iProver – An Instantiation-Based Theorem Prover for First-Order Logic (System Description)

Konstantin Korovin*

The University of Manchester
School of Computer Science
korovin@cs.man.ac.uk

Abstract. iProver is an instantiation-based theorem prover which is based on Inst-Gen calculus, complete for first-order logic. One of the distinctive features of iProver is a modular combination of instantiation and propositional reasoning. In particular, any state-of-the art SAT solver can be integrated into our framework. iProver incorporates state-of-the-art implementation techniques such as indexing, redundancy elimination, semantic selection and saturation algorithms. Redundancy elimination implemented in iProver include: dismatching constraints, blocking non-proper instantiations and propositional-based simplifications. In addition to instantiation, iProver implements ordered resolution calculus and a combination of instantiation and ordered resolution. In this paper we discuss the design of iProver and related implementation issues.

1 Introduction

iProver is based on an instantiation framework for first-order logic Inst-Gen, developed in [3–5, 7]. We are working with clause logic and the main problem we are investigating is proving (un)satisfiability of sets of first-order clauses. The basic idea behind Inst-Gen is as follows. Given a set of first-order clauses S, we first produce a ground abstraction of S by mapping all variables into a distinguished constant, say $\bot$, obtaining a set of ground clauses $S\bot$. If $S\bot$ is unsatisfiable then S is also unsatisfiable and we are done. Otherwise, we need to refine the abstraction by adding new instances of clauses, witnessing unsatisfiability at the ground level. Instances are generated by an inference system called DSInst-Gen, which incorporates dismatching constraints (D) and semantic selection (S). We repeat this process until we obtain either (i) an unsatisfiable ground abstraction (this can be checked by any off-the-shelf SAT solver), or (ii) a saturated set of clauses, that is no non-redundant inference is applicable, and in this case completeness of the calculus [3, 7] implies that S is satisfiable. Moreover, if S is unsatisfiable and the inference process is fair, i.e., all persistent eligible inferences eventually become redundant, then completeness of the calculus guarantees that after a finite number of steps we obtain an unsatisfiable ground abstraction of S. The main ingredients which make this general scheme a basis for a useful implementation are the following.

* Supported by The Royal Society.

1. The Instantiation calculus DSInst-Gen.
2. Redundancy elimination techniques.
3. Flexible saturation strategies.
4. Combination with other calculi, such as resolution.
5. State-of-the-art implementation techniques.

In the following sections we describe these components in more detail.

2 Instantiation Calculus

In order to define our main calculus DSInst-Gen we first need to define selection functions and dismatching constraints.

Semantic Selection. Selection functions allow us to restrict applicability of inferences to selected literals in clauses. Our selection functions are based on models of the propositional abstraction of the current set of clauses. In practice, such models are generated by the SAT solver, used for the ground reasoning. A *selection function* sel for a set of clauses S is a mapping from clauses in S to literals such that $\mathsf{sel}(C) \in C$ for each clause $C \in S$. We say that sel is based on a model $I_\bot$ of $S\bot$, if $I_\bot \models \mathsf{sel}(C)\bot$ for all $C \in S$. Thus, DSInst-Gen inferences are restricted to literals, whose propositional abstraction is true in a model for the propositional abstraction of the current set of clauses.

Dismatching constraints. In order to restrict instance generation further, we consider dismatching constraints. Among different types of constraints used in automated reasoning, dismatching constraints are particularly attractive. On the one hand they provide powerful restrictions for the instantiation calculus, and on the other, checking dismatching constraints can be efficiently implemented. A *simple dismatching constraint* is a formula $ds(\bar{s}, \bar{t})$, also denoted as $\bar{s} \lhd_{ds} \bar{t}$, where $\bar{s}$, $\bar{t}$ are two variable disjoint tuples of terms, with the following semantics. A solution to a constraint $ds(\bar{s}, \bar{t})$ is a substitution σ such that for every substitution γ, $\bar{s}\sigma \not\equiv \bar{t}\gamma$, where $\equiv$ is the syntactic equivalence. We will use conjunctions of simple dismatching constraints, called *dismatching constraints*, $\wedge_{i=1}^{n} ds(\bar{s}_i, \bar{t}_i)$, where every $\bar{t}_i$ is variable disjoint from all $\bar{s}_j$, and $\bar{t}_k$, for $i \neq k$. A substitution σ is a *solution* of a dismatching constraint $\wedge_{i=1}^{n} ds(\bar{s}_i, \bar{t}_i)$ if σ is a solution of each $ds(\bar{s}_i, \bar{t}_i)$, for $1 \leq i \leq n$. We will assume that for a constrained clause $C \mid [\ \wedge_{i=1}^{n} ds(\bar{s}_i, \bar{t}_i)\ ]$, the clause C is variable disjoint from all t_i, $1 \leq i \leq n$. A *constrained clause* $C \mid [\ \varphi\ ]$ is a clause C together with a dismatching constraint φ. An unconstrained clause C can be seen as a constrained clause with an empty constraint $C \mid [\ ]$. Let S be a set of constrained clauses, then $\widetilde{S}$ denotes the set of all *unconstrained clauses* obtained from S by dropping all constraints.

Proper instantiators. Another restriction on the instantiation calculus is that only proper instantiations need to be considered. A substitution θ is called a *proper instantiator* for an expression (literal, clause, etc.) if it maps a variable in this expression into a non-variable term.

DSInst-Gen Calculus. Now we are ready to formulate the DSInst-Gen calculus. Let S be a set of constrained clauses such that $\widetilde{S}\bot$ is consistent and let sel be a selection function based on a model $I_\bot$ of $\widetilde{S}\bot$. Then, the DSInst-Gen inference system is defined as follows.

DSInst-Gen

$$\frac{L \vee C \mid [\,\varphi\,] \quad \overline{L'} \vee D \mid [\,\psi\,]}{L \vee C \mid [\,\varphi \wedge \bar{x} \lhd_{ds} \bar{x}\theta\,] \quad (L \vee C)\theta}$$

where (i) $\bar{x}$ is a tuple of all variables in L, and
(ii) θ is the most general unifier of L and L', wlog. we assume that the domain of θ consist of all variables in L and L', and the range of θ is variable disjoint from the premises, and
(iii) $\mathsf{sel}(L \vee C) = L$ and $\mathsf{sel}(\overline{L'} \vee D) = \overline{L'}$, and
(iv) θ is a proper instantiator for L, and
(v) $\varphi\theta$ and $\psi\theta$ are both satisfiable dismatching constraints.

DSInst-Gen is a replacement rule, which is replacing the clause in the left premise by clauses in the conclusion. The clause in the right premise can be seen as a side condition. In [7] we have shown that DSInst-Gen calculus is sound and complete. DSInst-Gen is the main inference system implemented in iProver.

3 Redundancy Elimination

In [3] an abstract redundancy criterion is given which can be used to justify concrete redundancy elimination methods [7], implemented in iProver. In order to introduce redundancy notions we need some definitions. A *ground closure*, denoted as $C \cdot \sigma$, is a pair consisting of a clause C and a substitution σ grounding for C. Ground closures play a similar role in our instantiation framework as ground clauses in resolution. Let S be a set of clauses and C be a clause in S, then a ground closure $C \cdot \sigma$ is called a *ground instance* of S and we also say that the closure $C \cdot \sigma$ is a *representation* of the clause $C\sigma$ in S. A *closure ordering* is any ordering $\succ$ on closures that is total, well-founded and satisfies the following condition. If $C \cdot \sigma$ and $D \cdot \tau$ are such that $C\sigma = D\tau$ and $C\theta = D$ for some proper instantiator θ, then $C \cdot \sigma \succ D \cdot \tau$.

Let S be a set of clauses. A ground closure $C \cdot \sigma$ is called *redundant* in S if there exist ground closures $C_1 \cdot \sigma_1, \ldots, C_k \cdot \sigma_k$ that are ground instances of S such that, (1) $C_1 \cdot \sigma_1, \ldots, C_k \cdot \sigma_k \models C \cdot \sigma$, and (2) $C \cdot \sigma \succ C_i \cdot \sigma_i$, for each $1 \leq i \leq k$. A clause C (possibly non-ground) is called redundant in S if each ground closure $C \cdot \sigma$ is redundant in S. This abstract redundancy criterion can be used to justify many standard redundancy eliminations such as tautology elimination and strict subsumption, where the subsuming clause has strictly less literals than the subsumed.

Global Subsumption. One of the novel simplifications implemented in iProver is based on utilising propositional reasoning [7]. First, let us consider simplifications of ground clauses and then later we show how to extend this to the general case. Consider a set of clauses S. Let C be a ground clause we would like to simplify wrt. S. If we can show that a strict subclause $D \subsetneq C$ is entailed by S, then we can simplify C by D. Since our ground abstraction $S\bot$ is implied by S we can use $S\bot$ to approximate the entailment above. In particular, if we can show that $S\bot \models D$, this can be checked by the SAT solver, then C can be simplified by D. We call this simplification *global propositional subsumption* wrt. $S\bot$.

Global propositional subsumption

$$\frac{D \vee D'}{D}$$

where $S\bot \models D$ and D' is not empty.

Global propositional subsumption is a simplification rule, which allows us to remove the clause in the premise after adding the conclusion. Let us note that although the number of possible subclauses is exponential wrt. the number of literals, in a linear number of implication checks we can find a minimal wrt. inclusion subclause $D \subsetneq C$ such that $S\bot \models D$, or show that such a subclause does not exist. In [7] we have shown that global propositional subsumption generalises many known simplifications, such as strict subsumption and subsumption resolution.

Now we describe an extension of this idea to the general non-ground case (see [7] for details). First, we note that in the place of $S\bot$ we can use any ground set S_{gr}, implied by S. Let Σ_C be a signature consisting of an infinite number of constants not occurring in the signature of the initial set of clauses Σ. Let Ω be a set of injective substitutions mapping variables to constants in Σ_C. We call C' an Ω-instance of a clause C if $C' = C\gamma$ where $\gamma \in \Omega$. Let us assume that for any clause $C \in S$ there are some Ω-instances of C in S_{gr}. In [7] we have shown that if some Ω-instance of a given clause D is implied by S_{gr}, then S implies D. Now we can formulate an extension of global subsumption to the non-ground case:

Global subsumption (non-ground)

$$\frac{(D \vee D')\theta}{D}$$

where $S_{gr} \models D\gamma$ for some $\gamma \in \Omega$, and D' is not empty.
Global subsumption is one of the main simplifications implemented in iProver.

4 Saturation Algorithm: The Inst-Gen Loop

Now we are ready to put inferences and simplifications together into a saturation algorithm, called the Inst-Gen Loop, which is implemented in iProver.[1] As shown in Fig 1, the Inst-Gen Loop is a modification of the standard given clause algorithm, which accommodates propositional reasoning. Let us overview key components of the Inst-Gen Loop and how they are implemented in iProver. One of the main ideas of the given clause algorithm is to separate clauses into two sets, called Active and Passive, with the following properties. The set of Active clauses is such that all non-redundant inferences between clauses in Active are performed (upon selected literals). The set of Passive clauses are the clauses waiting to participate in inferences. Initially, the input clauses are preprocessed and groundings of the preprocessed clauses are added to the

[1] iProver is available at http://www.cs.man.ac.uk/~korovink/iprover/

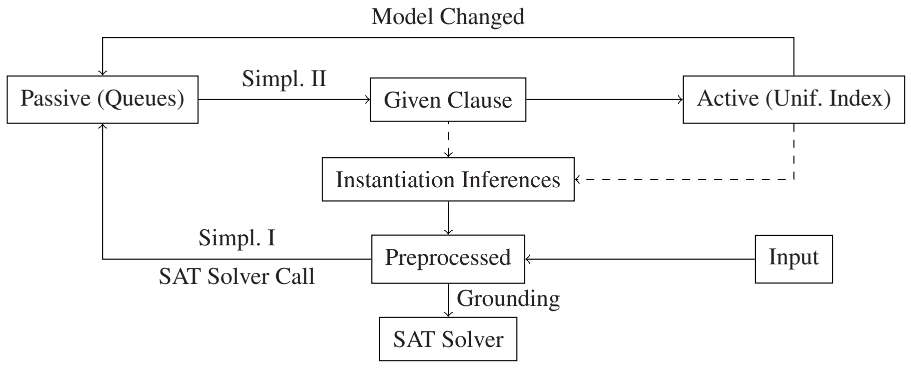

Fig. 1. The Inst-Gen Loop

SAT solver. Preprocessing currently consists of optional splitting without backtracking on variable disjoint subclauses [8]. The given clause algorithm consists of a loop and at each loop iteration the following actions are executed. First, a clause is taken from the Passive set, called the Given Clause. Then, all inferences between the Given Clause and the clauses in Active are performed and the Given Clause is moved to Active. Finally, all newly derived clauses are preprocessed and groundings of the obtained clauses are added to the SAT solver. The SAT solver is called in regular, user-defined intervals until either (i) unsatisfiability is found, in this case the input set of clauses is unsatisfiable, or (ii) all clauses are in Active, in this case the input set of clauses is satisfiable. Let us describe the components of the Inst-Gen Loop.

Passive. The Passive set are the clauses waiting to participate in inferences. Experience with resolution-based systems shows that the order in which clauses are selected for inferences from Passive is an important parameter. Usually, preference is given to the clauses which are heuristically more promising to derive the contradiction, or to the clauses on which basic operations are easier to perform. In iProver, Passive clauses are represented by two priority queues. In order to define priorities we consider numerical/boolean parameters of clauses such as: number of symbols, number of variables, age of the clause, number of literals, whether the clause is ground, conjecture distance and whether the clause contains a symbol from the conjecture (other than equality). Then, each queue is ordered by a lexicographic combination of orders defined on parameters. For example, if a user specifies an iProver option: '- *-inst_pass_queue1 [+age; -num_symb;+ground]*', then priority in the first queue is given to the clauses generated at the earlier iterations of the Inst-Gen Loop (older clauses), then to the clauses with fewer number of symbols and finally to ground clauses. The clauses are taken from the queues according to a user-specified ratio.

Selection functions. Selection functions are based on the current model $I_\perp$ of the propositional abstraction of the current set of clauses. A clause can have several literals true in $I_\perp$, and the selection function can be restricted to choose one of them. Selection functions are defined by priorities based on a lexicographic combination of the literal parameters. The following parameters currently can be selected by the user: sign, ground, num_var, num_symb, and split. For example if a user specifies an iProver option: '- *-inst_lit_sel [+sign;+ground;-num_symb]*'. Then, priority (among the literals in the clause true in $I_\perp$) is given to the positive literals, then to the ground literals and then to literals with a fewer number of symbols.

Active. After the Given Clause is selected from Passive all eligible inferences between the Given Clause and clauses in Active should be performed. A unification index is used for efficient selection of clauses eligible for inferences. In particular, Active clauses are indexed by selected literals. The unification index implemented in iProver is based on non-perfect discrimination trees [6]. Let us note that since the literal selection is based on a propositional model (of a ground abstraction of the current set of clauses), selection can change during the Inst-Gen Loop iterations. This can result in moves of clauses from Active to Passive, as shown in Fig 1. Such moves can be a source of inefficiency and we minimise them by considering the selection change only in clauses participating in the current inference.

Instantiation Inferences. iProver implements DSInst-Gen calculus. In particular, constrained clauses, dismatching constraint checking and semantic-based literal selections are implemented.

Redundancy elimination. In addition to dismatching constraints, global subsumption for clauses with variables and tautology elimination are implemented.

Grounding and SAT Solver. Newly derived clauses are grounded and added to the propositional solver. Although, in our exposition we used the designated constant $\perp$ for grounding, all our arguments remain valid if we use any ground term in place of $\perp$. In particular, for grounding, iProver selects a constant with the greatest number of occurrences in the input set of clauses. After grounding, clauses are added to the propositional solver. Currently, iProver integrates MiniSat [2] solver for propositional reasoning. Incrementality of MiniSat is essential for global subsumption.

Learning Restarts. Initially, the propositional solver contains only few instances of the input clauses, and therefore selection based on the corresponding propositional model may be inadequate. Although the model and selection can be changed at the later iterations, by that time, the prover may have consumed most of the available resources. In order to overcome this, iProver implements restarts of the saturation process, keeping the generated groundings of clauses in the SAT solver. After each restart, the propositional solver will contain more instances of clauses, this can help to find a better literal selection. In addition, after each restart, global subsumption becomes more powerful.

Combination with Resolution. Instantiation, by itself, is not very well suited for generating small clauses which can be later used in simplifications such as (global) subsumption. For this, iProver implements a complete saturation algorithm for ordered resolution. The saturation algorithm for resolution is based on the same data structures as Inst-Gen Loop and implements a number of simplifications such as forward and backward subsumption (based on a vector index [11]), subsumption resolution, tautology deletion and global subsumption. Resolution is combined with instantiation by sharing the propositional solver. In particular, groundings of clauses generated by resolution are added to the propositional solver and propositional solver is used for global subsumption in both resolution and instantiation saturation loops. The user can select between combination of instantiation with resolution, pure instantiation and pure ordered resolution.

5 Implementation Details and Evaluation

iProver is implemented in a function language OCaml and integrates MiniSat solver [2] for propositional reasoning, which is implemented in C/C++. iProver v0.3.1 was evaluated on the standard benchmark for first-order theorem provers – TPTP library v3.2.0[2]. Currently, iProver does not have a built-in clausifier and we used E prover[3] for clausification. Experiments were run on a cluster of PCs with CPU 1.8GHz, Memory 512 Mb, Time Limit 300s, OS Linux v2.6.22. Out of 8984 problems in the TPTP library, iProver (single strategy) solved 4843 problems: 4000 unsatisfiable and 843 satisfiable. Problems in TPTP are rated from 0 to 1, problems with the rating 0 are easy and problems with the rating 1 cannot be solved by any state-of-the-art automated reasoning system. iProver solved 7 problems with the rating 1, and 27 with rating greater than 0.9. We compare iProver v0.2 with other systems, based on the results of the CASC-21 competition, held in 2007 [12]. In the major FOF devision, iProver is in the top three provers along with established leaders Vampire [9] and E [10]. In the effectively propositional division (EPR), iProver is on a par with the leading system Darwin [1]. We are currently working on integrating equational and theory reasoning into iProver.

References

1. Baumgartner, P., Fuchs, A., Tinelli, C.: Implementing the model evolution calculus. International Journal on Artificial Intelligence Tools 15(1), 21–52 (2006)
2. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004)
3. Ganzinger, H., Korovin, K.: New directions in instantiation-based theorem proving. In: Proc. 18th IEEE Symposium on LICS, pp. 55–64. IEEE Computer Society Press, Los Alamitos (2003)
4. Ganzinger, H., Korovin, K.: Integrating equational reasoning into instantiation-based theorem proving. In: Marcinkowski, J., Tarlecki, A. (eds.) CSL 2004. LNCS, vol. 3210, pp. 71–84. Springer, Heidelberg (2004)
5. Ganzinger, H., Korovin, K.: Theory Instantiation. In: Hermann, M., Voronkov, A. (eds.) LPAR 2006. LNCS (LNAI), vol. 4246, pp. 497–511. Springer, Heidelberg (2006)
6. Graf, P.: Term Indexing. LNCS, vol. 1053. Springer, Heidelberg (1996)
7. Korovin, K.: An invitation to instantiation-based reasoning: From theory to practice. In: Podelski, A., Voronkov, A., Wilhelm, R. (eds.) Volume in memoriam of Harald Ganzinger (to appear) (invited paper)
8. Riazanov, A., Voronkov, A.: Splitting without backtracking. In: Proc. of the 17 International Joint Conference on Artificial Intelligence (IJCAI 2001), pp. 611–617. Morgan Kaufmann, San Francisco (2001)
9. Riazanov, A., Voronkov, A.: The design and implementation of VAMPIRE. AI Communications 15(2-3), 91–110 (2002)
10. Schulz, S.: E - a brainiac theorem prover. AI Commun. 15(2-3), 111–126 (2002)
11. Schulz, S.: Simple and Efficient Clause Subsumption with Feature Vector Indexing. In: Sutcliffe, G., Schulz, S., Tammet, T. (eds.) Proc. of the IJCAR-2004 Workshop on Empirically Successful First-Order Theorem Proving, Cork, Ireland. ENTCS. Elsevier Science, Amsterdam (2004)
12. Sutcliffe, G.: CASC-21 proceedings of the CADE-21 ATP system competition (2007)

[2] http://www.cs.miami.edu/~tptp/
[3] http://www.eprover.org/