Instantiation for Theory Reasoning in Vampire

Giles Reger Martin Riener

University of Manchester, Manchester, UK

Abstract: Reasoning with theories and quantifiers in first-order logic is very hard. Over the past 3 years we have extended the Vampire theorem prover with various techniques for reasoning with problems mixing arithmetic, quantifiers, and uninterpreted functions. In this most recent work we introduce a new method for instantiation that makes use of SMT solvers to find simplifying instances of clauses and a new approach to unification that enables the application of this rule.

1 Introduction

We are interested in extending automated theorem provers for first-order logic to reason effectively with problems containing non-trivial quantification and theories such as arithmetic or datatypes. Such problems arise naturally in, for example, program analysis where quantifiers are required to axiomatise features such as dynamic memory, and arithmetic is central to most real-world programs.

Our work is in the context of the Vampire theorem prover [1]. This is an automated theorem prover (ATP) that is saturation-based and implements the superposition and resolution calculus. In saturation-based theorem provers the approach is to first transform the input problem into clausal form and then saturate the set of clauses with respect to an inference system. Vampire is also a refutational prover; its first step is always to negate the goal, which means that it aims to derive a contradiction. In pure first-order logic this approach can be refutationally complete. This breaks down in the presence of theories such as arithmetic.

Over the past 3 years we have been exploring different approaches for theory reasoning within Vampire. This has included using an SMT solver to guide proof search [2] and heuristics to control the use of theory axioms such as $x + y = y + x$ [3]. This work considers the problem of *instantiation* (for theories) in this context.

2 Background

We consider a many-sorted first-order logic over the signature $\Sigma = (\Xi, \Omega)$. The set Ω contains predicate and function symbols with argument and return values in the set of sorts Ξ (which contains the sort $\mathbb{B}$ of truth values). A *term* is a constant c, a variable x or an application $f(t_1, \ldots, t_n)$ of the n-ary function symbol f to the terms t_1 to t_n. We assume terms are well-sorted. A function symbol p with return sort $\mathbb{B}$ is called a predicate symbol. Its application $p(t_1, \ldots, t_n)$ is called an *atom*. We assume the presence of an equality predicate for each sort. A *literal* is either an atom A or a negated atom $\neg A$. We abbreviate $\neg(c \simeq_s d)$ as $c \not\simeq_s d$. A subterm s of t at position p is written as $t[s]_p$.

A clause is a multiset of literals which is interpreted as a disjunction $L_1 \vee \ldots \vee L_n$. A substitution $\theta = \{x_1 \mapsto t_1, \ldots x_n \mapsto t_n\}$ maps variables to terms; applying θ to a term simultaneously replaces the variables by the corresponding terms. A *unifier* θ of two terms s and t is a substitution such that $(s \simeq t)\theta$ is valid; a *most general unifier* of s and t is a unifier that is not an instance of any other unifier of those terms up to renaming of variables.

A theory defines a class of interpretations. All interpretations in a theory $\mathcal{T}$ agree on the assignment for a set of *theory symbols*. A symbol that does not have a fixed interpretation is called a *non-theory symbol*.

A literal is a *theory literal* if its predicate symbol is a theory symbol. The equality $\simeq_s$ predicate of a sort s is a theory symbol if the sort s is interpreted by a theory. A *pure* literal contains only theory symbols or only non-theory symbols. A clause is *fully abstracted* if it only contains pure literals and *partially abstracted* if non-theory symbols no not appear inside applications of theory symbols. A non-variable term t is a *theory term* (*non-theory term*) if its top function symbol is a theory symbol (non-theory symbol).

Given a clause $L[t] \vee C$, where L is a theory literal and t is a non-theory literal or vice versa, we can separate them by introducing a fresh variable x for t to obtain $L[x] \vee C \vee x \not\simeq t$. Repeating this process leads to an abstracted clause.

2.1 Does Vampire Need Instantiation?

To see why Vampire can benefit from instantiation, consider the first-order clause

$$14x \not\simeq x^2 + 49 \vee p(x) \tag{1}$$

for which there is a single integer value for x that makes the first literal false with respect to the underlying theory of arithmetic, namely $x = 7$. However, 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. Indeed, Vampire cannot find a refutation of $14x \not\simeq x^2 + 49$ in reasonable time using our previous approaches [2, 3].

3 What Kind of Instances Do We Want?

Since there are possibly infinitely many instantiations, we only want to create instances with an immediate benefit. The inference rule we consider is of the form

$$\frac{P \vee D}{D\theta} \text{ theory instance} \tag{2}$$

where P contains only pure theory literals and $P\theta$ is unsatisfiable in the given theory. As P contains only pure theory literals we can use a SMT solver to find a model of $\neg P$ and use this to generate θ. In the case of clause 1 above, we pick $P = 14x \simeq x^2 + 49$ to extract $\{x \mapsto 7\}$ from the model generated by the SMT solver. From this we can conclude $p(7)$. If the SMT solver finds $\neg P$ to be unsatisfiable then P is a tautology and $P \vee D$ can instead by removed. Note that we assume that the theory is complete. The result of this approach is that we produce instances that are *shorter.*

4 Instantiation in a Saturation-Based Theorem Prover

For clauses containing inequalities, we would prefer to apply the equality resolution rule

$$\frac{s \not\simeq t \vee C}{C\theta}\ \theta = \mathsf{mgu}(s,t), \text{equality resolution}$$

instead of instantiation. For the clause $x \not\simeq 1 + y \vee p(x,y)$, equality resolution leads to $p(y+1,y)$ which is more general than $p(1,0)$ obtained from instantiating with $\{x \mapsto 0, y \mapsto 0\}$. Moreover, abstraction and instantiation may work against each other. If we consider the clause $p(1,5)$, it will be abstracted to $x \not\simeq 1 \vee y \not\simeq 5 \vee p(x,y)$. But the substitution $\{x \mapsto 1, y \mapsto 5\}$ makes $\neg(x \not\simeq 1 \vee y \not\simeq 5)$ valid. If we use it to instantiate $p(x,y)$, we re-obtain the original clause $p(1,5)$.

To prevent these effects, we introduce a further restriction on P. A literal L is *trivial* in clause C if

- L is of the form $x \not\simeq t$ and x does not occur in t
- L is a pure theory literal
- every occurrence of x in C is either $x \not\simeq t$, in a literal that is not pure or another literal trivial in C

The inference rule (2) then has the restrictions that

- P contains only pure literals
- P contains no literals trivial in $P \vee D$
- $\neg P\theta$ is valid in $\mathcal{T}$

Note that there is no requirement on P to be maximal. The more literals P has, the more precise the instantiation becomes. This comes at the risk of over-specialising, even after the removal of trivial literals.

5 Extending Unification to Help

So far we have left out the role of abstraction. In principle, the rule (2) works on any clause. However, abstracted clauses have more pure theory literals to apply the rule to. For example, the clauses

$$r(14y) \text{ and } \neg r(x^2 + 49) \vee p(x)$$

permit neither the application of resolution nor of theory instantiation. But their abstracted form

$$r(u) \vee u \not\simeq 14y \text{ and } \neg r(v) \vee v \not\simeq x^2 + 49 \vee p(x)$$

can be resolved to $u \not\simeq 14x \vee u \not\simeq x^2 + 49 \vee p(x)$ which becomes $p(7)$ after theory instantiation.

However, fully abstracting every clause has a devastating impact on proof search because it significantly increases the clause length. If we only apply theory instantiation after such a resolution step, we can modify the unification procedure to generate an abstraction on the fly. Unification with abstraction, written $\mathbf{mgu}_{abs}(s,t)$, returns a pair (θ, D), if possible, where D is a disjunction of inequalities and θ is a substitution making $(D \vee s \simeq t)\theta$ valid in a theory $\mathcal{T}$. If we can show that D are unsatisfiable then we have performed unification modulo $\mathcal{T}$. By happy coincidence, the theory instantiation rule can handle such theory constraints.

Unification with abstraction should not be applied to eagerly as it, in the limit, it can be used to make any two terms unify. For example, we would like to prevent abstraction in the case of resolving $r(1)$ with $r(2)$ because the generated constraint $1 \simeq 2$ can never be true. In general, $\mathbf{mgu}_{abs}$will never produce constraints that can not be equal in the underlying theory. We have also experimented with heuristics that decide for which subterms abstractions are generated.

The calculus can be adapted to use unification with abstraction instead of the traditional one. The resolution rule then becomes

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

where $(\theta, D) = \mathbf{mgu}_{abs}(A, A')$. The factoring, superposition and equality resolution rules can be similarly adapted.

6 Summary

We have implemented a new approach to reasoning with theories and quantifiers in a saturation-based theorem prover. This approach utilises an SMT solver to find useful instances and extends unification to produce clauses that are likely to have useful instances. We have implemented these approaches in Vampire[4], our experiments indicate that unification with abstraction is beneficial for some cases.
Acknowledgements. We describe work published by the first author, Martin Suda, and Andrei Voronkov [4].

References

[1] Laura Kovács and Andrei Voronkov. First-order theorem proving and Vampire. In *CAV 2013*, volume 8044 of *LNCS*, pages 1–35, 2013.

[2] Giles Reger, Nikolaj Bjørner, Martin Suda, and Andrei Voronkov. AVATAR modulo theories. In *GCAI 2016*, volume 41 of *EPiC Series in Computing*, pages 39–52. EasyChair, 2016.

[3] Giles Reger and Martin Suda. Set of support for theory reasoning. In *IWIL Workshop and LPAR Short Presentations*, volume 1 of *Kalpa Publications in Computing*, pages 124–134. EasyChair, 2017.

[4] Giles Reger, Martin Suda, and Andrei Voronkov. Unification with abstraction and theory instantiation in saturation-based reasoning. In *TACAS*, 2018.