. The option `all` applies the criteria given in Section 4.

We ran experiments on all 7224 TPTP problems that may contain an equality between variables. Our results are summarized in Table 3, where the first row reports on using Vampire$^{\text{EX}}$ with the original collection of strategies of Vampire. The second row uses Vampire$^{\text{EX}}$ in the combination of the option `known` and the original strategies of Vampire, whereas the third row uses the option `all` with the original strategies of Vampire.

The `original` strategies solved 4015 problems, and 156 were uniquely solved by these collection of strategies. The `original+known` and `original+all` solved 3870 and 3747 problems, respectively. They uniquely solved 8 and 50 problems respectively. Using however `original+known` and `original+all` in combination, Vampire$^{\text{EX}}$ solved 84 problems which were not solved by the original collection of strategies `original`. We have listed these 84 problems in Table 4. Out of these 84 solved problems, 12 problems are rated with difficulty 1 in the CASC system competition. That is, these 12 problems were never solved in any previous CASC competition by any existing prover, including all existing first-order provers and the SMT solvers Z3 and CVC4 [5]. Vampire$^{\text{EX}}$ hence outperforms all modern solvers when it comes to reasoning with both theories and quantifiers.

Note that for first-order theorem provers the average number of problems solved by a strategy does not mean much in general. The reason is that these provers show the best performance when they treat problems by a cocktail of strategies. Normally, if a problem is solvable by a prover, there is a strategy that solves it in nearly no time, so running many short-lived strategies gives better results than running a small number of strategies for longer times. When we introduce a new option to a theorem prover, the main question is, if this option can complement the cocktail of strategies so that more problems are solved by these strategies all together. This means that an option that solves many unique problems is, in general, much more valuable than an option solving many problems on the average.

Our results indicate that the use extensionality resolution in first-order theorem proving can solve a significant number of problems not solvable without it. Therefore it is a powerful addition to the toolbox of first-order theorem proving methods. Further extensive experiments with combining extensionality resolution with various other options are required for better understanding of how it can be integrated in first-order provers.

Table 4. Fields and ratings of TPTP problems only solved by VAMPIRE with extensionality resolution.

Field	Subfield	Problem	Rating
Computer Science	Commonsense Reasoning	CRS075+6	0.97
Computer Science	Commonsense Reasoning	CRS076+2	0.97
Computer Science	Commonsense Reasoning	CRS076+6	1.00
Computer Science	Commonsense Reasoning	CRS076+7	1.00
Computer Science	Commonsense Reasoning	CRS078+2	1.00
Computer Science	Commonsense Reasoning	CRS079+6	0.97
Computer Science	Commonsense Reasoning	CRS080+1	0.97
Computer Science	Commonsense Reasoning	CRS080+2	1.00
Computer Science	Commonsense Reasoning	CRS081+4	0.93
Computer Science	Commonsense Reasoning	CRS083+4	0.90
Computer Science	Commonsense Reasoning	CRS083+6	0.97
Computer Science	Commonsense Reasoning	CRS084+2	0.93
Computer Science	Commonsense Reasoning	CRS084+4	0.93
Computer Science	Commonsense Reasoning	CRS084+5	0.93
Computer Science	Commonsense Reasoning	CRS088+1	0.97
Computer Science	Commonsense Reasoning	CRS088+2	1.00
Computer Science	Commonsense Reasoning	CRS088+4	0.93
Computer Science	Commonsense Reasoning	CRS088+6	1.00
Computer Science	Commonsense Reasoning	CRS089+6	1.00
Computer Science	Commonsense Reasoning	CRS092+6	1.00
Computer Science	Commonsense Reasoning	CRS093+4	0.93
Computer Science	Commonsense Reasoning	CRS093+6	1.00
Computer Science	Commonsense Reasoning	CRS093+7	1.00
Computer Science	Commonsense Reasoning	CRS094+6	0.97
Computer Science	Commonsense Reasoning	CRS109+6	0.93
Computer Science	Commonsense Reasoning	CRS118+6	0.97
Computer Science	Commonsense Reasoning	CSR057+5	0.97
Computer Science	Software Creation	SWC021-1	0.64
Computer Science	Software Creation	SWC160-1	0.93
Computer Science	Software Verification	SWV474+1	0.83
Computer Science	Software Verification	SWV845-1	0.86
Computer Science	Software Verification Continued	SWW284+1	0.87
Logic	Combinatory Logic	COL081-1	0.64
Mathematics	Algebra/Lattices	LAT298+1	0.90
Mathematics	Algebra/Lattices	LAT324+1	0.80
Mathematics	Category Theory	CAT009-1	0.00
Mathematics	Category Theory	CAT010-1	0.00
Mathematics	Graph Theory	GRA007+1	0.60
Mathematics	Graph Theory	GRA007+2	0.63
Mathematics	Number Theory	NUM459+1	0.70
Mathematics	Number Theory	NUM493+1	0.87
Mathematics	Number Theory	NUM493+3	0.97
Mathematics	Number Theory	NUM495+1	0.60
Mathematics	Number Theory	NUM508+3	0.63
Mathematics	Number Theory	NUM515+1	1.00
Mathematics	Number Theory	NUM515+3	1.00
Mathematics	Number Theory	NUM517+3	0.70
Mathematics	Number Theory	NUM535+1	0.63
Mathematics	Number Theory	NUM542+1	0.83
Mathematics	Number Theory	NUM544+1	0.90
Mathematics	Set Theory	SET018+1	0.90
Mathematics	Set Theory	SET041-3	0.36
Mathematics	Set Theory	SET066-6	1.00
Mathematics	Set Theory	SET066-7	1.00
Mathematics	Set Theory	SET069-6	0.93
Mathematics	Set Theory	SET069-7	0.93
Mathematics	Set Theory	SET070-6	0.93
Mathematics	Set Theory	SET070-7	0.93
Mathematics	Set Theory	SET097-7	0.64
Mathematics	Set Theory	SET099+1	0.87
Mathematics	Set Theory	SET128-6	0.71
Mathematics	Set Theory	SET157-6	0.71
Mathematics	Set Theory	SET262-6	0.86
Mathematics	Set Theory	SET497-6	0.71
Mathematics	Set Theory	SET510-6	0.43
Mathematics	Set Theory	SET606+3	0.53
Mathematics	Set Theory	SET613+3	0.83
Mathematics	Set Theory	SET634+3	0.67
Mathematics	Set Theory	SET671+3	0.90
Mathematics	Set Theory	SET673+3	0.90
Mathematics	Set Theory	SET674+3	0.90
Mathematics	Set Theory	SET831-1	0.86
Mathematics	Set Theory	SET837-1	0.93
Mathematics	Set Theory Continued	SEU007+1	1.00
Mathematics	Set Theory Continued	SEU049+1	0.87
Mathematics	Set Theory Continued	SEU058+1	0.93
Mathematics	Set Theory Continued	SEU059+1	0.97
Mathematics	Set Theory Continued	SEU073+1	1.00
Mathematics	Set Theory Continued	SEU194+1	0.70
Mathematics	Set Theory Continued	SEU205+1	0.97
Mathematics	Set Theory Continued	SEU265+2	0.97
Mathematics	Set Theory Continued	SEU283+1	0.73
Mathematics	Set Theory Continued	SEU384+1	0.90
Social Sciences	Social Choice Theory	SCT162+1	0.87

6 Related Work

Reasoning with both theories and quantifiers is considered as a major challenge in the theorem proving and SMT communities. SMT solvers can process very large formulas in ground decidable theories [10, 5]. Quantifier reasoning in SMT solvers is implemented using trigger-based E-matching, which is not as powerful as the use of unification in superposition calculi. Combining quantifiers with theories based on SMT solving is described in [17, 19].

Unlike SMT reasoning, first-order theorem provers are very efficient in handling quantifiers but weak in theory reasoning. Paper [4] introduces the hierarchical superposition calculus by combining the superposition calculus with black-box style theory reasoning. This approach has been further extended in [7] for first-order reasoning modulo background theories under the assumption that theory terms are ground. A similar approach is also addressed in the instantiation-based theorem proving method of [12, 14], where quantifier-free instances of the first-order problem are generated. These ground instances are passed to the reasoning engine of the background theory for proving unsatisfiability of the original quantified problem. In case of satisfiability, the original problem is refined based on the generated ground model and new instances are next generated. All mentioned approaches separate the theory-specific and quantifier reasoning. This is not the case with our work, where theory reasoning using extensionality is a natural extension of the superposition calculus.

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. While there has been a considerable amount of work on deciding universal theories of collection types, including using superposition provers [2] and decidability or undecidability of their extensions [9, 13], our work is different since we consider collections in first-order logic with quantifiers. 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. Unlike [2], we impose no additional constraints on the used simplification ordering and can deal with arbitrary axioms on top of array axioms.

In a way, our approach is similar to the one of [8], 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.

7 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.

References

1. Experimental results of this paper. `http://vprover.org/extres`.
2. A. Armando, M. P. Bonacina, S. Ranise, and S. Schulz. New Results on Rewrite-Based Satisfiability Procedures. *ACM Trans. Comput. Log.*, 10(1), 2009.
3. L. Bachmair and H. Ganzinger. Resolution Theorem Proving. In *Handbook of Automated Reasoning*, pages 19–99. Elsevier and MIT Press, 2001.
4. L. Bachmair, H. Ganzinger, and U. Waldmann. Refutational Theorem Proving for Hierarchic First-Order Theories. *Appl. Algebra Eng. Commun. Comput.*, 5:193–212, 1994.
5. 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.
6. C. Barrett, A. Stump, and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2010.
7. P. Baumgartner and U. Waldmann. Hierarchic Superposition with Weak Abstraction. In *Proc. of CADE*, pages 39–57, 2013.
8. W. W. Bledsoe and R. S. Boyer. Computer Proofs of Limit Theorems. In *Proc. of IJCAI*, pages 586–600, 1971.
9. A. Bradley, Z. Manna, and H. Sipma. What's Decidable About Arrays? In *Proc. of VMCAI*, pages 427–442, 2006.
10. L. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Proc. of TACAS*, pages 337–340, 2008.
11. L. de Moura and N. Bjørner. Generalized, efficient array decision procedures. In *FMCAD*, pages 45–52. IEEE, 2009.
12. H. Ganzinger and K. Korovin. Theory Instantiation. In *Proc. of LPAR*, pages 497–511, 2006.
13. P. Habermehl, R. Iosif, and T. Vojnar. What Else Is Decidable about Integer Arrays? In *Proc. of FoSSaCS*, pages 474–489, 2008.
14. K. Korovin. iProver - An Instantiation-Based Theorem Prover for First-Order Logic (System Description). In *Proc. of IJCAR*, pages 292–298, 2008.
15. L. Kovács and A. Voronkov. First-Order Theorem Proving and Vampire. In *Proc. of CAV*, pages 1–35, 2013.
16. R. Nieuwenhuis and A. Rubio. Paramodulation-Based Theorem Proving. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 7, pages 371–443. Elsevier Science, 2001.
17. V. Prevosto and U. Waldmann. SPASS+T. In *Proc. of ESCoR*, pages 18–33, 2006.
18. A. Riazanov and A. Voronkov. Limited Resource Strategy in Resolution Theorem Proving. *J. of Symbolic Computation*, 36(1-2):101–115, 2003.
19. P. Rümmer. E-Matching with Free Variables. In *Proc. of LPAR*, pages 359–374, 2012.
20. S. Schulz. System Description: E 1.8. In *Proc. of LPAR*, pages 735–743, 2013.
21. G. Sutcliffe. The TPTP Problem Library and Associated Infrastructure. *J. Autom. Reasoning*, 43(4):337–362, 2009.
22. G. Sutcliffe and C. Suttner. The State of CASC. *AI Communications*, 19(1):35–48, 2006.