Unification with Abstraction and Theory Instantiation in Saturation-based Reasoning

Giles Reger, Martin Suda and Andrei Voronkov

September 13, 2017

Unification with Abstraction and Theory Instantiation in Saturation-based Reasoning *

Giles Reger[1], Martin Suda[2], and Andrei Voronkov[1,3,4]

[1] University of Manchester, Manchester, UK
[2] TU Wien, Vienna, Austria
[3] EasyChair

Abstract

This paper explores two new inference rules for reasoning with quantifiers and theories in a saturation-based first-order theorem prover. The focus here is on non-ground clauses, complementing our recent work on AVATAR modulo theories for ground theory reasoning. The current implementation focuses on complete theories, e.g. various versions of arithmetic, but we also sketch how to work with incomplete theories. The first new rule utilises theory constraint solving (an SMT solver) to perform reasoning within a clause to find an instance where we can remove one or more theory literals. This utilises the power of SMT solvers for theory reasoning with non-ground clauses, reasoning which is currently achieved by the addition of prolific theory axioms. The second new rule is *unification with abstraction* where the notion of unification is extended to introduce constraints where theory terms may not otherwise unify e.g. $p(2)$ may unify with $\neg p(x+1) \vee q(x)$ to produce $2 \not\simeq x+1 \vee q(x)$. This abstraction is performed lazily, as needed, to allow the superposition theorem prover to make as much progress as possible without the search space growing too quickly. Additionally, the first rule can be used to discharge the constraints introduced by the second. These rules were implemented within the Vampire theorem prover and experimental results show that they are useful for solving a considerable number of previously unsolved problems.

1 Introduction

Reasoning in quantifier-free first-order logic with theories, such as arithmetic, is hard. Reasoning with quantifiers and first-order theories is very hard. It is undecidable in general and Π^1_1-complete for many simple combinations, for example linear (real, rational or integer) arithmetic and uninterpreted functions [16]. At the same time such reasoning is essential to the future success of certain application areas, such as program analysis and software verification, that rely on quantifiers to, for example, express properties of objects, inductively defined data structures, the heap and dynamic memory allocation. This paper presents a new approach to theory reasoning with quantifiers that uses a SMT solver to do *local* theory reasoning within a clause and provides an extension of unification that avoids the need to explicitly separate theory and non-theory parts of clauses.

There are two directions of research in the area of reasoning with problems containing quantifiers and theories. The first is the extension of SMT solvers with *instantiation* heuristics such as E-matching [12, 9]. The second is the extension of first-order reasoning approaches with support for theory reasoning. There have been a number of varied attempts in this second direction with some approaches extending various calculi [3, 13, 16, 27, 2, 8, 7] or using a SMT solver to deal with the ground part of the problem [20, 21]. The last of these references, [21], is our previous work developing AVATAR

*This work was supported by EPSRC Grant EP/K032674/1. Martin Suda and Andrei Voronkov were partially supported by ERC Starting Grant 2014 SYMCAR 639270. Martin Suda was also partially supported by the Austrian research projects FWF S11403-N23 and S11409-N23. Andrei Voronkov was also partially supported by the Wallenberg Academy Fellowship 2014 - TheProSE. Part of this work was done when Andrei Voronkov was part-time employed by Chalmers University of Technology.

modulo theories and later we will discuss how this interacts with this new approach (they complement each other). A surprisingly effective approach to theory reasoning with first-order theorem provers is to simply add *theory axioms* (i.e. axioms from the theory of interest). Whilst this has no hope of being complete, it can be used to prove a large number of problems of interest. However, theory axioms can be highly prolific in saturation-based proof search and often swamp the search space with irrelevant consequences of the theory [22]. This combinatorial explosion prevents theory axioms from being useful in cases where *deep* theory reasoning is required. The aim of this paper is to provide a solution that allows for a combination of these approaches i.e. the integration with an SMT solver, the use of theory axioms, and the heuristic extension of the underlying calculi.

Our paper contains two main ideas and we start with examples (which we revisit later) to motivate and explain these ideas. The first idea is motivated by the observation that the theory part of a first-order clause might already be restricting the interesting instances of a clause, sometimes uniquely, and we can use this to produce simpler instances that are useful for proof search. For example, the first-order clause

$$14x \not\simeq x^2 + 49 \lor p(x)$$

has a single solution for x which makes the first literal false with respect to the underlying theory of arithmetic, namely $x = 7$. Therefore, every instance of this clause is a logical consequence of its single instance

$$p(7)$$

in the underlying theory. If we apply standard superposition rules to the original clause and a sufficiently rich axiomatisation of arithmetic, we will most likely end up with a very large number of logical consequences and never generate $p(7)$, or run out of space before generating it. For many clauses the solution will not be unique but can still provide useful instances, for example by taking the clause

$$7 \leq x \lor p(x)$$

and using the instance of

$$7 \leq 0 \lor p(0)$$

we can derive the clause

$$p(0).$$

This clause does not represent all solutions for $7 \leq x$, but it results in a clause with fewer literals. Moreover, this clause is ground and can be passed to an SMT solver (this is where this approach complements the work of AVATAR modulo theories).

Finally, there are very simple cases where this kind of approach can immediately find inconsistencies. For example, the clause

$$x \leq 0 \lor x \leq y$$

has instances making it false, for example $[x = 1, y = 0]$.

As explained in Section 3, these observations lead to the development of an instantiation rule that rewrites clauses to be in the form $T \to C$, where T is the theory part, and uses an SMT solver to find a substitution θ under which T is valid in the given theory, thus producing the instance $C\theta$. Which, in the case where $C = \bot$, can find general inconsistencies.

The second rule is related to the use of *abstraction*. By an *abstraction* we mean (variants of) the rule obtaining from a clase $C[t]$, where t is a non-variable term, a clause $x \not\simeq t \lor C[x]$, where x is a new variable. Abstraction is implemented in several theorem provers, including the previous version of our theorem prover VAMPIRE [18] used for experiments described in this paper.

Take, for example, the formula

$$(\forall x : int.\, p(2x)) \rightarrow p(10)$$

which is ARI189=1 from the TPTP library [31]. When negated and clausified, this formula gives two unit clauses

$$p(2x) \quad \text{and} \quad \neg p(10),$$

from which we can derive nothing without abstracting at least one of the clauses.

If we abstract $p(10)$ into $p(y) \vee y \not\simeq 10$ then a resolution step would give us $2x \not\simeq 10$ and simple evaluation would provide $x \not\simeq 5$, which is refutable by equality resolution. However, the abstraction step is necessary. Some approaches rely on *full abstraction* where theory and non-theory symbols are fully separated. This is unattractive for a number of reasons which we enumerate here:

1. A fully abstracted clause tends to be much longer, especially if the original clause contains deeply nested theory and non-theory symbols. Long clauses are problematic for resolution-based approaches as algorithms implementing techniques such as subsumption and subsumption resolution are typically exponential in the length of the clause. Additionally, applying inferences such as resolution to long clauses leads to even longer clauses. Getting rid of long clauses was one of the motivations of our previous AVATAR work on *clause splitting* [32]. However, the long clauses produced by abstraction will share variables, reducing the impact of AVATAR.

2. The AVATAR modulo theories approach [21] ensures that the first-order solver is only exploring part of the search space that is theory-consistent in its ground part (using a SMT solver to achieve this). This is a powerful technique but relies on ground literals remaining ground, even those that mix theory and non-theory symbols as SMT solvers are effective in the theory of uninterpreted functions. Full abstraction destroys such ground literals.

3. As mentioned previously, the addition of theory axioms can be effective for problems requiring shallow theory reasoning. Working with fully abstracted clauses forces us to make first-order reasoning to treat the theory part of a clause differently. This makes it difficult to take full advantage of theory axiom reasoning.

The final reason we chose not to fully abstract clauses in our work is that the main advantage of full abstraction for us would be that it deals with the above problem, but we have a solution which we believe solves this issue in a more satisfactory way, as confirmed by our experiments described in Section 5.

The second idea is to perform this abstraction *lazily*, i.e., only where it is required to perform inference steps. As described in Section 4, this involves extending unifications to produce theory constraints under which two terms will unify. As we will see, these theory constraints are exactly the kind of terms that can be handled easily by the instantiation technique introduced in our first idea.

As explained above, the contributions of this paper are

1. a new instantiation rule that uses an SMT solver to provide instances consistent with the underlying theory (Section 3),

2. an extension of unification that provides a mechanism to perform *lazy* abstraction, i.e., only abstracting as much as is needed, which results in clauses with theory constraints that can be discharged by the previous instantiation technique (Section 4),

3. an implementation of these techniques in the VAMPIRE theorem prover (described in Sections 3 and 4),

4. an experimental evaluation that demonstrate the effectiveness of these techniques both individually and in combination with the rest of the powerful techniques implemented within VAMPIRE (Section 5).

Next we will introduce the necessary background material.

2 Preliminaries and Related Work

First-Order Logic and Theories. We consider a many-sorted first-order logic with equality. A *signature* is a pair $\Sigma = (\Xi, \Omega)$ where Ξ is a set of *sorts* and Ω a set of *predicate* and *function* symbols with associated argument and return sorts from Ξ. *Terms* are of the form c, x, or $f(t_1, \ldots, t_n)$ where f is a *function symbol* of arity $n \geq 1$, $t_1, \ldots, t_n$ are terms, c is a zero arity function symbol (i.e. a constant) and x is a variable. We assume that all terms are well-sorted. Atoms are of the form $p(t_1, \ldots, t_n), q$ or $t_1 \simeq_s t_2$ where p is a predicate symbol of arity n, $t_1, \ldots, t_n$ are terms, q is a zero arity predicate symbol and for each sort $s \in \Xi$, $\simeq_s$ is the *equality symbol for the sort* s. We write simply $\simeq$ when s is known from the context or irrelevant. A *literal* is either an atom A, in which case we call it *positive*, or a negation of an atom $\neg A$, in which case we call it *negative*. When L is a negative literal $\neg A$ and we write $\neg L$, we mean the positive literal A. We write negated equalities as $t_1 \not\simeq t_2$. We write $t[s]_p$ and $L[s]_p$ to denote that a term s occurs in a term t (in a literal L) at a position p.

A *clause* is a disjunction of literals $L_1 \vee \ldots \vee L_n$ for $n \geq 0$. We disregard the order of literals and treat a clause as a multiset. When $n = 0$ we speak of the *empty clause*, which is always false. When $n = 1$ a clause is called a *unit clause*. Variables in clauses are considered to be universally quantified. Standard methods exist to transform an arbitrary first-order formula into clausal form (e.g. [19] and our recent work in [25]). If C is a clause $L_1 \vee \ldots \vee L_n$, we denote by $\neg C$ the conjunction of literals $\neg L_1 \wedge \ldots \wedge \neg L_n$.

A *substitution* is any expression θ of the form $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$, where $n \geq 0$. $E\theta$ is the expression obtained from E by the simultaneous replacement of each x_i by t_i. By an expression here we mean a term, an atom, a literal, or a clause. An expression is *ground* if it contains no variables. We will call the expression $E\theta$ the θ*-instance* of E. An *instance of* E is any expression $E\theta$ and a *ground instance* of E is any instance of E that is ground.

A *unifier* of two terms, atoms or literals E_1 and E_2 is a substitution θ such that $E_1\theta = E_2\theta$. It is known that if two expressions have a unifier, then they have a so-called most general unifier.

We assume a standard notion of a (first-order, many-sorted) *interpretation* $\mathcal{I}$, which assigns a non-empty domain $\mathcal{I}_s$ to every sort $s \in \Xi$, and maps every function symbol f to a function $\mathcal{I}_f$ and every predicate symbol p to a relation $\mathcal{I}_p$ on these domains so that the mapping respects sorts. We will call $\mathcal{I}_f$ the *interpretation of* f *in* $\mathcal{I}$, and similarly for $\mathcal{I}_p$ and $\mathcal{I}_s$. Interpretations are also sometimes called *first-order structures*. A *sentence* is a closed formulas, that is, a formula with no free variables. We will use the standard notions of validity and satisfiability of sentences in such interpretations. An interpretation is a *model* for a set of clauses if (the universal closure of) each of these clauses is true in the interpretation.

A *theory* $\mathcal{T}$ is identified by a class of interpretations. A sentence is *satisfiable* in $\mathcal{T}$ if it true in at least one of these interpretations and *valid* if it is true in all of these interpretations. A function (or predicate) symbol f is called *uninterpreted* in $\mathcal{T}$, if for every interpretation $\mathcal{I}$ of $\mathcal{T}$ and every interpretation $\mathcal{I}'$ which agrees with $\mathcal{I}$ on all symbols apart from f, $\mathcal{I}'$ is also an interpretation of $\mathcal{T}$. A theory is called *complete* if, for every sentence F of this theory, either F or $\neg F$ is valid in this theory. Evidently, every theory of a single interpretation is complete. We can define satisfiability and validity of arbitrary formulas in an interpretation in a standard way by treating free variables as new uninterpreted constants.

For example, the theory of integer arithmetic fixes the interpretation of a distinguished sort $s_{int} \in \Xi_{IA}$ to the set of mathematical integers $\mathbb{Z}$ and analogously assigns the usual meanings to $\{+, -, <, >$

$, *\} \in \Omega_{IA}$.

We will mostly deal with theories in which their restriction to interpreted symbols is a complete theory, for example, integer or real linear arithmetic.

In the sequel we assume that $\mathcal{T}$ is an arbitrary but fixed theory and give definitions relatively to this theory.

Abstracted Clauses. Here we discuss how a clause can be separated into a theory and non-theory part. To this end we need to divide all symbols into theory and non-theory symbols. Generally, when we deal with a theory or a combination of theories, we consider as *theory symbols* those symbols interpreted in at least one of the theories and all other symbols as *non-theory symbols*. That is, non-theory symbols are uninterpreted in all theories.

A non-equality literal is a *theory literal* if its predicate symbol is a theory symbol. An equality literal $t_1 \simeq_s t_2$ is a theory literal, if the sort s is a theory sort. A *non-theory literal* is any literal that is not a theory literal. A literal is *pure* if it contains only theory symbols or only non-theory symbols. A clause is *fully abstracted*, or simply *abstracted*, if it only contains pure literals. A clause is *partially abstracted* if non-theory symbols do not appear in theory literals. Note that in partially abstracted clauses theory symbols are allowed to appear in non-theory literals.

A non-variable term t is called a *theory term* (respectively *non-theory term*) if its main function symbol is a theory (respectively non-theory) symbol. When we say that a term is a theory or a non-theory term, we assume that this term is not a variable.

Given a non-abstracted clause $L[t] \vee C$ where L is a theory literal and t a non-theory term (or the other way around), we can construct the equivalent clause $L[x] \vee C \vee x \not\simeq t$ for a fresh variable x. Repeated application of this process will lead to an abstracted clause, and doing this only for theory literals will result in a partially abstracted clause. In both cases, the results are unique (up to variable renaming). We write $\mathcal{A}(C)$ and $\mathcal{A}_P(C)$ for the abstraction and partial abstraction of C, respectively.

The above abstraction process will take $a + a \simeq 1$, where a is a non-theory symbol, and produce $x + y \simeq 1 \vee x \not\simeq a \vee y \not\simeq a$. There is a simpler equivalent fully abstracted clause $x + x \simeq 1 \vee x \not\simeq a$, and we would like to avoid unnecessarily long clauses. For this reason, we will assume that abstraction will abstract syntactically equal subterms using the same fresh variable, as in the above example. If we abstract larger terms first, the result of abstractions will be unique up to variable renaming.

Superposition Calculus. Later we will show how our underlying calculus, the superposition and resolution calculus, can be updated to use an updated notion of unification. For space reasons we do not replicate this calculus here (but it is given in our previous work [15]). We do, however, draw attention to the following *Equality Resolution* rule

$$\frac{\underline{s \not\simeq t} \vee C}{C\theta} \quad \theta \text{ is a most general unifier of } s \text{ and } t$$

as, without modification, this rule will directly undo any abstractions. This rule will be used in Section 3 to justify ignoring certain literals when performing instantiation.

Saturation-Based Proof Search (and Theory Reasoning). We introduce our new approach within the context of saturation-based proof search. The general idea in saturation is to maintain two sets of *Active* and *Passive* clauses. A saturation-loop then selects a clause C from *Passive*, places C in *Active*, applies *generating inferences* between C and clauses in *Active*, and finally places the newly derived clauses in *Passive* after applying some retention tests. The retention tests involve checking whether the new clause is itself redundant (i.e. a tautology) or redundant with respect to existing clauses.

To perform theory reasoning within this context it is common to do two things. Firstly, to *evaluate* new clauses to put them in a common form (e.g. rewrite all inequalities in terms of $<$) and evaluate ground theory terms and literals (e.g. $1+2$ becomes 3 and $1 < 2$ becomes false). Secondly, as previously mentioned, relevant theory axioms can be added to the initial search space. For example, if the input clauses use the $+$ symbol one can add the axioms $x + y \simeq y + x$ and $x + 0 \simeq x$, among others.

3 Generating Simpler Instances

In the introduction, we showed how useful instances can be generated by finding substitutions that make theory literals false. We begin by providing further motivation for the need for instances and then describe a new inference rule capturing this approach and how we implement it within VAMPIRE.

There are some very simple problems that are difficult to solve by the addition of theory axioms. Consider, for example, the following conjecture valid in the theory integer arithmetic:

$$(\exists x)(x + x \simeq 2),$$

which yields the following unit clause after being negated for refutation

$$x + x \not\simeq 2.$$

It takes VAMPIRE almost 15 seconds to refute this clause using theory axioms (and non-trivial search parameters) and involves the derivation of intermediate theory consequences such as $x + 1 \simeq y + 1 \vee y + 1 \leq x \vee x + 1 \leq y$. In contrast, applying the substitution $\{x \mapsto 1\}$ immediately leads to a refutation via evaluation.

A similar case is `ARI120=1` from the TPTP library which consists of the conjecture

$$p(2) \rightarrow (\exists x \exists y)(p(x) \wedge x \not\simeq y \wedge y * y \simeq 4)$$

which leads to the following two clauses

$$x * x \not\simeq 4 \vee x \simeq y \vee \neg p(y) \qquad p(2)$$

that immediately resolve to give

$$x * x \not\simeq 4 \vee 2 \simeq x.$$

which is not solved using theory axiom reasoning alone within VAMPIRE (given reasonable time limits and proof search options). At the same time, application of the substitution $\{x \mapsto -2\}$ and a simple evaluation derives the empty clause.

The generation of instances in this way is not only useful where theory axiom reasoning explodes, it can also significantly shorten proofs where theory axiom reasoning succeeds. For example, there is a proof of the problem `DAT101=1` from the TPTP library using theory axioms that involves generating just over 230k clauses. Whereas, instantiating an intermediate clause

$$\mathsf{inRange}(x, \mathsf{cons}(1, \mathsf{cons}(5, \mathsf{cons}(2, \mathsf{nil})))) \vee x < 4 \tag{1}$$

with $\{x \mapsto 4\}$ solves the problem after generating just 171 clauses.

Theory Instantiation. From the above discussion it is clear that generating instances of theory literals may drastically improve performance of saturation-based theorem provers. The problem is that the set of all such instances can be infinite, so we should try to generate only those instances that are likely not to degrade the performance.

There is a special case of instantiation that allows us to derive from a clause C a clause with fewer literals than C. Consider an inference

$$\frac{P \vee D}{D\theta} \tag{2}$$

such that

1. P contains only pure theory literals;
2. $\neg P\theta$ is valid in $\mathcal{T}$ (equivalently, $P\theta$ is unsatisfiable in $\mathcal{T}$).

Let us now discuss potential problems related to the use of instantiation and abstraction.

We do not want to make a theory literal valid in the theory (a theory tautology) after instantiation. For example, if we had instantiated clause (1) with $\{x \mapsto 3\}$ then the clause would have been evaluated to $true$ (because of the literal $3 < 4$) and thrown away as a theory tautology. However, one must be careful that this process does not simply undo abstraction. For example, consider the unit clause $p(1, 5)$ which will be abstracted as

$$x \not\simeq 1 \vee y \not\simeq 5 \vee p(x, y). \tag{3}$$

The substitution $\theta = \{x \mapsto 1, y \mapsto 5\}$ makes the formula $x \simeq 1 \wedge y \simeq 5$ valid. Its application, followed by evaluation produces $p(x, y)\theta = p(1, 5)$ i.e. the original clause.

More generally, a clause does not need to be abstracted to contain such literals. For example, the clause

$$x \not\simeq 1 + y \vee p(x, y)$$

might produce, after applying (2) and evaluation, the instance $p(1, 0)$ but it can also be used to produce the more general clause $p(y + 1, y)$ using equality resolution.

Based on the above discussion we define literals that we do not want to use for applying (2) since we can use a sequence of equality resolution steps to solve them.

Let C be a clause. The set of *trivial literals in* C is defined recursively as follows. A literal L is trivial in C if

1. L is of the form $x \not\simeq t$ such that x does not occur in t;
2. L is a pure theory literal;
3. every occurrence of x in C apart from its occurrence in $x \not\simeq t$ is either in a literal that is not a pure theory literal, or in a literal trivial in C.

We call such literals trivial as they can be removed by a sequence of equality resolution steps. For example, in clause (3) both $x \not\simeq 1$ and $y \not\simeq 5$ are trivial.

Consider another example: the clause

$$x \not\simeq y + 1 \vee y \not\simeq z \cdot z \vee p(x, y, z).$$

The literal $x \not\simeq y + 1$ is trivial because, apart from this literal x occurs only in the non-theory literal $p(x, y, z)$. The literal $y \not\simeq z \cdot z$ is also trivial because y occurs only in non-theory literal $p(x, y, z)$ and in a trivial literal $x \not\simeq y + 1$.

It is not hard to argue that all pure theory literals introduced by abstraction are trivial.

We can now define *theory instantiation* as the inference rule

$$\frac{P \vee D}{D\theta}\ (\mathit{TheoryInst})$$

such that

1. P contains only pure theory literals;
2. P contains no literals trivial in $P \vee D$;
3. $\neg P\theta$ is valid in $\mathcal{T}$ (equivalently, $P\theta$ is unsatisfiable in $\mathcal{T}$).

Implementation. To use $\mathit{TheoryInst}$, we apply the following steps to each given clause C:

1. abstract relevant literals;
2. collect (all) non-trivial pure theory literals $L_1, \ldots, L_n$;
3. run an SMT solver on $T = \neg L_1 \wedge \ldots \wedge \neg L_n$;
4. if the SMT solver returns a model, turn it into a substitution θ such that $T\theta$ is valid in $\mathcal{T}$;
5. if the SMT solver returns *unsatisfiable* then C is a theory tautology and can be removed.

Note that the abstraction step is not necessary for using $\mathit{TheoryInst}$, since it will only introduce trivial literals. However, for each introduced theory literal $x \not\simeq t$ the variable x occurs in a non-theory literal and inferences applied to this non-theory literal may instantiate x to a term s such that $s \not\simeq t$ is non-trivial.

The next step is to understand when and how we can turn the model returned by the SMT solver into a substitution making T valid. Note that T can contain

1. interpreted symbols that have a fixed interpretation in $\mathcal{T}$, such as 0 or $+$ over integer arithmetic;
2. other interpreted symbols;
3. variables of T.

In general, there are no standards on how SMT solvers return models or solutions. We assume that the model returned by the underlying SMT solver can be turned into a conjunction S of literals such that

1. S is satisfiable in $\mathcal{T}$;
2. $S \rightarrow T$ is valid in $\mathcal{T}$.

Note that, checking that T is satisfiable and returning T as a model satisfies both these conditions but does not give us a substitution that can be used to apply the theory instantiation rule.

To apply this rule, we need models of a special form defined below. A conjunction S of literals is said to be in *triangle form* if S has the form

$$x_1 \simeq t_1 \wedge \ldots \wedge x_n \simeq t_n \qquad (4)$$

such that for all $i = 1, \ldots, n$ the variable x_i does not occur in $t_i, \ldots, t_n$. Any model S in a triangle form can be converted into a substitution θ such that $x_i\theta = t_i\theta$ for all $i = 1, \ldots, n$. Note that $S\theta$ is then valid, hence (by validity of $S \rightarrow T$), $T\theta$ is valid too, so we can use θ to apply the theory instantiation rule.

We will consider later what to do when S is not (or even cannot be) in triangle form.

Let us now discuss the implementation of each step in further detail.

Performing Abstraction. We do not fully abstract clauses because we perform abstraction via unification (see the next section).

Selecting Pure Theory Literals. Note that in the definition of $\mathit{TheoryInst}$ we did not specify that P contains *all* pure theory literals in the premise. The reason is that some pure theory literals may be unhelpful. For example, consider

$$x \simeq 0 \vee p(x).$$

Here the SMT solver could select any value for x, apart from 0. In general, positive equalities are less helpful than negative equalities or interpreted predicates as they restrict the instances less. We introduce three options to control this selection:

- `strong`: Only select *strong* literals where a literal is strong if it is a negative equality or an interpreted literal
- `overlap`: Select all strong literals and additionally those theory literals whose variables overlap with a strong literal
- `all`: Select all non-trivial pure theory literals

At this point there may not be any pure theory literals to select, in which case the inference will not be applied.

Interacting with the SMT solver. In this step we replace variables in selected pure theory literals by new constants and negate the literals. Once this has been done, the translation of literals to the format understood by the SMT solver is straightforward (and outlined in [21]). We use Z3 [11] in this work.

Additional care needs to be taken when translating partial functions, such as division. In SMT solving they are treated as total underspecified functions. For example, when $\mathcal{T}$ is integer arithmetic with division, interpretations for $\mathcal{T}$ are defined in such a way that for all integers a, b and interpretation $\mathcal{I}$, the theory also has the interpretation defined exactly as $\mathcal{I}$ apart from having $a/0 = b$. In a way, division by 0 behaves as an uninterpreted function.

Thanks to this convention, Z3 is allowed to generate an arbitrary value for the result in order to satisfy a given query. As a result, Z3 can produce a model that is output as an ordinary solution except for the assumptions about division by 0. For example solving $2/x = 1$ can return $x = 0$. If we accept that $x \simeq 0$ is a solution in triangle form, the theorem prover may become unsound.

As an example, consider a problem consisting of the following two clauses

$$1/x \not\simeq 0 \vee p(x) \qquad 1/x \simeq 0 \vee \neg p(x).$$

The example is satisfiable as witnessed by an interpretation that assigns false to $p(z)$ for every real number z and interprets $1/0$ as a non-zero real, e.g. 1. However, the $\mathit{TheoryInst}$ rule as presented so far could produce conflicting instances $p(0)$ and $\neg p(0)$ of the two clauses, internally assuming $1/0 = 0$ for the first instances and $1/0 \neq 0$ for the second.

To deal with this issue, we additionally assert that $s \not\simeq 0$ whenever we translate a term of the form t/s. This implies that we do not pass to the SMT solver terms of the form $t/0$ at all.

Instance Generation. To produce the necessary substitution we must evaluate the introduced constants (i.e. those introduced for each of the variables in the above step) in the given model. In some cases, this evaluation fails to give a numeric value. For example, if the result falls out of the range of values internally representable by VAMPIRE or when the value is a proper algebraic number, which currently also cannot be represented internally by our prover. In this case, we cannot produce a substitution and the inference fails.

Theory Tautology Deletion. As we pointed out above, if the SMT solver returns *unsatisfiable* then C is a theory tautology and can be removed. We only do it when we do not pass to the solver additional assumptions related to division by 0.

4 Abstraction Through Unification

As shown in the introduction, there are cases where we cannot perform a necessary inference step, because we are using a *syntactic* notion of equality rather than a *semantic* one. The previous section introduced an inference rule ($\mathit{TheoryInst}$) able to derive $p(7)$ from the clause

$$14x \not\simeq x^2 + 49 \lor p(x)$$

but unable to deal with a pair of clauses such as

$$r(14y) \qquad \neg r(x^2 + 49) \lor p(x),$$

as it only performs theory reasoning *inside* a clause whereas this requires us to reason *between* clauses. Semantically, the terms $14y$ and $x^2 + 49$ can be made equal when $y = x = 7$ so we would like to get the result $p(7)$ here also.

Notice that if the clauses had been abstracted as follows:

$$r(u) \lor u \not\simeq 14y \qquad \neg r(v) \lor v \not\simeq x^2 + 49 \lor p(x),$$

then the resolution step would have been successful, producing

$$u \not\simeq 14y \lor u \not\simeq x^2 + 49 \lor p(x)$$

which could be given to $\mathit{TheoryInst}$ to produce $p(7)$. One solution would be to store clauses in abstracted form, but we argued in the introduction why this solution is not suitable and will later confirm this by experimental results. Instead of abstracting fully we incorporate the abstraction process into unification so that only abstractions necessary for a particular inference are performed. Performing abstraction in this way is *lazy*, i.e., it delays abstraction until it is needed.

Unification with Abstraction. Here we define a partial function $\mathsf{mgu}_{\mathsf{Abs}}$ on pairs of terms and pairs of atoms such that $\mathsf{mgu}_{\mathsf{Abs}}(t, s)$ is either undefined, in which case we say that it *fails* on (s, t), or $\mathsf{mgu}_{\mathsf{Abs}}(t, s) = (\theta, D)$ such that

1. θ is a substitution and D is a (possibly empty) disjunction of disequalities;
2. $(D \lor t \simeq s)\theta$ is valid in the underlying theory (and even valid in predicate logic).

To define the algorithm

Algorithm 1 gives a unification algorithm extended so that it implements $\mathsf{mgu}_{\mathsf{Abs}}$. The algorithm is parameterised by a `canAbstract` predicate. The idea here is that some abstractions are not useful. For example, consider the two clauses

$$p(1) \qquad \neg p(2).$$

Algorithm 1 Unification algorithm with constraints

function $\mathsf{mgu}_{\mathsf{Abs}}(l, r)$
 let $\mathcal{E}$ be a set of equations; $\mathcal{E} := \{l = r\}$
 let $\mathcal{D}$ be a set of disequalities; $\mathcal{D} := \emptyset$
 let θ be a substitution; $\theta := \{\}$
 loop
  if $\mathcal{E}$ is empty **then**
   return (θ, D), where D is the disjunction of literals in $\mathcal{D}$
  Select an equation $s = t$ in $\mathcal{E}$ and remove it from $\mathcal{E}$
  if s coincides with t **then**
   do nothing
  else if s is a variable and s does not occur in t **then**
   $\theta := \theta \circ \{s \mapsto t\}$; $\mathcal{E} := \mathcal{E}\{s \mapsto t\}$
  else if s is a variable and s occurs in t **then**
   fail
  else if t is a variable **then**
   $\mathcal{E} := \mathcal{E} \cup \{t = s\}$
  else if s and t have different top-level symbols **then**
   if `canAbstract`(s, t) **then**
    $\mathcal{D} := \mathcal{D} \cup \{s \not\simeq t\}$
   else fail
  else if $s = f(s_1, \ldots, s_n)$ and $t = f(t_1, \ldots, t_n)$ for some f **then**
   $\mathcal{E} := \mathcal{E} \cup \{s_1 = t_1, \ldots, s_n = t_n\}$

Allowing 1 and 2 to unify under the constraint that $1 \simeq 2$ is not useful in any context. Therefore, `canAbstract` will always be false if the two terms are always non-equal in the underlying theory, e.g. if they are distinct numbers in the theory of arithmetic. Beyond this obvious requirement we also want to control how prolific such unifications can be. Therefore, we include the following options here:

- `interpreted_only`: only produce a constraint if the top-level symbol of both terms is a theory symbol,
- `one_side_interpreted`: only produce a constraint if the top-level symbol of at least one term is a theory symbol,
- `one_side_constant`: only produce a constraint if the top-level symbol of at least one term is a theory symbol and the other is an uninterpreted constant
- `all`: allow all terms of theory sort to unify and produce constraints.

Updated Calculus. So far we have only considered resolution as a rule that could be updated with this new form of unification, but in principle it can be used wherever we use unification. Figure 1 gives the superposition and resolution calculus updated to make use of unification with abstraction. Inference rules are renamed to indicate that they are performed with Abstraction. Now given the problem from the introduction involving $p(2x)$ and $\neg p(10)$ we can apply Resolution-wA to produce $2x \not\simeq 10$ which can be resolved using evaluation and equality resolution as before.

Notice that the Superposition-wA rule has an additional constraint that if l is not a variable, then l

Resolution-wA

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

Factoring-wA

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

where, for both inferences, $(\theta, D) = \mathsf{mgu}_{\mathsf{Abs}}(A, A')$ and A is not an equality literal.

Superposition-wA

$$\frac{\underline{l \simeq r} \vee C_1 \quad \underline{L[s]_p} \vee C_2}{(D \vee L[r]_p \vee C_1 \vee C_2)\theta} \;; \qquad \frac{\underline{l \simeq r} \vee C_1 \quad \underline{t[s]_p \simeq t'} \vee C_2}{(D \vee t[r]_p \simeq t' \vee C_1 \vee C_2)\theta} \;; \qquad \frac{\underline{l \simeq r} \vee C_1 \quad \underline{t[s]_p \not\simeq t'} \vee C_2}{(D \vee t[r]_p \not\simeq t' \vee C_1 \vee C_2)\theta} \;,$$

where (i) $(\theta, D) = \mathsf{mgu}_{\mathsf{Abs}}(l, s)$; (ii) s is not a variable; (iii) if l is not a variable, then l and s have the same top-level symbols; (iv) $r\theta \not\succeq l\theta$; (v) $t'\theta \not\succeq t[s]\theta$, (vi) in the first rule $L[s]$ is not an equality literal.

Equality Resolution-wA

$$\frac{\underline{s \not\simeq t} \vee C}{(D \vee C)\theta} \;,$$

where $(\theta, D) = \mathsf{mgu}_{\mathsf{Abs}}(s, t)$

Equality Factoring-wA

$$\frac{\underline{s \simeq t} \vee s' \simeq t' \vee C}{(D \vee t \not\simeq t' \vee s' \simeq t' \vee C)\theta} \;,.$$

where $(\theta, D) = \mathsf{mgu}_{\mathsf{Abs}}(s, s')$, $t\theta \not\succeq s\theta$, and $t'\theta \not\succeq s'\theta$.

Figure 1: The updated rules of the superposition and resolution calculus.

and s have the same top-level symbols. This is to prevent the situation where the unit clauses

$$f(a) \simeq b \qquad p(1)$$

are used to produce $f(a) \not\simeq 1 \vee p(b)$. Notice that if l and s have different top-level symbols then this is the same as abstracting to get

$$f(a) \simeq b \qquad p(x) \vee x \not\simeq 1$$

and then performing superposition into variables, which we do not do in general. We note at this point that a further advantage of this updated calculus is that it directly resolves the issue of losing proofs via eager evaluation, i.e. where $p(1+3)$ is eagerly evaluated to $p(4)$, missing the chance to resolve with $\neg p(x+3)$.

Implementation. In VAMPIRE, as in most modern theorem provers, inferences involving unification are implemented via *term indexing* [29]. Therefore, to update how unification is applied we need to update our implementation of term indexing. As the field of term indexing is highly complex we only give a sketch of the update here.

Term indices provide the ability to use a *query term* t to extract terms that unify (or match, or generalise) with t along with the relevant substitutions. Like many theorem provers, VAMPIRE uses substitution trees [14] to index terms. The idea behind substitution trees is to abstract a term into a series of substitutions required to generate that term and store these substitutions in the nodes of the tree. To search for unifying terms we perform a backtracking search over the tree, composing substitutions from the nodes when descending down edges and checking at each node whether the query term is consistent with the current substitution. This involves unifying subterms of the query term against terms at nodes and a backtrackable result substitution must be maintained to store the results of these unifications. The result substitution must be backtracked as appropriate i.e. when backtracking past the point of unification.

To update this process we do two things. Firstly, wherever previously a unification failed we will produce a set of constraints using Algorithm 1. Secondly, alongside the backtrackable result substitution we maintain a backtrackable stack of constraints so that whenever we backtrack past a point where we made a unification that produced some constraints we remove those constraints from the stack.

5 Experimental Results

We present experimental results evaluating the effectiveness of the new techniques.

Experimental Setup. Our experiments were carried out on a cluster on which each node is equipped with two quad core Intel processors running at 2.4 GHz and 24GiB of memory. We used all the problems with quantifiers and theories from the SMTLIB library [5] (version 2016-05-23) apart from problems using the theory of bit vectors.

Comparing New Options. We were interested in comparing how various proof option values affect the performance of a theorem prover. For the purpose of this research, we consider the two new options referred to here by their short names: uwa (unification with abstraction) and thi (theory instantiation). In addition, we consider the boolean option fta (full theory abstraction), applying full abstract to input clauses. This option was implemented in previous versions of VAMPIRE.

Making such a comparison is hard, since there is no obvious methodology for doing so, especially considering that VAMPIRE has over 60 options commonly used in experiments (see [24]). The majority of these options are Boolean, some are finitely-valued, some integer-valued and some range over other infinite domains. The method we used here was based on the following ideas, already described in [17].

1. We use a set of problems obtained by discarding problems that are too easy or currently unsolvable, and
2. we repeatedly select a random problem P in this set, a random strategy S and run P on variants of S obtained by choosing possible values for the three options using the same time limit.

We considered combinations of option values satisfying the following natural conditions: either fta or uwa must be off, since it does not make sense to use unification with abstraction when full abstraction is performed. This resulted in 24 possible combinations of values. We ran about $100\,000$ tests with the time limit of 30 seconds, which is about 4000 per a combination of options. The results are shown in Table 1.

It may seem surprising that the overall best strategy has all the three options turned off. In fact, it is due to what we have already observed before: many SMTLIB problems with quantifiers and theories require very little theory reasoning. Indeed, VAMPIRE solves a large number of problems (including problems unsolvable by existing SMT solvers) just by adding theory axioms and then running superposition with no theory-related rules. Such problems do not gain from the new options, because new inference rules result only in more generated clauses.

Let us summarise the behaviour of three options, obtained by a more detailed analysis of our experimental results.

Table 1: Comparison of Three Options

fta	uwa	thi	solutions	fta	uwa	thi	solutions
on	off	all	252	off	one_side_interpreted	strong	387
on	off	overlap	265	off	off	all	392
on	off	strong	266	off	one_side_constant	strong	397
on	off	off	276	off	one_side_constant	overlap	401
off	all	all	333	off	interpreted_only	overlap	407
off	all	overlap	351	off	one_side_interpreted	off	407
off	all	strong	354	off	interpreted_only	strong	409
off	one_side_interpreted	all	364	off	one_side_constant	off	417
off	all	off	364	off	off	overlap	428
off	one_side_constant	all	374	off	interpreted_only	off	430
off	interpreted_only	all	379	off	off	strong	431
off	one_side_interpreted	overlap	385	off	off	off	450

Full theory abstraction. Probably the most interesting observation from these results is that the use of full abstraction (fta) results in an observable degradation of performance. This confirms our intuition that unification with abstraction is a good replacement for abstraction. As a result, we will remove the fta option from VAMPIRE.

Unification with abstraction. This option turned out to be very useful. Many problems had immediate solutions with uwa turned on and no solutions when it was turned off. Further, the value all resulted in 12 unique solutions. We have decided to keep the values all, interpreted_only and off.

Theory instantiation. This option turned out to be very useful too. Many problems had immediate solutions with this turned on and no solutions when it was turned off. We have decided to keep the values all, strong and off.

Contribution of New Options to Strategy Building. Since modern provers normally run a portfolio of strategies to solve a problem (strategy scheduling), there are two ways new strategies can be useful in such a portfolio:

1. by reducing the overall schedule time because problems are solved faster or because a single strategy replaces one or more old strategies;
2. by solving previously unsolved problems.

While for decidable classes, such as propositional logic, the first way can be more important, in first-order logic it is usually the second way that matters. The reason is that, if a problem is solvable by a prover, it is usually solvable with a short running time.

We ran VAMPIRE trying to solve, using the new options, problems previously unsolved by VAMPIRE. We took all such problems from the TPTP library [31] and SMT-LIB [5] and Table 2 shows the results. In the table, new solutions are meant with respect to what VAMPIRE could previously solve and uniquely solved stands for the number of new problems with respect to what can be solved by other entrants into SMT-COMP[1] and CASC[2] where the main competitors are SMT solvers such as Z3 [11] and CVC4 [4] and ATPs such as Beagle [6] and Princess [27, 28]. Figure 2 lists the problems uniquely solved by VAMPIRE (the last two are from TPTP, the rest from SMT-LIB).

[1] http://smtcomp.sourceforge.net/
[2] http://www.cs.miami.edu/~tptp/CASC/

SMT-LIB

Logic	New solutions	Uniquely solved
ALIA	1	0
LIA	14	0
LRA	4	0
UFDTLIA	5	0
UFLIA	28	14
UFNIA	13	4

TPTP

Category	New solutions	Uniquely solved
ARI	13	0
SWW	3	1

Table 2: Results from finding solutions to previously unsolved problems.

```
UFLIA/sledgehammer/FFT/smtlib.724613
UFLIA/sledgehammer/QEpres/smtlib.1037205
UFLIA/sledgehammer/QEpres/smtlib.1043920
UFLIA/sledgehammer/QEpres/smtlib.1124616
UFLIA/sledgehammer/QEpres/smtlib.1133209
UFLIA/sledgehammer/QEpres/smtlib.670238
UFLIA/sledgehammer/QEpres/z3.1116453
UFLIA/sledgehammer/QEpres/z3.1125133
UFLIA/sledgehammer/QEpres/z3.667771
UFLIA/sledgehammer/TwoSquares/smtlib.718092
UFLIA/sledgehammer/TwoSquares/smtlib.721402
UFLIA/sledgehammer/TwoSquares/smtlib.734258
UFLIA/sledgehammer/TwoSquares/smtlib.737811
UFLIA/sledgehammer/TwoSquares/smtlib.779788
UFNIA/sledgehammer/FFT/z3.723395
UFNIA/sledgehammer/FFT/z3.723638
UFNIA/sledgehammer/QEpres/z3.714625
UFNIA/sledgehammer/TwoSquares/z3.615666
SWW580=2
```

Figure 2: Problems solved uniquely.

We note that the UFDTLIA category is a new category covering *data types* (along with uninterpreted functions and linear integer arithmetic) added as an experimental track to SMT-COMP for the first time this year.

With the help of the new options VAMPIRE solved 19 problems previously unsolved by any other theorem prover or SMT solver.

6 Related Work

We review relevant related work.

SMT Solving. SMT solvers such as Z3 [11] and CVC4 [4] implement E-matching [12, 9], model based quantifier instantiation [12, 9] and conflict instantiation [26] techniques to deal with quantifiers. These take the form of instantiation and are generally heuristic, although can be used to decide some fragments. The work of $DPLL(\Gamma)$ [10] combines a superposition prover with an SMT solver such that the ground literals decided and implied by the SMT solver are used as hypotheses to first-order clauses.

AVATAR Modulo Theories. Our previous work on AVATAR Modulo Theories [21] uses the AVATAR architecture [32, 23] for clause splitting to integrate an SMT solver with a superposition prover. The idea behind AVATAR is to abstract the clause search space as a SAT problem and use a SAT solver to decide on at least one literal per clause to have in the current search space of the superposition prover. In AVATAR modulo theories the SAT solver was replaced by a SMT solver. In this abstraction, non-ground components (sub-clauses sharing variables) are abstracted as propositional symbols whilst ground literals are translated directly. The result is that the superposition prover only deals with a set of clauses that is theory-consistent in its ground part.

Theory Resolution. Stickel's Theory Resolution [30] is a generalisation of the resolution inference rule whose aim is to exclude the often prolific theory axioms from the explicit participation on reasoning about the uninterpreted part of a given problem.

Let us define the theory resolution rule exactly as in [30], where it is defined for the ground case with a note that "lifting to the general case is straightforward". Let $\mathcal{T}$ be a theory. We call an inference $\mathcal{T}$-sound if its the conclusion is implied by its premises in $\mathcal{T}$.

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

$$\frac{K_1 \vee L_1 \quad \cdots \quad K_m \vee L_m}{L_1 \vee \ldots \vee L_m \vee \neg R_1 \vee \ldots \vee \neg R_n} \ ,$$

where $R_1, \ldots, R_n$ are literals and $\{K_1, \ldots, K_m, R_1, \ldots, R_n\}$ are $\mathcal{T}$-unsatisfiable.

It is not hard to argue that every theory resolution inference is $\mathcal{T}$-sound. Let us show that every $\mathcal{T}$-sound inference is also a theory resolution inference. Suppose that $K_1, \ldots, K_m$ are clauses and a clause $S_1 \vee \ldots \vee S_n$ is implied by $K_1, \ldots, K_m$ in T and consider the inference

$$\frac{K_1 \quad \cdots \quad K_m}{S_1 \vee \ldots \vee S_n} \ .$$

Then this inference is a special case of theory resolution when $L_1, \ldots, L_m$ are empty clauses and $R_i = \neg S_i$ for all $i = 1, \ldots, n$.

In view of the fact that theory resolution is a re-definition of $\mathcal{T}$-sound inferences, it does not make sense to compare it with our approach.

Literal Deletion. In their work on reasoning with real-closed-fields, Akbarpour and Paulson [1] introduce a technique called *literal deletion* that is somewhat similar to our $\mathit{TheoryInst}$ rule. The general idea is to take a clause $C \vee L$ and use the inconsistency of $\neg C \wedge L$ to justify the deletion of literal L, for example, $x \leq 3 \vee x = 3$ can be reduce to $x \leq 3$. The similarity lies in the use of local theory reasoning to simplify clauses.

Hierarchic Superposition. Hierarchic Superposition (HS) [3] is a generalisation of the superposition calculus for black-box style theory reasoning. Notably, the approach uses full abstraction to separate the theory and non-theory part of the problem and introduces a conceptual hierarchy between uninterpreted reasoning (with the calculus) and theory reasoning (delegated to a theory solver) by making pure theory terms smaller than everything else. HS guarantees refutational completeness under certain conditions including *sufficient completeness*, which essentially requires that every ground non-theory term of theory sort should be provably equal to a pure theory term. This is rather restrictive, for example, the clauses

$p(x)$ and $\neg p(f(c))$ cannot be resolved if the return sort of function f is a theory sort as the required substitution is not *simple*. The strategy of *weak abstractions* introduced by Baumgartner and Waldmann [7] partially addresses the downsides of the original approach. They introduce a distinction of abstraction versus ordinary variables which (i) allows abstraction to preserve sufficient completeness, and (ii) enables the use of theory axioms (which HS cannot use). By treating the variable x in the above example as ordinary the problem can be refuted. However, they do not address how to make a non-sufficiently complete input sufficiently complete by treating certain variables as ordinary or adding theory axioms. Although this cannot be answered in general for decidability reasons, a practical approach is clearly desired.

Other Theory Instantiation. SPASS+T [20] implements a theory instantiation rule that is analogous to E-matching in the sense that it uses ground theory terms from the search space to perform instantiations as a last resort. This is not related to our approach.

Unification Modulo Theories. There is a large amount of work on unification modulo various theories, such as AC. This work is not related since we are not looking for the set of all or most general solutions to unification. Instead, we postpone finding such solutions by creating constraints, which will at some moment be processed by the SMT solver.

7 Conclusion

We have introduced two new techniques for reasoning with problems containing theories and quantifiers. The first technique allows us to utilise the power of SMT solving to find useful instances of non-ground clauses. The second technique presents a solution to the issue of full abstraction by lazily abstracting clauses to allow them to unify under theory constraints. Our experimental results show that these approaches can solve problems previously unsolvable by VAMPIRE and other solvers.

There are two directions for future research that we believe will further increase the power of this technique. Firstly, whilst this approach complements the AVATAR modulo theories work by producing ground instances that can be used by the SMT solver employed by AVATAR, the relationship between these techniques could be further exploited. Notice that the SMT solver used in AVATAR modulo theories contains information about the possible status of the ground part of the search space, notably the possible values of uninterpreted constants. By including this information in the theory instantiation check it would be possible to also remove literals containing uninterpreted constants. Secondly, the theory instantiation technique is restricted to single concrete models. Extending this technique to deal with general solutions would further increase its power. Both of these approaches will be explored in future work.

References

[1] B. Akbarpour and L. C. Paulson. *Extending a Resolution Prover for Inequalities on Elementary Functions*, pp. 47–61. Springer Berlin Heidelberg, 2007.

[2] E. Althaus, E. Kruglov, and C. Weidenbach. Superposition modulo linear arithmetic SUP(LA). In *Frontiers of Combining Systems, 7th International Symposium, FroCoS 2009, Trento, Italy, September 16-18, 2009. Proceedings*, vol. 5749 of *Lecture Notes in Computer Science*, pp. 84–99. Springer, 2009.

[3] L. Bachmair, H. Ganzinger, and U. Waldmann. Refutational theorem proving for hierarchic first-order theories. *Appl. Algebra Eng. Commun. Comput.*, 5:193–212, 1994.

[4] C. Barrett, C. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli. CVC4. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, number 6806 in Lecture Notes in Computer Science, pp. 171–177. Springer-Verlag, 2011.

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

[6] P. Baumgartner, J. Bax, and U. Waldmann. Beagle - A Hierarchic Superposition Theorem Prover. In *Proceedings of the 25th International Conference on Automated Deduction*, number 9195 in Lecture Notes in Computer Science, pp. 285–294. Springer-Verlag, 2015.

[7] P. Baumgartner and U. Waldmann. Hierarchic Superposition With Weak Abstraction. In *Proceedings of the 24th International Conference on Automated Deduction*, number 7898 in Lecture Notes in Artificial Intelligence, pp. 39–57. Springer-Verlag, 2013.

[8] M. P. Bonacina, C. Lynch, and L. M. de Moura. On deciding satisfiability by theorem proving with speculative inferences. *J. Autom. Reasoning*, 47(2):161–189, 2011.

[9] L. M. de Moura and N. Bjørner. Efficient e-matching for SMT solvers. In *Automated Deduction - CADE-21, 21st International Conference on Automated Deduction, Bremen, Germany, July 17-20, 2007, Proceedings*, pp. 183–198, 2007.

[10] L. M. de Moura and N. Bjørner. Engineering DPLL(T) + saturation. In *Automated Reasoning, 4th International Joint Conference, IJCAR 2008, Sydney, Australia, August 12-15, 2008, Proceedings*, pp. 475–490, 2008.

[11] L. M. de Moura and N. Bjørner. Z3: an efficient SMT solver. In *Proc. of TACAS*, vol. 4963 of *LNCS*, pp. 337–340, 2008.

[12] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.

[13] H. Ganzinger and K. Korovin. Theory instantiation. In *Logic for Programming, Artificial Intelligence, and Reasoning, 13th International Conference, LPAR 2006, Phnom Penh, Cambodia, November 13-17, 2006, Proceedings*, vol. 4246 of *Lecture Notes in Computer Science*, pp. 497–511. Springer, 2006.

[14] P. Graf. *Substitution tree indexing*, pp. 117–131. Springer Berlin Heidelberg, 1995.

[15] K. Hoder, G. Reger, M. Suda, and A. Voronkov. Selecting the selection. In *Automated Reasoning: 8th International Joint Conference, IJCAR 2016, Coimbra, Portugal, June 27 – July 2, 2016, Proceedings*, pp. 313–329. Springer International Publishing, 2016.

[16] K. Korovin and A. Voronkov. Integrating linear arithmetic into superposition calculus. In *Computer Science Logic, 21st International Workshop, CSL 2007, 16th Annual Conference of the EACSL, Lausanne, Switzerland, September 11-15, 2007, Proceedings*, vol. 4646 of *Lecture Notes in Computer Science*, pp. 223–237. Springer, 2007.

[17] L. Kovács, S. Robillard, and A. Voronkov. Coming to terms with quantified reasoning. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pp. 260–270. ACM, 2017.

[18] L. Kovács and A. Voronkov. First-order theorem proving and Vampire. In *CAV 2013*, vol. 8044 of *Lecture Notes in Computer Science*, pp. 1–35, 2013.

[19] A. Nonnengart and C. Weidenbach. Computing small clause normal forms. In *Handbook of Automated Reasoning (in 2 volumes)*, pp. 335–367. Elsevier and MIT Press, 2001.

[20] V. Prevosto and U. Waldmann. SPASS+T. In *Proceedings of the FLoC'06 Workshop on Empirically Successful Computerized Reasoning, 3rd International Joint Conference on Automated Reasoning*, number 192 in CEUR Workshop Proceedings, pp. 19–33, 2006.

[21] G. Reger, N. Bjørner, M. Suda, and A. Voronkov. AVATAR modulo theories. In *GCAI 2016. 2nd Global Conference on Artificial Intelligence*, vol. 41 of *EPiC Series in Computing*, pp. 39–52. EasyChair, 2016.

[22] G. Reger and M. Suda. Set of support for theory reasoning. In *IWIL Workshop and LPAR Short Presentations*, vol. 1 of *Kalpa Publications in Computing*, pp. 124–134. EasyChair, 2017.

[23] G. Reger, M. Suda, and A. Voronkov. Playing with AVATAR. In *Automated Deduction - CADE-25: 25th*

International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings, pp. 399–415. Springer International Publishing, 2015.

[24] G. Reger, M. Suda, and A. Voronkov. The challenges of evaluating a new feature in vampire. In *Proceedings of the 1st and 2nd Vampire Workshops*, vol. 38 of *EPiC Series in Computing*, pp. 70–74. EasyChair, 2016.

[25] G. Reger, M. Suda, and A. Voronkov. New techniques in clausal form generation. In *GCAI 2016. 2nd Global Conference on Artificial Intelligence*, vol. 41 of *EPiC Series in Computing*, pp. 11–23. EasyChair, 2016.

[26] A. Reynolds, C. Tinelli, and L. M. de Moura. Finding conflicting instances of quantified formulas in SMT. In *Formal Methods in Computer-Aided Design, FMCAD 2014, Lausanne, Switzerland, October 21-24, 2014*, pp. 195–202, 2014.

[27] P. Rümmer. A Constraint Sequent Calculus for First-Order Logic with Linear Integer Arithmetic. In *Proceedings of the 15th International Conference on Logic for Programming Artificial Intelligence and Reasoning*, number 5330 in Lecture Notes in Artificial Intelligence, pp. 274–289. Springer-Verlag, 2008.

[28] P. Rümmer. E-Matching with Free Variables. In *Proceedings of the 18th International Conference on Logic for Programming Artificial Intelligence and Reasoning*, number 7180 in Lecture Notes in Artificial Intelligence, pp. 359–374. Springer-Verlag, 2012.

[29] R. Sekar, I. Ramakrishnan, and A. Voronkov. Term indexing. In *Handbook of Automated Reasoning*, vol. II, chapter 26, pp. 1853–1964. Elsevier Science, 2001.

[30] M. E. Stickel. Automated deduction by theory resolution. *J. Autom. Reasoning*, 1(4):333–355, 1985.

[31] G. Sutcliffe. The TPTP problem library and associated infrastructure. *J. Autom. Reasoning*, 43(4):337–362, 2009.

[32] A. Voronkov. AVATAR: The architecture for first-order theorem provers. In *Computer Aided Verification*, vol. 8559 of *Lecture Notes in Computer Science*, pp. 696–710. Springer International Publishing, 2014.