Experimenting with superposition in iProver

André Duarte Konstantin Korovin

University of Manchester, {andre.duarte,konstantin.korovin}@manchester.ac.uk

Abstract: In this work we extend iProver with support for the superposition calculus. Then, we develop a flexible simplification setup that subsumes and generalises common architectures such as Discount or Otter. This also includes the concept of "immediate simplification", wherein newly derived clauses are more aggressively simplified among themselves, which can make the given clause redundant and thus its children discarded.

1 Introduction

iProver [1] is an automated theorem prover for first-order logic. It implements primarily the Inst-Gen calculus, but it also implements resolution and supports running them in combination. In this work we extend iProver with support for the superposition calculus.

Superposition is a set of inference rules that is complete for first-order logic with equality predicates only (and therefore for all first-order logic via an embedding in the former fragment). We do not present it here (see e.g. [2]).

The calculus is performed in a conventional given clause loop [3]. In iProver, it can either be run standalone, or in combination with the main instantiation calculus. In the latter mode, superposition is run simultaneously with instantiation to generate clauses for simplifications in the instantiation loop (but not to perform instantiation inferences).

2 Simplifications

Apart from the generating inferences, necessary for completeness, we can add *simplification inferences*. These are inferences where some or all of the premises are deleted. They are not required for completeness but are crucial for performance. In this work, we use the following rules (where a ~~crossed out~~ premise indicates that it can be deleted after adding the conclusion):

$$\text{Tautology deletion} \quad \frac{\cancel{l \vee \bar{l} \vee C} \qquad \cancel{t = t \vee C}}{} \tag{1}$$

$$\text{Syntactic eq. res.} \quad \frac{\cancel{t \neq t \vee C}}{} \tag{2}$$

$$\text{Subsumption} \quad \frac{\cancel{C\theta \vee D} \quad C}{} \tag{3}$$

$$\text{Subset subsumption} \quad \frac{\cancel{C \vee D} \quad C}{} \tag{4}$$

$$\text{Subsumption res.} \quad \frac{p \vee C \quad \cancel{\bar{q} \vee D}}{D} \tag{5}$$

where there exists θ such that $(p \vee C)\theta \subseteq q \vee D$.

$$\text{Demodulation} \quad \frac{l = r \quad \cancel{C[l\theta]}}{C[l\theta \mapsto r\theta]} \tag{6}$$

where $l\theta \succ r\theta$ and $\{l\theta = r\theta\} \prec C[l\theta \mapsto r\theta]$.

Light normalisation In addition, we introduce the following rule:

$$\text{Light normalisation} \quad \frac{l = r \quad \cancel{C[l]}}{C[l \mapsto r]} \tag{7}$$

which is a special case of the demodulation rule. It's advantageous to formulate this separately because it may be implemented much more efficiently than demodulation (simple replacement, no instantiation), and as such we may want e.g. to apply light normalisation wrt. all clauses but demodulation only wrt. active clauses

Simplification scheduling How these simplifications are performed can greatly impact the performance of the solver, so care is needed, and tuning this part of the solver can pay off significantly. We can choose to perform some simplifications at different times, or not at all. Additionally, some of these simplifications require auxiliary data structures (here referred to generally as 'indices') to be done efficiently, and some indices support several rules. Therefore we also need to choose which clauses to add to each indices at which stages.

For example, Otter-style loops [3] perform simplifications on clauses before adding them to the passive set. The problem with that is that the passive set is often orders of magnitude larger than the active set, therefore performance will degrade significantly as this set grows, and the system will spend most of its time performing simplifications on clauses that may not even end up being used. On the other hand, Discount-style loops [4] perform simplifications only with clauses that have been added to the active set. This has the benefit of reducing the time spent in simplifications, at the cost of potentially missing many valuable simplifications wrt. passive clauses. It is not clear where the "sweet spot" is, in terms of these setups, so we want a flexible configuration to experiment with and compare different approaches.

Immediate simplification In addition, we also introduce the idea of *immediate simplification*. The intuition is as follows. Clauses that are derived in each loop are "related" to each other. It may be beneficial to keep the set of immediate conclusions inter-simplified. Also,

throughout the execution of the program the set of generated clauses in each loop remains small compared to the set of passive or active clauses. Therefore, we can get away with applying more expensive rules that we don't necessarily want to apply on the set of all clauses (e.g. only "light" simplifications between newly derived clauses and passive clauses, but more expensive "full" simplifications among newly derived clauses). Finally, during this process, it is possible that the given clause itself becomes redundant (e.g. subsumed by one of its children). If this happens, we can add the responsible clauses to the passive set, remove the given clause from the set, and then throw away this iteration's newly generated clauses and abort the iteration and proceed to the next given clause. This may speed things up if many iterations are thus aborted.

Also, we may want to apply a distinct set of (more expensive) simplifications among the input clauses. We also take this into consideration.

Simplification setup We propose a general and flexible framework to specify how these simplifications are performed. This lets us experiment with and evaluate many different configurations. In pseudocode:

```
input_set = ∅
for i in input_clauses:
  simplify(i wrt input_set via input)
  add(i to indices_input)
add(input_set to indices_passive)

main_set = ∅
loop:
  immed_set = ∅
  given = take(clause from passive)
  simplify(given wrt main_set via
      rules_active)
  add(given to indices_active)
  for i in all generating inferences between
      given and active:
    simplify(i wrt immed_set via rules_immed)
    if given was eliminated in immed_set
        by clauses:
      add(clauses to indices_passive)
      goto loop
    simplify(i wrt main_set via rules_passive)
    add(i to indices_immed)
  add(immed_set to indices_passive)
```

where `add(clause to indices)` adds a `clause` to some simplification `indices`, and `simplify(clause wrt set via rules)` simplifies a `clause`, via a set of `rules`, by some clause(s) in `set`.

This general scheme gives great flexibility for the user to specify which simplifications are done at which stages. Namely, we can specify: to which indices are clauses added after generation (`indices_passive`), after adding to passive (`indices_active`), during immediate simplification (`indices_immed`), during input preprocessing (`indices_input`); also which simplifications are done before activation (`rules_active`), after generatuion, wrt. the main set (`rules_passive`), and wrt. the immediate set (`rules_immed`), and among input clauses (`rules_input`).

An Otter loop would be

$$\texttt{indices\_passive} = \text{all} \qquad \texttt{rules\_passive} = \emptyset$$
$$\texttt{indices\_active} = \emptyset \qquad \texttt{rules\_active} = \text{all}$$

while a Discount loop would be

$$\texttt{indices\_passive} = \emptyset \qquad \texttt{rules\_passive} = \emptyset$$
$$\texttt{indices\_active} = \text{all} \qquad \texttt{rules\_active} = \text{all}$$

with the rest $=\emptyset$. In our experiments we will test several distinct setups.

Simultaneous superposition Another improvement is the usage of "simultaneous superposition" [5]. Recall the superposition rule:

$$\frac{l = r \vee C \quad t[s] \doteq u \vee D}{(t[s \mapsto r] = u \vee C \vee D)\theta} \tag{8}$$

where $\theta = \text{mgu}(l, s)$, $l\theta \not\preceq r\theta$, $t\theta \not\preceq u\theta$, and s is not a variable. The conventional rule is that by $t[s]$ and $t[s \mapsto r]$ we mean resp. "a distinguished occurrence of s as a subterm of t" and "replacing that subterm at that position by r. We call the variant *simultaneous superposition* where we mean instead "replacing all occurrences of s in t by r". This variant is still refutationally complete.

3 Results

We integrated the simultaneous superposition calculus into iProver and evaluated it over 15 168 first-order problems in TPTP-v7.2.0. The superposition loop can solve 7375 (49%), the instantiation loop (on the previous version of iProver) 7884 (52%), and their combination can solve 8708 (57%). Therefore we can conclude that combination with superposition improved the performance of iProver over the whole TPTP.

References

[1] K. Korovin, "Inst-Gen — A Modular Approach to Instantiation-Based Automated Reasoning," in *Programming Logics* (A. Voronkov and C. Weidenbach, eds.), vol. 7797, pp. 239–270, Springer Berlin Heidelberg.

[2] J. A. Robinson, *Handbook of automated reasoning*. Elsevier MIT Press, 2001.

[3] W. McCune, "OTTER 3.3 reference manual," *CoRR*, vol. cs.SC/0310056, 2003.

[4] J. Denzinger, M. Kronenburg, and S. Schulz, "DISCOUNT — A distributed and learning equational prover," *Journal of Automated Reasoning*, vol. 18, pp. 189–198, Apr 1997.

[5] D. Benanav, "Simultaneous paramodulation," in *10th International Conference on Automated Deduction, Kaiserslautern, FRG, July 24-27, 1990, Proceedings*, pp. 442–455, 1990.