First-order reasoning in Yices2

Master Thesis in Computer Science
Simon Cruanes*
supervised by Bruno Dutertre[†], Viktor Kuncak[‡], Benjamin Werner[§]
SRI International, Menlo Park, CA

September 2011 – March 2012

March 15, 2012

Abstract

SMT solvers are powerful tools to solve logic problems, with built-in support of theories such as linear arithmetic. YICES2 is a state-of-the-art SMT solver, but it is unable to deal with quantified formulæ. In this thesis, we present different techniques we have explored to make first-order reasoning possible in YICES2, and we describe their implementation. In addition to instantiation mechanisms inspired from the widely used "heuristic E-matching", we discuss the coupling of the ground satisfiability solver with a superposition prover — the technology behind many high performance first-order provers — and the implementation of the latter.

*École polytechnique, France, École Polytechnique Fédérale de Lausanne, Suisse
[†]SRI International, Menlo Park, California, USA
[‡]École Polytechnique Fédérale de Lausanne, Suisse
[§]École polytechnique, France

Contents

List of Figures

List of Tables

1 Introduction

In a world where a car can embed dozens of millions of lines of code [1], reliability of software is crucial. For critical systems, since testing does not provide enough guarantees, formal methods based on logic are commonly used.

SMT solvers are a powerful technology widely used to solve difficult problems that have a complex propositional structure in addition to theories such as linear arithmetic or bitvectors. YICES2 (hereafter called YICES) is a state-of-the-art SMT-solver developed in the Computer Science Lab of SRI International. YICES currently supports quantifier-free linear arithmetic, uninterpreted functions, arrays, and bitvectors.

However, many practical problems, such as proofs obligations that come from software verification, express some domain-specific axioms using quantified formulæ; some examples are list axioms like $\forall x.\forall y.car(cons(x, y)) = x$, frame axioms, representations of type systems, etc. On one hand, the historical approach used by SMT solvers such as YICES1 [DDM06], Z3 [dMB08a] or CVC3 [BT07] is based on *heuristic instantiation.* On the other hand, powerful general-purpose first-order provers exist, the most competitive of which (Vampire[RV], E [Sch02], SPASS [WSH$^+$07]) are based on systematic proof search using the superposition calculus.

We present, in this thesis, our attempt to integrate state-of-the-art first-order reasoning based on superposition into YICES. Although some research has already been done in this domain [dMB08b] [LT], our approach is different because we tried to keep the SMT-solver's main loop separate from the superposition solver's main loop. We detail the challenges raised by the presence of *interpreted theories* in a first-order deduction perspective. We also present some techniques we have used to combine the work of both solvers, by instantiations of the clauses generated by the superposition prover, guided by the partial models of the SMT solver.

The thesis is organized as follows: we first give some formal definitions and informal descriptions of the technologies involved, then we devote a section to general first-order algorithms, and one to the superposition prover and its implementation. After that, two algorithms for instantiation are presented, followed by some experimental results and a conclusion.

Acknowledgments

I am most grateful to the people who have made my visit at SRI International not only possible, but also a passioning experience (in no particular order): Jean-Christophe Filliâtre for his advices, Lori Truitt for her time and her smile, Bruno Dutertre — obviously — for being a great supervisor, explaining me subtle details of YICES' implementation and all the discussions about design choices we had, Natarajan Shankar and Sam Owre for their ideas and discussions about

[1] `http://spectrum.ieee.org/green-tech/advanced-cars/this-car-runs-on-code`

logic, programming and distributed asynchronous forward-chaining, and my supervisors from Switzerland and France Viktor Kuncak and Benjamin Werner. I also thank my sister Tiphaine and my parents Françoise and Jean-Michel for their support, rock-solid even from the opposite side of the Earth.

I would also like to thank the authors of the free software I have been using extensively during this master's thesis and before: the authors and contributors of vim, GNU/linux, python, zsh, ctags — invaluable to crawl through such a big project, graphviz (to print graphs of data structures using `dot`) and many more.

2 Definitions and Background

We dedicate this section to some general definitions for first-order logic, and to some background explanations about SMT solvers.

2.1 Signature and Terms

Signatures: A *signature* is a pair of finite sets of objects (F, S) with a function $sort : F \to S$. F is the set of *function symbols* and S is the set of *sorts*, or *types*. S must contain the function sort $\tau_1, \tau_2, \dots, \tau_n \to \tau$ whenever $\tau_1, \dots, \tau_n, \tau$ are in S; symbols with sort τ_f have an *arity* of n. A judgement $sort(f) = \tau$ will also be written as $f : \tau$. For any function sort $\tau = \tau_1, \dots, \tau_n \to \tau$, the tuple $(\tau_1, \dots, \tau_n)$ is called the *domain* of the sort, and τ is called the *range* of τ. A symbol of arity 0 is called a *constant*.

Terms: Given the set of *variables* $X = \bigcup_{\tau \in S} X_\tau$, where X_τ is the infinite set of variables of sort τ, we inductively define the set of *terms* $\mathscr{T}(F, S, X)$ as the minimal set closed by the rules

$$\frac{x \in X}{x \in \mathscr{T}(F, S, X)}$$

$$\frac{t_1 : \tau_1 \ \dots \ t_n : \tau_n \qquad f : (\tau_1, \dots, \tau_n \to \tau)}{f(t_1, \dots, t_n) : \tau \in \mathscr{T}(F, S, X)}$$

We will usually write a, b, c for constant terms, f, g for non-constant function symbols, x, y, z for variables, s, t, u, v for arbitrary terms, and τ for sorts. We will say that a term is *ground* if it does not contain any variable.

Positions: A *position* in a term t is either Λ (the empty position) or $i.p$ if $t = f(t_1, \dots, t_n)$ and p is a position in some t_i with $i \in \{1, \dots, n\}$. Intuitively, a position is a path in the tree representation of a term, with $i.p$ meaning that the path goes through the i^{th} subterm of the current terms, and continues recursively with the subpath p. We write $t|_p$ for the subterm of t at position p (in particular, $t|_\Lambda = t$) and $t[p \leftarrow s]$ for the term t where the subterm at position p has been replaced by s.

2.2 First-Order Logic

Formulæ: For a given signature (F, S) that contains a given sort $bool \in S$ of propositions, we define the set of *atomic formulæ* as the set that contains all symbols of F with arity 0 and sort $bool$, all $\neg a$ where a is an atomic formula, and all terms $p(t_1, \dots, t_n)$ where $t_i : \tau_i \in \mathscr{T}(F, S, X)$ and $p : (\tau_1, \dots, \tau_n \to bool) \in F$. The set of *formulæ* $\mathscr{F}$ is defined recursively as containing all atomic formulæ, all $a \vee b$, $a \wedge b$, $a \Rightarrow b$, $a \Leftrightarrow b$, where a and b are formulæ, and all $\forall x.a$ and $\exists x.a$

where $x \in X$ and a is a formula. Classical first-order logic forbids terms with a sort $\tau_1, \dots, \tau_n \to \tau$ where $\tau_i = bool$ for some i. If t is a subterm of $\forall x.a$, then x is said to be *universally quantified* in t (conversely, a subterm of $\exists x.a$ is said to be *existentially quantified*). The term $\neg\neg a$ is the same term as a.

Equations: Equality as a theory is syntactically denoted by unordered pairs $s \simeq t$ — also named *equations*. Note that $s \simeq t$ and $t \simeq s$ are indistinguishable. The negation of an equation is denoted as $s \not\simeq t$, and has the same meaning as $\neg s \simeq t$. Equations are atomic formulæ, we always assume that the function symbols include equality: for each sort τ, there exists a symbol $\simeq_\tau : (\tau, \tau \to bool)$. A *literal* is a signed equation; we will write $s \dot{\simeq} t$ for a literal whose sign does not matter. Usual predicates, like $p(a, b)$ will be, in some context — for the superposition calculus, in particular — written as equations $p(a, b) \simeq \top$ where $\top : bool$, also named *true*, is a special symbol[2].

Free variables: The set of *free variables* of a term t (extended to formulæ in the trivial way), also noted $vars(t)$, is the set of variables that have occurrences in t that are not bound by any quantifier. For instance, $vars(\forall x.p(x, f(y))) = \{y\}$. A formula is said to be closed if it has no free variables. In many cases, non-ground terms and formulæ will be implicitly universally quantified. Variables that are not free are *bound*, and can be renamed (alpha equivalence).

Substitutions: A *substitution* is a function $\sigma : X \to \mathcal{T}(F, S, X)$ where $dom = \{x \in X | \sigma(x) \neq x\}$ is finite. dom is called the *domain* of σ, and $\sigma(dom)$ is its *range*. Substitutions only apply to free variables, since bound variables can be renamed.

Theories: A *theory* $\mathcal{T}$ is a set of formulæ, also called *axioms*. In the context of this thesis, a theory may also be *implicit*, i.e., represented as a decision procedure for this theory rather than by its axioms (the SMT solver combines such decision procedures, e.g., the Simplex algorithm for the theory of linear arithmetic).

Models: A *model* — or *interpretation* — for a given signature (F, S), is a pair $I = (D, [\![.]\!])$. D, also called *domain*, is a collection of sets of arbitrary objects D_s for each kind $s \in S$, and $[\![.]\!]$ is a function $\mathcal{T}(F, S, X) \to D$ which preserves kinds, i.e., $t \in \mathcal{T}(F, S, X) : s \in S \Rightarrow [\![t]\!] : S$. $[\![.]\!]$ must be a homomorphism: $[\![f(t_1, \dots, t_n)]\!] = [\![f]\!]([\![t_1]\!], \dots, [\![t_n]\!])$. Images of predicate symbols (function symbols with Boolean range) by $[\![.]\!]$ are relations in D, and images of logical connectives are the same

[2] this notation, strictly speaking, is not classical first-order logic, but one could use a sort $bool'$ for all predicates, so that $=_{bool'} : bool', bool' \to bool$. $\top$ would be of sort $bool'$.

connectives. We write $I \models F$ if $[\![F]\!] = [\![\top]\!]$, for a formula F.

$$\begin{aligned}
[\![a \vee b]\!] &= [\![a]\!] \cup [\![b]\!] \\
[\![a \wedge b]\!] &= [\![a]\!] \cap [\![b]\!] \\
[\![\neg a]\!] &= [\![a]\!]^c \\
[\![\forall \overline{x}.a]\!] &= \bigwedge_{\overline{t} \in D} [\![a[\overline{t}/\overline{x}]]\!] \\
[\![\exists \overline{x}.a]\!] &= \bigvee_{\overline{t} \in D} [\![a[\overline{t}/\overline{x}]]\!]
\end{aligned}$$

An interpretation I is a model of a theory $\mathscr{T}$ (or "I satisfies $\mathscr{T}$") if for all axioms $F \in \mathscr{T}$, we have $I \models F$. We can then write $I \models \mathscr{T}$. A theory is said to be *satisfiable* if at least one model satisfies it, otherwise it is said *unsatisfiable*. This notion of *satisfiability* is central for theorem proving, since a formula is a *theorem* in the theory $\mathscr{T}$ if the theory $\mathscr{T} \uplus \{\neg F\}$ is unsatisfiable (of course, in an inconsistent theory, every formula is a theorem).

The first-order provers based on saturation use *Herbrand models*, which are models in which the domain is the set of ground terms $\mathscr{T}(F, S)$ of all sorts that contain at least one ground term.

2.3 Propositional Satisfiability

The satisfiability of propositional logic (without quantifiers or variables) has been an active domain of research for decades. It is known to be decidable, although NP-complete, and most complete solvers[3] are based on the Davis-Putnam-Logemann-Loveland (DPLL) procedure [DLL62]. This procedure relies on the reduction of a formula to Conjunctive Normal Form — we will see later how this operation is performed. The Conjunctive Normal Form, or CNF, is a formula of the form $\bigwedge_i \bigvee_j F_{ij}$ where F_{ij} are literals. Each disjunction is called a *clause*.

The DPLL algorithm explores the space of Boolean assignments to literals — since there are no variables, there is a fixed, finite number of literals — by choosing a value for a literal, propagating the consequences of this choice, and do recursively the same with another literal. Following [NOT06], we define briefly this procedure as an abstract transition system in Figure 1. We will not detail the improvements of this procedure, the curious reader is invited to refer to, e.g. [dMDS07] for more details. In a nutshell, those improvements are *clause learning* (when a conflict occurs, a sequence of resolution steps builds a new clause that prevents the same conflict to occur ever again), *backjumping* (rather than backtracking just to the last assignment, the conflict clause is used to prune a larger portion of the search tree), *two-watched literals scheme* (an efficient implementation of unit propagation) and VSIDS heuristics [MMZ+01] (conflict-driven literals selection heuristics). Those improvements have led to orders of

[3]There also exist incomplete solvers, that are not guaranteed to find the answer, mostly based on stochastic methods.

magnitudes gains in the speed of solvers, which make them a very powerful tool to solve NP-complete problems.

In Figure 1, states are of the form $M||F$ where M is a *partial model*, i.e., a set of literals, and F is a set of clauses. All literals in M must occur in some clause of F. Transitions are non-deterministic, which means that their order does not matter (this choice is typically left to heuristics). If no rule applies and the solver is not in the *unsat* state, then the problem is satisfiable. If the solver enters the *unsat* state, then the problem is unsatisfiable. Literals l^d are called *decision literals*, the other literals are said to be *propagated*. We did not include the rule for pure literals elimination, as it is typically done in a simplification preprocessing step.

UnitPropagate	$M \Vert F, C \vee l$	$\rightsquigarrow$	$M\ l \Vert F, C \vee l$	if $M \models \neg C$, l undefined in M
Decide	$M \Vert F$	$\rightsquigarrow$	$M\ l^d \Vert F$	if l or $\neg l$ occurs in some clause, l undefined in M,
Unsat	$M \Vert F, C$	$\rightsquigarrow$	unsat	if $M \models \neg C$, M contains no decision literal
Backtrack	$M\ l^d\ N \Vert F, C$	$\rightsquigarrow$	$M\ \neg l \Vert F$	if $M\ l^d\ N \models \neg C$, N contains no decision literal

Figure 1: Abstract DPLL procedure

2.4 Satisfiability Modulo Theory

Although propositional satisfiability can be used for a wide domain of problems, it has a poor expressiveness for problems containing equality, arithmetic, or other theories used for software verification. Their explicit encoding as propositions can be error-prone and heavyweight, and the structure of the problem is lost. Therefore, more complex systems called *SMT solvers* (SMT standing for "satisfiability modulo theories") have been developed on top of propositional satisfiability solvers (called *SAT solvers*). Given a theory (or conjunction of theories) T, a formula F is said *satisfiable modulo T* if $F \wedge T$ is satisfiable. Consequently, if T is not consistent, then no formula can be satisfiable modulo T.

A standard input format, SMTLIB [RLT06], has been defined, to support a competition between SMT solvers[4]. It defines a set of theories, and a set of logics — a "logic" being a conjunction of theories, in this context. The usual theories for SMT solvers are:

Equality: equality between any pair of terms. Solvers usually use a structure called *E-graph*, which is responsible for maintaining the current *congruence closure* (the set of equalities containing the current ground equalities, transitively closed by the axioms of the theory). The only interpreted symbol in this theory is $\simeq$. Here are the axioms for this theory:

[4]`http://www.smtcomp.org`

Reflexivity	$x \simeq x$
Transitivity	$x \simeq y \wedge y \simeq z \Rightarrow x \simeq z$
Symmetry	$x \simeq y \Rightarrow y \simeq x$
Congruence	$x_1 \simeq y_1 \wedge \cdots \wedge x_n \simeq y_n \Rightarrow f(x_1, \ldots, x_n) \simeq f(y_1, \ldots, y_n)$

Linear Arithmetic: arithmetic on terms t that follow the grammar

$$t ::= f(t, \ldots, t) | t \leq t \mid t < t \mid t \simeq t \mid t + t \mid t - t \mid k \mid t * k$$

where f represents a function symbol, and k is an integer or a rational constant (for instance, the term 0). The associated theory solver used in YICES for this is based on the Simplex algorithm.

Arrays: functional arrays. The theory will be described in Section 4.3.

Bit-vectors: fixed-length arrays of bits, with several bitwise and arithmetic operations. Bitvectors are useful, for instance, to model the machine representation of integers (such as the type `int` of the C language).

To be handled by an SMT solver, any conjunctions of literals whose interpreted symbols belong only to a given theory must be decidable by the theory. Different theories can be combined if their respective interpreted symbols are disjoint. YICES uses the Nelson-Oppen framework [NO78] to combine different theories. Nelson-Oppen can be summarized as follows: formulæ are *purified*, by introducing names for subterms that belong to different theories — e.g., $g(f(x)+1) < a$ becomes $e_1 < a,\ e_1 \simeq g(e_2),\ e_2 = e_3 + 1,\ e_3 = f(x)$. Theory solvers communicate through equalities between opaque constants, each constant being interpreted by at most one theory solver (in the previous example, $e1$ would be interpreted by the arithmetic solver). Theories can be combined only if they are *stably-infinite* (i.e., if any T-satisfiable formula has an infinite model); in this case, a theorem of [NO78] states that if F_1 is satisfiable in T_1, and F_2 is satisfiable in T_2, then $F_1 \wedge F_2$ is satisfiable in the combined theory $T_1 \uplus T_2$. SMT solvers typically rely on a combination of DPLL and theory-specific decision procedures, called *theory solvers*.

The DPLL propositional solver is responsible for choosing assignments (valuation) for the literals. However, in addition to the usual unit clause propagation, it also asks theory solvers to check the consistency of the current partial model. A theory solver can propagate new literals (e.g., if $a \simeq b$ has been asserted by the propositional DPLL, the equality solver may propagate the literal $f(a) \simeq f(b)$) or trigger a conflict. In this case, it will have to provide a conflict clause (for the integration with clause learning). For more details, the reader is referred to several survey articles [dMDS07] [NOT06].

3 General First-Order Algorithms

3.1 Unification and Matching

Unification and *Matching* are two fundamental operations for any first-order prover. Given two terms a and b, a substitution σ is called a *unifier* of a and b if $\sigma(a) = \sigma(b)$. In first-order logic, if such a substitution exists for two given terms, then there is a unique (up to renaming) minimal such substitution, called the *most general unifier* of a and b, or $mgu(a, b)$. The unification problem is then the problem of finding this substitution, if it exists (in which case a and b are *unifiable*). We say that *a matches b*, or that b is a generalization of a, if there exist σ such that $\sigma(b) = a$.

Unification

Although some linear algorithms exist [MM82], we implemented a relatively simple almost-linear unification algorithm [BS01], exploiting the perfect sharing of terms in YICES. The algorithm is based on a improved *union-find* structure, which gives easy access to the elements of each equivalence class and priorities for merging, in addition to the two operations `union` and `find`. We give a brief description of this data structure before detailing the unification algorithm itself.

union: `union(a, b)` merges the equivalence classes of `a` and `b` in $O(1)$. In addition, we can specify a *priority* for `a` and `b` with the invariant that the representative of any equivalence class has the highest priority within its class.

find: `find(a)` returns the representative of the equivalence class to which `a` currently belongs, in $O(\alpha(n))$ where α is the inverse of the Ackermann function (almost linear in practice).

The unification algorithm `unify(a,b)` consists in two successive phases:

1. recursively build bindings between subterms of `a` and `b`. More precisely, given a set of pairs of terms (each pair representing a unification constraint), it picks up a pair (u, v), apply $union(u, v)$ in the union-find structure, and if $u = f(u_1, \ldots, u_n)$ and $v = f(v_1, \ldots, v_n)$ it adds the pairs $(u_1, v_1), \ldots, (u_n, v_n)$ to the set of pairs to be processed. Variables have a lower priority in the union-find than other terms, so that the root of each equivalence class containing at least one non-variable term is not a variable.

2. traverse the union-find to build the unifier. This second phase also performs occurs-check by looking for cycles in the graph of terms in which terms in the same equivalence class are linked together.

Although this algorithm has a good asymptotic complexity — almost linear in the size of the terms to unify — a simpler recursive version that does not exploit the DAG structure of terms eventually proved to be better. We think that the initialization overhead of the almost linear algorithm presented above is what makes it more costly. Pseudo-code for the simple unification algorithm is shown in Figure 2. Both algorithms assume that the set of variables of the two terms are disjoint — we will see later the influence of this restriction on the representation of clauses.

```
-- term representation (constants are Apply with empty arg list)
data Term = Var Symbol | Apply Symbol [Term]

-- apply substitution to term
applySubst :: Subst -> Term -> Term

-- add a binding to the substitution
addSubst :: Subst -> Term -> Term -> Subst

-- replace variables by their current binding (if any)
unify :: Term -> Term -> Subst -> Maybe Subst
unify x y sigma = if x' == y'
  then Just sigma           -- trivial case
  else unify' x' y' sigma   -- recursive case
 where x' = case x of Var _ -> applySubst sigma x
                      _ -> x
       y' = case y of Var _ -> applySubst sigma y
                      _ -> y

-- recursive unification
unify' (Apply f argsLeft) (Apply g argsRight) sigma
  | f == g                = let matchArg s (l,r) = match l r s in
                            foldM matchArg sigma (zip argsLeft argsRight)
                            -- unify subterms pairwise
  | otherwise             = fail "different function symbols"
unify' x@(Var _) y sigma  = Just $ addSubst sigma x y   -- bind x -> y
unify' x y@(Var _) sigma  = Just $ addSubst sigma y x   -- bind y -> x
unify' _ _ _              = fail "incompatible"


match (Apply f argsLeft) (Apply g argsRight) sigma
  | f == g      = let matchArg s (l,r) = match l r s in
                  foldM matchArg sigma (zip argsLeft argsRight)
  | otherwise   = fail "different function symbols"
```

Figure 2: Unification Algorithm

Matching

The matching algorithm is a simple recursive function that carries a substitution. Its pseudo-code (in Haskell) is given in Figure 3. The algorithm is very similar to the one we used for unification, but simpler because only the variables in the pattern need to (and can) be bound in the substitution.

```
-- match terms (using the same term representation as for unification)
match :: Term -> Term -> Subst -> Maybe Subst
match (Apply f argsLeft) (Apply g argsRight) sigma
  | f == g      = let matchArg s (l,r) = match l r s in
                  foldM matchArg sigma (zip argsLeft argsRight)
  | otherwise   = fail "different function symbols"
match x@(Var a) b sigma
  | applySubst sigma x == x     = Just (addSubst sigma x b)
  | applySubst sigma x /= b     = fail "inconsistent binding"
  | otherwise                   = Just sigma
match _ _ _     = fail "incompatible"
```

Figure 3: Matching Algorithm

3.2 Reduction to Conjunctive Normal Form

Many decision procedures require the problem to be expressed in *Conjunctive Normal Form*, that is, as a list of (universally quantified) clauses. However, as the input cannot be expected to always be in this form, we have to transform it since both SMT-solvers and superposition provers work on clauses.

The "naïve" reduction of a formula to CNF is based on skolemization, miniscoping (pushing quantifiers as deep in the terms as possible), recursive application of the De Morgan rules, and distributivity of $\land$ over $\lor$. Unfortunately, it is exponential in the worst case, and in many practical problems this behavior shows up. We therefore implemented a Tseitin-like transformation heuristically used for some formulæ, inspired from the chapter *Computing Small Clause Normal Forms* of the Handbook of Automated Reasoning [NW01] to convert arbitrary first order formulæ to CNF. This transformation only preserves satisfiability, i.e., the set of clauses obtained by this technique is not logically equivalent to the input first-order formula if renaming is used, but they are equisatisfiable (the formula has a model if and only if the set of clauses has one).

The basic idea of the transformation is to give "names" for subformulæ. If, for example, one has to convert $(a \Leftrightarrow b) \Rightarrow (c \land d)$ into clauses, one can create a new term $\eta \equiv (a \Leftrightarrow b)$, and use the new term in the formula: $\eta \Rightarrow (c \Leftrightarrow d)$. The CNF is then $(\neg\eta \lor c) \land (\neg\eta \lor d)$ with some additional clauses (to define η): $(\neg\eta \lor \neg a \lor b) \land (\neg\eta \lor \neg b \lor a)$.

The main issue was the representation that YICES uses for formulæ: there are no explicit $\land$, $\neg$ nor $\exists$ terms, because all terms are signed, and $a \land b$ is defined as $\neg((\neg a) \vee (\neg b))$, $\exists x.a$ is written $\neg(\forall x.\neg a)$. *Signed terms* are terms annotated with a *polarity*, + or − (the machine representation is an integer whose least significant bit indicates the sign) so that $+p$ is the same term as p, and $-p$ is the same as $\neg p$. To negate a term, it is sufficient to flip its polarity. Furthermore, the $\vee$ operator is not binary but n-ary, which makes miniscoping and reduction to CNF more complicated.

This design forced us to do skolemization during reduction to CNF, not before, because there is simply no way to do some operations like reduction to *negation normal form*. We present the resulting clausification algorithm as a set of rewriting rules in Figure 4. The *cnf* function takes a term and its polarity, and returns a list of lists of literals, representing a conjunction of clauses, to get around the representation of terms in YICES. The operator $l \oplus r$ denotes the concatenation of the lists l and r, and $x : l$ is the "cons" operator (as in Haskell or LISP) that creates a list with the element x as the head and the list l as the tail.

$$\begin{array}{rcl}
cnf(\neg a,+) & \leadsto & cnf(a,-) \\
cnf(\neg a,-) & \leadsto & cnf(a,+) \\
cnf(a \vee b,+) & \leadsto & cnf(a,+) \bowtie cnf(b,+) \\
cnf(a \vee b,-) & \leadsto & cnf(\neg a,+) \oplus cnf(\neg b,+) \\
cnf(a \Leftrightarrow b,+) & \leadsto & cnf((a \Rightarrow b) \wedge (b \Rightarrow a),+) \\
cnf(a \Leftrightarrow b,-) & \leadsto & cnf((a \wedge b) \vee (\neg a \wedge \neg b),-) \\
cnf(\forall x.a,+) & \leadsto & cnf(a,+) \\
cnf(\forall x.a,-) & \leadsto & cnf([c(\overline{y})/x]a),-) \text{ where } c(\overline{y}) \text{ is new (skolemization)} \\
 & & \text{and } \overline{y} = vars(a) \setminus \{x\} \\
cnf(a,\pm) & \leadsto & [[\pm a]] \text{ if } a \text{ atomic} \\
[] \bowtie r & \leadsto & [] \\
l \bowtie [] & \leadsto & [] \\
(l : l') \bowtie (r : r') & \leadsto & [l \oplus r] \oplus ([l] \bowtie r') \oplus ([r] \bowtie l') \oplus (l' \bowtie r')
\end{array}$$

Figure 4: Clausification Algorithm

Special terms and Renaming

The SMTLIB format [RLT06] defines an *if-then-else* construct: a term $ite(c, t_1, t_2)$ is t_1 when c is true, and t_2 when c is false. In YICES we do not deal with this kind of terms inside the solver (nor in the superposition solver), so we have to remove them during the clausification process. To do so, we introduce a new name $\eta(\overline{x})$ for $t = ite(c, t_1, t_2)$ (with $\overline{x}$ the set of free variables of t), replace t by

$\eta(\overline{x})$ everywhere, and add two clauses $\neg c \vee \eta(\overline{x}) \simeq t_1$ and $c \vee \eta(\overline{x}) \simeq t_2$. Also, the (n-ary) *xor* terms are re-encoded into nested equivalences, i.e., $a \otimes b \otimes c \equiv (a \Leftrightarrow \neg b) \Leftrightarrow \neg c$.

Note that the additional clauses for renaming, the ones which *define* the new name, are not returned by the *cnf* function, but directly appended to the set of clauses of the problem. The reason for this is that the clausal definition is not contextual, so for instance in $cnf(a \vee b, +)$ where η is a name for b, we do not want the clauses for the definition of η to be part of $cnf(b, +)$, for they would be joined with clauses from $cnf(a, +)$. Here are the additional rules:

$$\begin{array}{rcl}
cnf(ite(c, t_1, t_2), \pm) & \rightsquigarrow & [[\pm\eta]] \text{ with } \mathbf{def}(\eta) = (\neg c \vee \eta \simeq t_1) \wedge (c \vee \eta \simeq t_2) \\
cnf(a \otimes b, \pm) & \rightsquigarrow & cnf((\neg a) \Leftrightarrow b, \pm) \\
cnf(a \Leftrightarrow b, \pm) & \rightsquigarrow & cnf(a \Leftrightarrow \eta, \pm) \text{ with } \mathbf{def}(\eta) = cnf(\eta \Leftrightarrow b, +) \\
cnf(a \vee b, +) & \rightsquigarrow & cnf(a \vee \eta, +) \text{ with } \mathbf{def}(\eta) = cnf(\eta \Rightarrow b, +)
\end{array}$$

The last rules are renaming rules, they are applied heuristically when the regular reduction to CNF is likely to produce too many clauses. The $\Leftrightarrow$ case is triggered if both subterms are non-atomic formulæ, and the $\vee$ case is triggered if the clauses for b are too numerous (we can re-use the clauses for b to define η, by appending $\neg\eta$ to all clauses, thanks to the associativity of the disjunction). As we will see in Section 6, SPASS using our clausifier has almost the same performance as SPASS using FLOTTER (the sophisticated clausifier built in SPASS) on the TPTP benchmarks, although it occasionally blows up.

Our first implementation was doing a lot of renaming, which sometimes hampered the ability of the prover to actually solve the problem (it introduced a lot of new symbols), let alone the fact that the code was overly complicated and hard to understand. We later rewrote the clausification component with the algorithm described in Figure 4).

4 Implementation of a Superposition Prover

The rationale for using a calculus based on resolution and paramodulation comes from the popularity of those for general first-order problems. The most competitive provers are based on some variant of superposition. Another very interesting property is that the calculus also works with clauses (which makes communication between the ground DPLL solver and the first-order prover easier). State-of-the-art DPLL solvers conceptually perform a kind of *lazy resolution*, driven by conflicts.

A superposition solver embedded in YICES was thus expected to be used for several purposes:

1. Find inconsistencies among non-ground axioms,
2. Generate new non-ground clauses to be used by any instantiation mechanism (such as heuristic E-graph matching),
3. Solve problems that have few or no ground clauses.

4.1 The Superposition Calculus

Decision procedures for first-order logic have been the subject of extensive research for decades. The most powerful inference system is resolution [Rob] and the more sophisticated calculi that followed. In the context of embedding a first-order prover inside YICES, we needed a simple yet powerful calculus, able to deal with equality (ignoring other theories such as arithmetic). We took inspiration from the description of the E prover [Sch02], a state-of-the-art purely equational prover.

E is based on a superposition calculus with four inference rules, and more than a dozen simplification rules. Its main strength resides in the flexibility of its heuristics, and its main loop differs from the more conventional main loop of Otter [McC95] by the fact that non-processed clauses are totally ignored until they are selected. Hence, each iteration of the loop is fast – a valuable property for the integration in another decision procedure, since the main loops are interleaved. We implemented all the inference rules (see Figure 5), but, because of time constraints, we implemented only a subset (Figure 6) of the simplification rules described in [Sch02]. The single line in the inference rules denotes a *generating* rule, i.e., the premisses (clauses above the line) are preserved; the double line denotes a *destructive* rule, i.e., clauses above are replaced by clauses below.

The inferences and simplifications depend on some constraints on terms and literals to diminish the amount of redundant inferences. Those restrictions are based on a *term ordering* and on a notion of *selected literals* that will be explained later (cf. 4.2).

The two superposition rules from Figure 5 can be thought of as *conditional rewriting*, because a positive equation of a clause C is used to rewrite a literal

Equality Resolution:

$$\frac{s \not\simeq t \lor R}{\sigma(R)}$$

$\sigma = mgu(s,t)$, $\sigma(s \simeq t)$ eligible for resolution

Superposition into Positive literals:

$$\frac{s \simeq t \lor S \qquad u \simeq v \lor R}{\sigma(u[p \leftarrow t] \simeq v \lor S \lor R)}$$

$\sigma = mgu(s, u|_p)$, $\sigma(u) \not< \sigma(v)$, $\sigma(s \simeq t)$ eligible for paramodulation, $\sigma(u \simeq v)$ eligible for resolution, $u|_p$ is not a variable.

Superposition into Negative literals:

$$\frac{s \simeq t \lor S \qquad u \not\simeq v \lor R}{\sigma(u[p \leftarrow t] \not\simeq v \lor S \lor R)}$$

$\sigma = mgu(s, u|_p)$, $\sigma(s) \not< \sigma(t)$, $\sigma(u) \not< \sigma(v)$, $\sigma(s \simeq t)$ eligible for paramodulation, $\sigma(u \not\simeq v)$ eligible for resolution, $u|_p$ is not a variable.

Equality Factoring:

$$\frac{s \simeq t \lor u \simeq v \lor R}{\sigma(t \not\simeq v \lor u \simeq v \lor R)}$$

$\sigma = mgu(s,t)$, $\sigma(s) \not< \sigma(t)$, $\sigma(s \simeq t)$ eligible for paramodulation.

Figure 5: Inference rules of the Superposition Calculus

Destructive Equality Resolution: $\begin{array}{c} x \not\simeq t \lor R \\ \hline\hline \sigma(R) \end{array}$ if $\sigma = mgu(x,t)$, x is a variable

Syntactic Tautology Deletion 1: $\begin{array}{c} s \simeq s \lor R \\ \hline\hline \end{array}$

Syntactic Tautology Deletion 2: $\begin{array}{c} s \simeq t \lor s \not\simeq t \lor R \\ \hline\hline \end{array}$

Deletion of Duplicated Literals: $\begin{array}{c} s \mathrel{\dot{\simeq}} t \lor s \mathrel{\dot{\simeq}} t \lor R \\ \hline\hline s \mathrel{\dot{\simeq}} t \lor R \end{array}$

Deletion of Resolved Literals: $\begin{array}{c} s \not\simeq s \lor R \\ \hline\hline R \end{array}$

Rewriting of Positive Literals: $\begin{array}{c} s \simeq t \qquad u \simeq v \lor R \\ \hline\hline s \simeq t \qquad u[p \leftarrow \sigma(t)] \simeq v \lor R \end{array}$ if $u|_p = \sigma(s)$

where $\sigma(s) > \sigma(t)$, $u \simeq v$ not eligible for resolution or $u \not> v$ or $p \neq \Lambda$ or σ is not a renaming.

Rewriting of Negative Literals: $\begin{array}{c} s \simeq t \qquad u \not\simeq v \lor R \\ \hline\hline s \simeq t \quad u[p \leftarrow \sigma(t)] \not\simeq v \lor R \end{array}$ if $u|_p = \sigma(s)$, $\sigma(s) > \sigma(t)$

Clause Subsumption: $\begin{array}{c} T \qquad R \lor S \\ \hline\hline T \end{array}$ if $\sigma(T) = R$ for some σ

Figure 6: Simplification rules of the Superposition Calculus

of another clause D assuming the other literals of C are falsified (which is why those other literals are kept as conditions in the generated clause). Superposition, together with contraction rules, subsumes the classical resolution rule:

$$\dfrac{\dfrac{\dfrac{A \lor C \qquad \neg B \lor D}{A \simeq \top \lor C \qquad B \not\simeq \top \lor D}\ \text{equational literals}}{\sigma(\top) \not\simeq \sigma(\top) \lor \sigma(C) \lor \sigma(D)}\ \text{superposition with } \sigma(A) = \sigma(B)}{\sigma(C) \lor \sigma(D)}\ \text{deletion of resolved literals}$$

4.2 Term Ordering and Literal Selection

The inferences of the E prover produce a high number of redundant inferences, i.e., inferences that we could avoid performing without losing completeness. To reduce the search space, two kinds of constraints are used:

Ordering Constraint: By defining a (partial) order on terms, we can prevent inferences that will, intuitively, go “against” the ordering – that is to say, that produce bigger terms or literals.

Literal Selection Functions: By choosing some literals in the clause, we prevent the other literals from participating in any inference involving that clause. Those non-selected literals can be used only after we “get rid of” the selected literals.

Term Ordering

We follow the definitions of [NR99]: an *ordering* $\succ$ is a transitive irreflexive binary relation. A *reduction ordering* is an ordering with the following properties:

- (stable under substitution) $s \succ t \Rightarrow \sigma(s) \succ \sigma(t)$
- (monotonic) $s \succ t \Rightarrow u[p \leftarrow s] \succ u[p \leftarrow t]$
- (well founded) no infinite sequence $(t_i)_{i \in \mathbb{N}}$ with $\forall i. t_i \succ t_{i+1}$ exists.

We need such a total ordering on ground terms, but do not require it to be total in the general case. The ordering implemented in Yices is the Knuth-Bendix Ordering (KBO).

Given a *precedence* $\succ$ on the function symbols of some signature (F, S) (a total order on symbols), and a *weight function*[5] $w : F \to \mathbb{N}^*$ such that $w(c) > w_0$ for some w_0 and all constants $c \in F$, we define the weight $w(t)$ of any term t as follows:

$$w(t) = \begin{cases} w_0 & \text{if } t \in X, \\ w(f) + \sum_{i=1}^{n} w(t_i) & \text{if } t = f(t_1, \ldots, t_n). \end{cases}$$

Then KBO is defined as $s >_{kbo} t$ if any variable of X occurs at least as many times in s as in t, and either

[5] we added some assumptions to make the implementation simpler, e.g., the precedence is total and no symbol has a null weight

- $w(s) > w(t)$
- $w(s) = w(t)$ and one the following conditions holds:
  1. $t \in X$ and $s = \underbrace{f(f(\ldots f(t)))}_{n \text{ times}}$, $n \geq 1$
  2. $s = f(s_1, \ldots, s_n)$, $t = f(t_1, \ldots, t_n)$, and there exists $1 \leq i \leq n$ with $s_j = t_j$ for $j < i$ and $s_j >_{kbo} t_j$ (lexicographic path ordering on subterms)
  3. $s = f(s_1, \ldots, s_n)$, $t = g(t_1, \ldots, t_m)$, and $f \succ g$.

The intuition is that a term s can only be bigger than another term t if the multiset of its variables supersedes the multiset of the variables of t, because otherwise we could replace some variable with a very big term that would break the property that $>_{kbo}$ is stable under substitution. For instance, $f(x, a)$ cannot be bigger than $f(x, x)$, even though its weight is higher, because using $\sigma = [b/x]$ with $b \succ a$ would make $f(b, a) \prec f(b, b)$.

The weight function can be cached with the terms (we also use it to compute a heuristic weight for clauses in other parts of the prover). Once the variable occurrences criterion is established, comparing weight can be very fast if weights are cached. The lexicographic ordering on subterms is only used when all the other criteria fail.

Our implementation of KBO is relatively naïve (we did not implement the more sophisticated variant in [Löc06]) but still performs relatively well in practice. We also have a comparison cache, implemented as a hashmap, only used to save the results of comparing two non-constant terms (i.e., function applications). We found that the implementation of a correct term ordering is a delicate operation, for it can cause a loss of completeness (for instance, if the property that variables have a lower weight than any constant is not enforced) that is difficult to track and fix.

The ordering can be extended to literals (equations) using a *multiset extension* of $>_{kbo}$. The multiset extension, described in [NR99], is hard-coded for the only kinds of multisets we use. Equations are regarded as multisets in the following way:

$$\begin{aligned} s \simeq t &\quad\rightsquigarrow\quad \{\{s\}, \{t\}\} \\ s \not\simeq t &\quad\rightsquigarrow\quad \{s, t\} \end{aligned}$$

This ordering is also used for literal selection, where it allows us to define a partial ordering on the literals of a clause — several literals can be minimal or maximal in the same clause since the ordering is not total: l *maximal* means that there is no l' with $l' >_{kbo} l$.

Literal Selection

Literal selection functions are used to reduce the number of inferences performed by the system, while retaining completeness. They are defined as follow, assuming σ is a substitution and $l \vee R$ is a clause (l being a literal of the clause):

Selected literals: A literal selection function *sel* is a function that associates, to a clause C, a possibly empty set of literals $sel(C) \subseteq C$. The literals in $sel(C)$ are called *selected literals*. The following property is necessary:

$$sel(C) \cap C^- = \emptyset \Rightarrow sel(C) = \emptyset$$

where C^+ is the set of positive literals of C, and C^- is the set of negative literals of C.

Eligible for resolution: $\sigma(l)$ is eligible for resolution if either

- $sel(C) = \emptyset$ and $\sigma(l)$ maximal in $\sigma(C)$.
- $sel(C) \neq \emptyset$ and l negative and $\sigma(l)$ maximal in $\sigma(sel(C) \cap C^-)$.
- $sel(C) \neq \emptyset$ and l positive and $\sigma(l)$ maximal in $\sigma(sel(C) \cap C^+)$.

In particular, if some literal is selected, then only selected literals have a chance to be eligible for resolution.

Eligible for paramodulation: $\sigma(l)$ is eligible for paramodulation if $l \in C^+$, $sel(C) = \emptyset$ and $\sigma(l)$ maximal in $\sigma(C)$.

We can note that several literals can be eligible for resolution (or eligible for paramodulation) at the same time, since the ordering $>$ on terms, being only partial, allows several literals to be maximal. The intuitive meaning of those eligibility constraints is the following: $\sigma(l)$ is *eligible for resolution* if it has been chosen (by the ordering or by the selection function) for being rewritten; it is *eligible for paramodulation* if it has been chosen as an active (potentially conditional) rewriting rule.

4.3 Theory-specific Rules

As part of the integration of the superposition prover with YICES, we also added a few specialized simplification rules, shown in Figure 7. The AC (Associative-Commutative) rules are directly inspired from the description of E, and the addition of the *distinct* rules was motivated by its appearance in several SMTLIB problems. We found that this idea was also described in [SB05]. Here is a short description of those theories:

AC: A symbol f can be declared *Associative Commutative*, or AC for short, if it satisfies the following axioms:

Commutativity:	$f(x,y)$	$\simeq$	$f(y,x)$
Associativity:	$f(x,f(y,z))$	$\simeq$	$f(f(x,y),z)$

We therefore add those axioms for every symbol we know to be AC (e.g., +) in addition to the dedicated simplification rules. It would be even better to detect the presence of AC axioms in the input problem, or during the proof search. We have implemented an equality modulo AC check function for the ACS and ACD rules; it uses a hash function that is invariant modulo AC to rule out most equality checks (if two terms do not have the same hash, they cannot be equal modulo AC).

Distinct: Many SMTLIB problems use the *distinct* construct to specify that some symbols are pairwise distinct. Although this can be expressed using literals like $a \not\simeq b$, the number of such literals grows quadratically w.r.t. the number of distinct objects. Hence, we add a specific reasoning system in the superposition prover to handle this constraint. $s \neq_{distinct} t$ is to be read as "*distinct*$(s, t, \dots)$ occurs as a unit positive clause in the input problem"

AC Simplification (ACS): $$\frac{s \not\simeq t \vee R}{R}\; s =_{AC} t$$

AC Tautology Deletion (ACD): $$\frac{s \simeq t \vee R}{}\; s =_{AC} t$$

Distinct Simplification (DS): $$\frac{s \simeq t \vee R}{R}\; s \neq_{distinct} t$$

Distinct Tautology Deletion (DD): $$\frac{s \not\simeq t \vee R}{}\; s \neq_{distinct} t$$

Figure 7: Theory-specific Simplification Rules

We also add some axioms for the theory of total orderings and for the theory of arithmetic if the corresponding symbols are detected in the input problem.

Total Orderings: if the symbol $\geq$ is used in some non-ground clause, we add the following axioms to the superposition prover:

Reflexivity: $x \geq x$

Antisymmetry: $x \geq y \wedge y \geq x \Rightarrow x \simeq y$

We do not include transitivity, because it seems to produce a high number of clauses that are useless for the proof, hence interfering with the search for a proof.

Arithmetic Simplifications: Although we cannot hope to solve complex arithmetic problems in the superposition prover, we can add a few simplification rules (purely syntactically, using positive unit clauses) when we detect arithmetic symbols in the input problem:

$$\begin{array}{rcl} x + 0 & \simeq & x \\ x * 0 & \simeq & 0 \\ x * 1 & \simeq & 1 \end{array}$$

Mixing arithmetic and ordering: We can add the axiom $1 \geq 0$ in this case. We can also include the monotonicity of $\lambda x.x + c$ w.r.t. $\geq$:

$$a \geq b \Rightarrow a + c \geq b + c$$

Arrays: In the SMTLIB logics that include arrays, we add the following axioms for every type-specialized couple of *select* and *update* symbols:

$$\begin{array}{rcl} & & \mathit{select}(\mathit{update}(\mathrm{a}, \mathrm{i}, \mathrm{v}), \mathrm{i}) \simeq \mathrm{v} \\ \mathrm{i} \neq \mathrm{j} & \Rightarrow & \mathit{select}(\mathit{update}(\mathrm{a}, \mathrm{i}, \mathrm{v}), \mathrm{j}) \simeq \mathit{select}(\mathrm{a}, \mathrm{j}) \end{array}$$

Adding some theory-specific knowledge to the superposition prover can be useful, as it may allow it to find a refutation or just to make more interesting inferences, but the drawback is that many useless clauses can be produced – eating memory and slowing down the search. This phenomenon was most stressed when the transitivity axiom for $\geq$ was enabled.

4.4 Implementation Highlights

Representation of Terms

We re-used the representation of terms of YICES. Terms are perfectly shared (a technique sometimes called *hash-consing*) in a *term table*. The type `term_t` is an alias for `int32_t`; the least significant bit is used to encode the sign of a term, and the other 31 bits are the offset of the term descriptor in the term table.

The representation of terms as integers has some advantages; for instance, it provides a built-in (although arbitrary) total ordering on terms, which is often useful – e.g., most balanced trees need an ordering, and we can normalize vectors of terms by sorting them, etc.

Representation of Clauses

Unlike terms, the representation of clauses in YICES was not suitable for a first-order prover. We therefore built, in a similar fashion to terms, a *clause table* containing clauses descriptors. Clauses are also hash-consed, but a subtlety is that we may need several representations of a clause, so `fo_clause_repr_t` is an alias for a pointer.

Unification and term indexes require terms to have disjoint sets of free variables. To enforce this property, for each clause, there is a *canonical* representation – the one in the clause table – and possibly several *non-canonical* representations that have a shorter lifespan. The former and the latter use two disjoint sets of variables (respectively called *canonical variables* and *non-canonical variables*). When a clause becomes the *given clause*, a copy of the clause is created with non-canonical variables. Before inferences and simplifications are performed, the canonical clause is added to the set of active clauses (and therefore to various term indexes); this way, the given clause can be both a passive and an active partner of inferences and simplifications, with two disjoint sets of variables.

Inference and Simplification Rules

The inferences rules defined in Figure 5 are declarative, so we need to find an efficient implementation in the C language used to write YICES. Figures 9 and 10 sketch such an imperative implementation, in Python. The second figure is more interesting because less straightforward: we did not implement separate functions for the two superposition rules — which differ in the sign of the rewritten equation — but instead we wrote a function for the cases where the given clause is rewriting another clause, and a function for the cases where the given clause is being rewritten. The implementation of the superposition inference rules contains several functions:

superposition_active: Generates all the inferences where the given clause is used as a conditional rewrite rule. We thus have to iterate through positive equations of the clause, trying to rewrite any other subterm at any position in any other active clause using `generate_for_term`.

superposition_passive: Generates all the inferences where the given clause is conditionally rewritten by another clause. We iterate through all literals, both positive and negative, and traverse their subterms to try to conditionally rewrite them using any other active clause (using `generate_for_subterms`).

superpose: This function is used by the two previous functions to generate a new clause. In the C implementation, it receives one argument, of type `superposition_pair_t` (cf Figure 8 to see the definition of the structure).

generate_for_subterms: Given an equation $u \doteq v$, this function traverses all subterms of u and unifies them with indexed terms (that occur in some equation). Then it uses the *term occurrences index* (described in Section 4.5) to retrieve the equations and clauses in which the unifiable terms occur, and calls `superpose`.

generate_for_term: Given an equation $s \simeq t$, this function iterates through all the subterms that unify with s in any indexed clause. Then it uses the *term occurrences index* (described in Section 4.5) to retrieve all the reverse paths from the unifiable subterms to any equations in any indexed clauses, and calls `superpose`.

Overall, implementing a superposition solver is a relatively subtle task, for the data structures and algorithms are complex. Testing the behavior of single components (with some basic unit testing, for instance testing the correct

```
typedef struct {
  term_t passive; // passive (rewritten) term
  ilist2_t *path;  // path to the literal which contains the term
  fo_clause_repr_t *passive_clause; // rewritten clause
  int32_t passive_idx; // index of the literal in the passive clause

  term_t active; // active (rewriting) term
  fo_clause_repr_t *active_clause; // rewriting clause
  int32_t active_idx; // index of active term/literal in active_clause
} superposition_pair_t;
```

Figure 8: The `superposition_pair_t` structure. `ilist2_t` is a linked list of pairs of integers, `fo_clause_repr_t` is the type of a first-order clause.

behavior of lists, or unification functions) and the behavior of the whole prover proved invaluable to detect bugs as early as possible. We used the Pelletier problems [Pel86] as a testbed. To do so, we first translated most of them in the TPTP format. A simple shell script was then used to test both the clausification algorithm — by forwarding the clauses to SPASS [WSH$^+$07] — and the prover itself. Producing proof certificates (cf Section 4.7) also proved very useful to get some trust in the correctness of the prover.

4.5 Indexing Structures

Both the inference rules and the simplification rules involve the retrieval of terms and substitutions that have some relation with a given term. For instance, the rewriting rules (which are a crucial element of the calculus in practice) require to find all terms s, member of some positive equation, that are a generalization of a given t (i.e., such that $\sigma(s) = t$ for some substitution σ). On real problems, iterating over all members of positive equations and trying to match them against t becomes far too slow when the number of equations and terms grows.

All the state-of-the-art provers for first-order logic therefore implement some kind of term indexing structures, which allow for sub-linear retrieval of terms that satisfy some property w.r.t.some term. Usual queries for a term u are:

generalizations: retrieve all v such that $\exists\sigma.\sigma(v) = u$,

unifications: retrieve all v such that $\exists\sigma.\sigma(u) = \sigma(v)$,

instances: retrieve all v such that $\exists\sigma.\sigma(u) = v$,

renamings: retrieve all v such that $\exists\sigma.\sigma(u) = \sigma(v)$ and σ is a variable renaming.

A term index is called *perfect* if the set of terms it returns for a query is the exact set of terms that satisfy the query; the index is called *non-perfect* otherwise (in which case it returns a superset, i.e., it only filters the terms).

```
def equality_resolution(clause):
  for lit in clause.negative_lits:
    s, t= lit
    elif not is_variable(s) and not is_variable(t): continue
    elif not unifiable(s, t): continue
    elif not eligible_for_resolution(lit): continue
    else:
      sigma = unify(s, t)
      add sigma(clause - lit) to unprocessed

def equality_factoring(clause):
  for lit in clause.positives_lits:
    if not is_equation(lit): continue
    for l in clause.positives_lits if l != lit:
      if not is_equation(l): continue
      if compatible(lit, l):  # types, ordering...
        u, v = lit
        s, t = l
        sigma = unify(s, u)
        add sigma(clause - l + (t != v)) to unprocessed
```

Figure 9: Implementation of the “simple” inference rules

We defined a generic interface for such indexing structures based on iterators, with a “naïve” default implementation based on *root symbol indexing*. This implementation just relies on the first symbol of indexed terms to filter them, i.e., for each function symbol f, the index associates the list of terms of the form $f(t_1,\dots,t_n)$ (using a hash table). This structure allows a very simple implementation of all query types.

After a study of the state-of-the-art of term indexing, based on the Handbook of Automated Reasoning [RSV01] we then implemented two indexing structures: *Substitution Trees* [GS94] and *Perfect Discrimination Trees* [McC90]. None of our implementations supports querying for *renamings*, because it was of no use to us. The implementations provide functions to print trees in the *dot*[6] format, for debugging purpose.

4.5.1 Substitution Trees

We implemented the general-purpose term indexing structure named *substitution trees*, from the original paper [GS94]. Those trees are based on the concept of *most specific common generalization*. Each node of the tree contains a substitution that must be matched or unified against the query substitution – hence the name of the structure. Figure 11 provides an example of such a tree.

The substitution tree operates only on substitutions. To perform a retrieval operation on a term t, one actually performs some operation on the substitu-

[6] `http://graphviz/org`

```
def superposition_passive(clause):
  for lit in clause.literals:
    if has_selected_literals(clause) and not is_selected(lit): continue
    u, v = lit
    if u > v:
      generate_for_subterms(u, c)
    elif v > u:
      generate_for_subterms(v, c)
    else:
      generate_for_subterms(u, c)
      generate_for_subterms(v, c)

def superposition_active(clause):
  for lit in clause.positive_lits:
    if has_selected_literals(clause) and not is_selected(lit): continue
    s, t = lit
    if s > t:
      generate_for_term(s, c)
    elif t > s:
      generate_for_term(t, c)
    else:
      generate_for_term(s, c)
      generate_for_term(t, c)

def generate_for_subterms(u, c):
  for p in positions(u):
    u_p = u at position p
    for s in unifiable(u_p):  # term index
      sigma = unify(s, u_p)
      for d in superclauses(s):  # occurrences index
        for t such that s=t in d:
          pair = Pair(u, p, c, s, d)
          superpose(pair, sigma)

def generate_for_term(s, c):
  for u_p in unifiable(s):  # term index
    sigma = unify(s, u_p)
    for u, p such that u at position p == u_p:  # occurrences index
      for d in superclauses(u):  # occurrences index
        for v such that u=v in d:
          pair = Pair(u, p, d, s, c)
          superpose(pair, sigma)

def superpose(pair, sigma):
  # perform all the constraint checks, return if any fails
  # generate the clause, entirely determined by pair and sigma
```

Figure 10: Implementation of the superposition inference rules

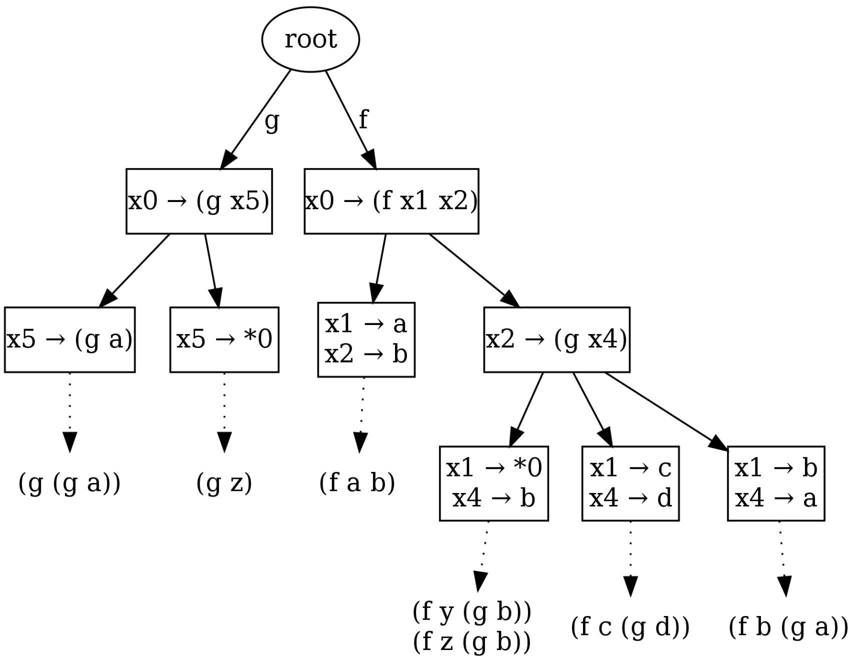

Figure 11: Example of Substitution Tree

tion $\{x_0 \to t\}$, where x_0 is a fixed variable (in practice, there is one such variable for every type). Substitutions in the tree operate on two disjoint sets of variables: the variables x_i (x_0, for instance) which are used to represent some pattern common to one or more indexed terms, and the starred variables $*i$ that represent actual variables in the indexed terms. An indexed term, before being inserted, is converted into a canonical form where it only contains starred variables, so $f(x, a, g(x, y))$ will become $f(*0, a, g(*0, *1))$. The insertion operations are quite complicated (see the reference paper) as they require merging and splitting of substitutions. The retrieval is based on the matching or unification (depending on the query) of substitutions. As we traverse the tree, we carry a substitution. For every encountered node, if we can unify (or match) our substitution with the substitution it contains, we explore its child nodes with the new bindings from the unification (or matching). Leaves of the tree contain a list of index terms that are variant of one another.

We found that the implementation of this indexing structure is quite subtle and error-prone. The insertion algorithm in particular involves non trivial operations on substitutions (computing the most specific common generalization, for instance). We believe the representation of terms in YICES does not favor performance for this implementation – using self-referring pointers to represent variables allows for a much faster (backtrackable) binding of variables during the traversal of the tree. Since the implementation is quite complicated, when a query is issued we compute all answers that are then stored in a linked list; the iterator used by the caller then just iterates over the list.

4.5.2 Perfect Discrimination Trees

Perfect Discrimination Trees are the most widespread indexing structure in first-order automated theorem provers. They are mainly used to retrieve generalizations of a query term (used for forward subsumption and rewriting).

We decided to implement this data structure after we noticed that the rewriting simplification rules (cf. Figure 6) were an important bottleneck (the prover could spend up to 70% of its time matching terms against unit equations). The discrimination trees are quite straightforward to implement if the only supported operation is the retrieval of generalizations.

An indexed term is transformed into a *flat term*, i.e., the string of symbols obtained by prefix traversal of the term DAG. For instance, the *flat term* corresponding to $f(g(x,y),a,h(b))$ is $f.g.x.y.a.h.b.\epsilon$ where ϵ marks the end of the string.

The flat term is then inserted in a prefix tree, or *trie*. Upon retrieval, the query term is also translated into a flat term and a backtracking search in the trie is carried on. The basic Discrimination Trees use a universal variable $*$ to represent variables of the indexed terms, and are thus non perfect if the indexed terms are not linear. To make a Discrimination Tree perfect, we use the proper variables of indexed terms (which decreases sharing in the trie) and check the consistency of the bindings. An example of discrimination tree is shown in Figure 12.

Insertion and removal are very simple operations since they only require to transform the term into its flat representation (which requires a prefix order traversal of the DAG structure of the term) and then a classical trie insertion or removal of the flat term considered as a string of symbols. We only implement the generalizations retrieval, using a backtracking stack of nodes to explore.

4.5.3 Term Occurrences Index

Some inference rules and simplification rules require to match a literal or a subterm of a literal from the current clause with terms occurring in other clauses (clauses that have been processed, more precisely). However, even once a suitable term is found, using one of the previously described term index, we still have to know in which term or clause it occurs. This is what the *term occurrences index* is for.

The queries this index is designed to answer are queries such as “whose indexed terms u is the given term t a subterm”, “what are the positions p such that $u|_p = t$”, and “in which clauses is t a literal (i.e., appears as left-hand side or right-hand side of some equation of the clause)”. In the positive and negative superposition rules, notably, the first query is used to retrieve the subterms that can be conditionally rewritten.

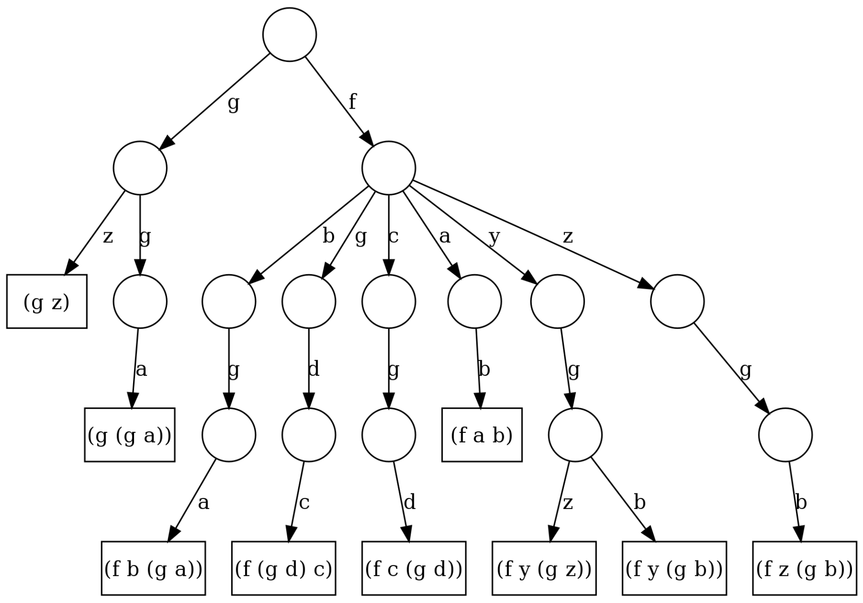

Figure 12: Example of Perfect Discrimination Tree

The *term occurrences index* is in reality a pair of indexes:

1. One which associates to a term the set of clauses in which the term occurs in some equation.
2. One which associates to terms all their occurrences in other terms. This index keeps a set of *reverse paths* from terms to any literal in which the term occurs as a subterm (i.e., if t is in this index, all paths p from some literal u — that occurs in some equation — to $t = u|_p$ are associated to t, but in a reverse order).

A *path* is defined as a non empty list of pairs $(t_1, p_1), (t_2, p_2), \dots, (t_n, p_n)$ with $p_1 = \bot$, such that t_{i+1} is the p_{i+1}^{th} subterm of t_i. We say that the path *leads to t* if $t = t_n$ is the term in the last pair. A *reverse path* is the corresponding reverse list. **Example**: If $t = f(a, b)$ and two literals $u = p(g(f(a, b)))$ and $v = q(f(a, b), g(f(a, b)))$ are in the *term occurrences index*, then in this index t will be associated the reverse paths $(g(f(a, b)), 1)p.1$, $q.1$, and $g.1.q.2$ (Figure 13).

4.6 Subsumption Check Algorithm

Although a lot of work has been done on ways to diminish the amount of subsumption checks to perform in clause-based provers, the implementation of the subsumption check itself is, to our knowledge, not often described. Therefore, we implemented our own algorithm.

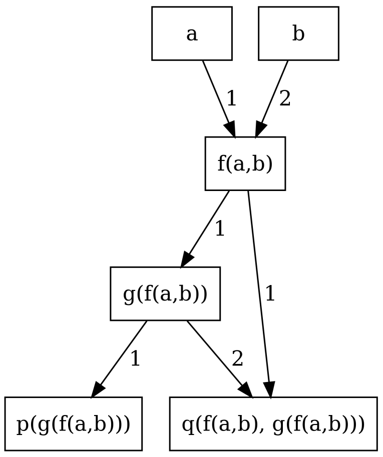

Figure 13: Example of Reverse Path Index

The subsumption algorithm is used to detect whether two clauses C and D satisfy the *subsumption property*, that is, whether there exists some substitution σ such that $\sigma(C) \subseteq D$ (in which case C is said to *subsume* D). The $\subseteq$ relation is the multiset inclusion property, which means that $p(X) \vee p(Y)$ does not subsume $p(a)$, even though $p(X)$ subsumes $p(a)$. The multiplicity of literals counts. Intuitively, if C subsumes D it means that all instances of D are implied by instances of C, that is, D provides only information redundant w.r.t. C – which allows the prover to get rid of D without missing any inference.

Our approach is to generate literal-to-literal subsuming substitutions — that is, for each pair of literals $l \in C, l' \in D$, if $\sigma(l) = l'$ for some σ, we add σ to a set s_l. Then, we try to merge substitutions [7]for each literal of C. If a substitution "compatible" with every literal of C is found, then C subsumes D. More formally, a substitution σ is compatible with C if $\forall l \in C.\exists \theta \in s_l.\theta \subseteq \sigma$. We can restrict ourselves to compatible substitutions that bind only variables present in C, because other variables have no influence over the subsumption property. Some pseudocode description of the algorithm is shown in Figure 14.

So far, the subsumption deletion rule did not prove to be a bottleneck (unlike rewriting rules), so we did not use the *feature vector indexing* [Sch04] to filter out subsumptions checks. However, that may be a useful extension of the prover.

4.7 Proof certificates and Proof checking

In order to get more confidence in the answers of the prover, we developed a simple file format to describe unsatisfiability proofs. A certificate for a proof is a list of axioms and a list of inferences, one of which must have $\Box$ as a conclusion.

[7] the reader may notice that this is very similar to the Terminator algorithm presented in Section 5.2

```
def subsumes(c, d):
  # compute the set of substitutions for each literal of c
  s = {}
  for lc in c:
    s[lc] = emptyset
    for ld in d:
      sigma = match(lc, ld)
      if sigma:
        s[lc].add(sigma)
  # try to find a compatible substitution
  return len(compatible_substs(emptysubst, set(s))) > 0

def compatible_substs(sigma, sets):
  if sets is empty:
    return True  # success
  else:
    s = sets.head  # a set of substitutions for one literal
    for theta in s:
      if compatible(theta, sigma):
        # continue with sigma+theta and remaining sets
        if compatible_substs(sigma + theta, sets.tail):
          return True

def compatible(sigma, theta):
  for x in dom(sigma) inter dom(theta):
    if sigma(x) != theta(x):
      return False  # x has different bindings
  return True
```

Figure 14: subsumption algorithm

Clauses are referenced by a unique integer. A sample certificate for the theorem (1) is shown in Figure 15. The certificate begins with the token `CERTIFICATE`, then the index of the empty clause is specified ("`EMPTY 6`") with the indexes of the axioms. A list of inferences, one per line, follows; each inference is of the form $\Gamma \vdash A$ where Γ is a list of clauses and A is the inferred clause.

The exact formatting of the inferences is designed to simplify the proof checker as much as possible. The proof checker will check each inference using an external prover (in our case, SPASS) to prove the inference, so we chose to write the clauses in the TPTP format. Each inference is thus a non-empty list of tokens of the form `({index} clause)`, the first clause being the conclusion and the following ones being the premises.

In Figure 15, for instance, the first inference line can be read as the sequent (2). Some clauses in the left part of the sequent are axioms, some of them are deduced from other clauses and therefore have their own inference in the certificate. The associated DAG of clauses, the structure of which is to be checked, is shown in Figure 16.

$$(\forall x. p \Leftrightarrow f(x)) \Rightarrow (p \Leftrightarrow \forall x. f(x)) \tag{1}$$

$$\forall x. f(x), p, \forall x. (\neg p \vee \neg f(x)) \vdash \Box \tag{2}$$

```
CERTIFICATE
EMPTY 6
AXIOMS 0, 1, 2, 3
{6} $false {5} ! [X]: f(X) {4} p {3} ! [X]: (~ p | ~ f(X))
{5} ! [X]: f(X) {4} p {0} ! [X]: (~ p | f(X))
{4} p {2} (p | f(skc0)) {1} ! [X]: (~ f(X) | p)
```

Figure 15: Sample proof certificate

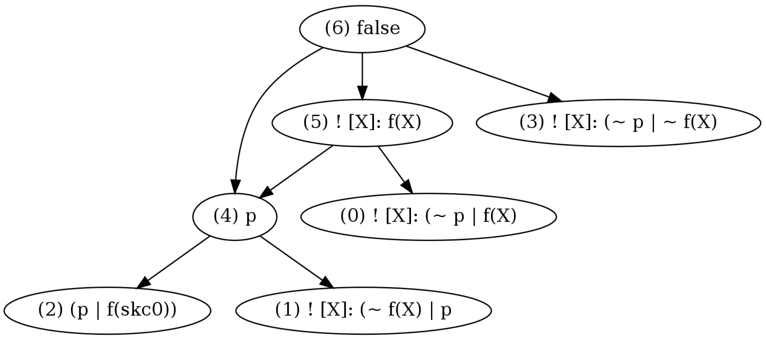

Figure 16: DAG for the proof certificate

We wrote a very simple proof checker in the functional language Haskell. After the certificate has been parsed, the proof checking decomposes in two successive phases:

1. Building a DAG of clauses by checking the inferences. Edges pointing from a clause to other clauses denote that the clause is inferred from the other clauses, and that the corresponding inference has been checked using an external prover.

2. Checking the structure of the DAG; more precisely, checking that the empty clause ($\Box$, or *false*) is in the DAG, and checking that any path starting at the empty clause eventually reaches an axiom.

We also implemented a tool that converts the proof DAG into a `dot` file, to be able to visualize the proofs. An example (for a problem from the SET category of the TPTP library) is shown in Figure 17. The DAG is the same as the one built by the proof checker, but with reversed edges (i.e., edges go from premises to conclusions of inferences). The brightness of nodes indicate which clause is active (light grey), redundant (dark grey) or the empty clause (white). Small round nodes represent inferences or simplifications. The folded corners characterize input axioms.

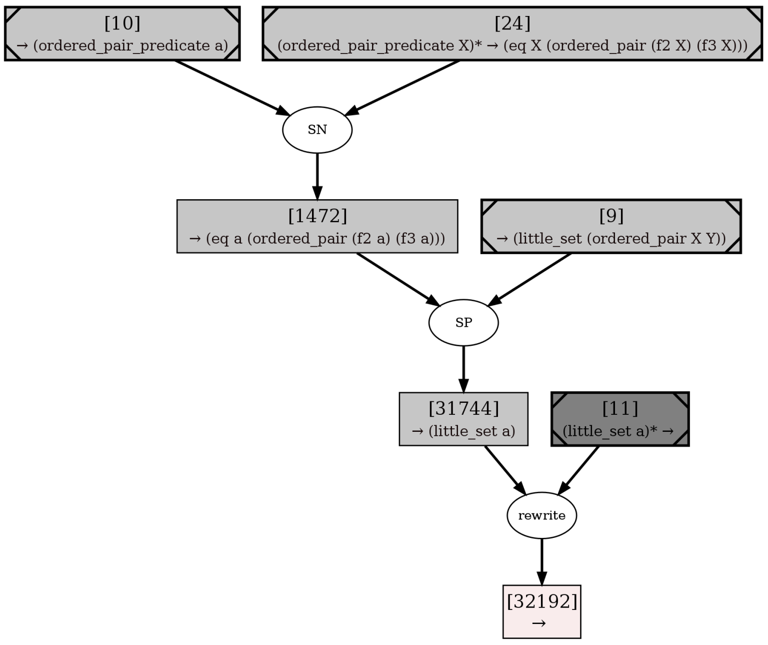
[10]
→ (ordered_pair_predicate a)
[24]
(ordered_pair_predicate X)* → (eq X (ordered_pair (f2 X) (f3 X)))
SN
[1472]
→ (eq a (ordered_pair (f2 a) (f3 a)))
[9]
→ (little_set (ordered_pair X Y))
SP
[31744]
→ (little_set a)
[11]
(little_set a)* →
rewrite
[32192]
→

Figure 17: Visual representation of a proof (problem `SET025-8.p`)

5 Instantiation Mechanisms

The enhanced DPLL algorithm used by most SMT solvers for propositional reasoning has proved to be extremely powerful to deal with a huge number of ground clauses and literals. However, when some clauses contain variables, one must *instantiate* those clauses in order for them to participate in the DPLL search. The challenge is that any non-ground clause stands for an infinite number of ground clauses, so we must choose a finite subset of those. In this section, we detail the classical method used to choose instantiations, and we introduce a new technique that does not require user-defined triggers.

During the search, clauses processed by the superposition prover are sent to the instantiation mechanism in order to be instantiated. This is made possible by the absence of user-provided triggers, since some clauses used for instantiation are not part of the input problem.

5.1 E-matching

Although we did not implement heuristic E-graph matching based on triggers, our instantiation mechanisms use the abstract machine described in [dMB07] – a simple version limited to retrieve all terms in some equivalence class that match a given pattern, rather than a set of patterns. More formally, the problem solved by the algorithm is

$$match(p,t) = \{(\sigma, t') | t' =_E t \wedge \sigma(p) = t'\}$$

for a given pattern p and a term t (t' ranges over terms that exist in the E-graph).

We present here briefly the algorithm used to find terms that match a given pattern modulo E-graph. The pattern is compiled into code for an abstract machine, the instruction set of which is described in Figure 18. Instructions are instances of a structure `vm_instr_t`, and the code a term is compiled into is an array of such instructions. Note that we dropped the `init` instruction from [dMB07], using the instruction `bind` instead; the `choose` instruction is implemented but not used.

To run some code, we need an abstract machine, a stateful structure containing several items:

PC register: a pointer to the *current instruction.*

`reg[]` **array:** an array of registers, each register storing a term.

backtracking stack: a stack `bstack` of instructions used for backtracking.

Upon initialization of the abstract machine, the term to match t is put in `reg[0]` of the abstract machine, and the instruction pointer `pc` is set to the first instruction of the compiled pattern. Then, each call to `e_match_vm_next()` (the function to get the next element of the iterator) runs instructions of the abstract

`bind(i, f, o, next)`	`bstack.push(` `choose-app(o, f, apps_f(reg[i]), 0, next));` `pc = backtrack;`
`check(i, t, next)`	`if (egraph_equal(reg[i], t))` `then pc = next; else pc = backtrack;`
`compare(i, j, next)`	`if (egraph_equal(reg[i], reg[j]))` `then pc = next; else pc = backtrack;`
`yield(index_list)`	`yield substitution` $\{x_{i_j} \to \mathrm{reg}[i_j] \| i_j \in \mathrm{index\_list}\}$; `pc = backtrack;`
`backtrack`	`if (bstack.empty())` `then stop; else pc = bstack.pop();`
`choose-app(o, f, list, i, next)`	`if (i == list.length)` `then pc=backtrack; else {` `list[i] =` $f(t_1,\dots,t_n)$`;` `reg[o+j] =` t_j `for j` $\in \{1,\dots,n\}$`;` `i ++; pc=next;` `}`

Figure 18: Instructions of the E-matching Abstract Machine

machine until a `yield` is executed – in which case the corresponding substitution and term t' are returned – or the execution stops – because `backtrack` was called and the backtracking stack was empty. For a pattern $f(x, g(y), a, x)$, the compiled code will be as follows:

$$\text{bind}(0, f, 1, \text{check}(3, a, \text{compare}(1, 4, \text{bind}(g, 2, 4, \text{yield}(x = 1, y = 5)))))$$

which can also be written as a sequence of instructions with labels, as follows:

l_1: bind$(0, f, 1, l_2)$
l_2: check$(3, a, l_3)$
l_3: compare$(1, 4, l_4)$
l_4: bind$(g, 2, 4, l_5)$
l_5: yield$(x = 1, y = 5)$

Intuitively, during execution, `bind(i,f,o,next)` is used to match the term t in `reg[i]` against the E-graph terms whose first symbol is f. To do so, it gives to a fresh `choose-app` instruction the elements of the equivalence class of t that begin with f, and gives the control to this new instruction (by pushing it on the backtracking stack and setting `pc=backtrack`). There is only one instance of the `backtrack` instruction, since it has no arguments. Its intuitive purpose is to cancel the last choice to select the next possibility. With our set of instructions, choices occur when `choose-app(o, f, list, i, next)` is executed: given the list of possible terms `list`, it pushes `list[i]` into registers (starting at the offset `o`) and remembers what is the next choice by incrementing `i` (so that next time the instruction is run, it will choose the next term in `list`). The instruction `yield` is used to bind variables (the variables of the pattern) to terms currently stored in registers. Instructions `check` and `compare` check constraints respectively between a register and a term (corresponding to a constant in the pattern), and between two registers (corresponding to two occurrences of the same variable in the pattern). This abstract machine can be used as an iterator, which is useful if one wants to find matches on demand.

5.2 Terminator

Heuristic E-graph matching is a widely used technique, but it may perform a lot of useless instantiations if some of the selected triggers are too eager (i.e., the trigger matches too many terms in the E-graph). While retaining the idea of using the terms present in the E-graph to direct instantiation, we sought to find a more conservative algorithm, which would ideally only select for instantiation the grounding substitutions that would close the current (partial) model.

Thus, our purpose was to somehow lift the DPLL unit propagation to first-order clauses – first-order clauses would participate only when instantiating them would yield a ground clause with at most one non-falsified literal, i.e., a unit clause or an empty clause.

For instance, if we had two non-ground clauses $c_1 = p(x) \vee \neg q(x)$ and $c_2 = q(x) \vee r(x)$, in the partial model $q(a), r(a)$ the only "interesting" instantiation is c_1 with $\sigma = [a/x]$, because the resulting ground clause $\sigma(c_1) = p(a) \vee \neg q(a)$ immediately propagates and adds $p(a)$ to the partial model. Selecting $q(x)$ as a heuristic trigger for the two clauses would also instantiate c_2 into a satisfied clause that does not contribute to the (refutation of the) current model.

Overview of the Terminator algorithm

We adapted some ideas from [AO83] to this problem. The aspect that we were interested in was doing hyper-resolution between some non-ground clauses (the ones we try to instantiate) and some unit ground clauses (the DPLL partial model). The Terminator algorithm is designed for unit refutation in the general first-order case (i.e., both non-unit clauses and unit clauses can contain variables), but without any notion of equality nor E-graph.

Terminator maintains a graph of clauses, in which nodes are either unit clauses or non-unit clauses (as arrays of literals). Non directed edges exist between unit clauses and literals if they are unifiable and of opposite polarity (for instance, there is an edge between $p(a)$ and the first literal of $\neg p(x) \vee q(x)$). Terminator then performs unit hyper-resolution between unit clauses and non-unit clauses, by finding substitutions that are compatible with all literals of a clause, or all literals but one if propagation is allowed.

The problem solved by our algorithm is different from the one Terminator solves in two ways: unit clauses are ground, and there is a set of equations that depend on the current model. Our goal is, given a clause $l_1 \vee \cdots \vee l_n$ and a current model, to find some grounding substitutions that, if used to instantiate the clause, yield unit or false ground clauses *in the current model.* It works in two steps (more formal definitions are given in the next section):

1. for each literal l of the clause, find a set of substitutions σ that *close* the literal in the current model, i.e., such that $\sigma(l)$ is false in the current model,
2. *merge* substitutions to find some substitutions that close at least $n-1$ literals in the current model.

Adapting Terminator to Instantiation Modulo E-graph

We introduce some definitions that are useful to explain our adaptation of the Terminator algorithm. For a given signature $\mathscr{T}(F, S, X)$, let S be the set of substitutions over $\mathscr{T}(F, S, X)$; $P(S)$ is then the set of sets of substitutions that are valid over the signature.

Merge: the *merge* between two sets of substitutions s and s' in the current model, noted $s * s'$, is defined by

$$s * s' = \{\theta \in S | \exists \sigma \in s. \exists \sigma' \in s'. \forall x \in X. \theta(x) =_E \sigma(x) =_E \sigma'(x)\}$$

The binary operator $*$ is commutative and associative, so we will write $s_1 * s_2 * \cdots * s_n$ for $(\ldots(s_1 * s_2) * \ldots) * s_n$. Note that the merge of two sets of substitutions is only meaningful given a ground (partial) model.

Closing substitutions: Unlike the Terminator algorithm, we have no need of actually using a graph with links between clauses. We only need to know, in the current model, for each literal l of a clause, the set of *closing substitutions.* We say a substitution σ *closes* l if $\sigma(l)$ is false in the current model. We note $c(l) \in P(S)$ for the set of closing substitutions for l.

Eligible for instantiation: A substitution σ is *eligible for instantiation* of a clause $C = l_1 \vee \cdots \vee l_n$ if σ closes at least $n-1$ literals of C in the current model. We can compute such substitutions from the closing substitutions of each literal $l_1 \ldots l_n$ by merging them.

Implementing the n-ary Merge

Once sets of closing substitutions are created, for each clause, Terminator must merge those sets to find those which are eligible for instantiation. This is quite similar to the technique we used for the subsumption check (Section 4.6). However, if there are several literals in a clause C with many closing substitutions each, the number of combinations to check can grow quickly: $\prod_{l \in C} c(l)$ combinations. This becomes worse when we also consider substitutions that close all literals but one: $\sum_{l \in g(C)} \prod_{l' \in C, l' \neq l} c(l')$ combinations, where $g(C)$ is the set of literals of C that are *groundable*, i.e., which set of variables is included in the set of variables of the other literals. Intuitively, a literal is *groundable* if any substitution that grounds the other literals also grounds it. More formally,

$$l \in g(C) \Leftrightarrow vars(l) \subseteq \bigcup_{l' \in C \setminus \{l\}} vars(l')$$

To reduce the number of merge operations (merging two substitution sets together can be expensive), we introduce a divide-and-conquer mechanism. The implementation of the merge operation itself is orthogonal to this problem – see the Section on implementation 5.4. We memoize results of the merge operations. Let Given a clause C with n literals $l_1, \dots, l_n$, we associate to each literal l_i a point $p_{l_i} \in \mathscr{P}$, with coordinates $(\delta_{ji})_{j=1,\dots,n}$, where δ is the Kronecker delta defined as

$$\delta_{ij} = \begin{cases} 1 \text{ if } i = j \\ 0 \text{ if } i \neq j \end{cases}$$

We define a function $s : \mathscr{P} \to P(S)$ that maps, to any $p \in \mathscr{P}$, a set of substitutions, as follows:

- $s((0,\dots,0)) = \emptyset$
- For points p_i with exactly one non-null coordinate, $s(p_i)$ is $c(l_i)$, the set of substitutions that close the literal l_i.
- For any other point $p = (b_1,\dots,b_n)$ with $b_i \in \{0,1\}$ (b stands for "bit"), $s(p) = *_{j\in\{1,\dots,n\},b_j=1} p_{l_j}$. The point p is the result of the merge of the $c(l_i)$ for all b_i equal to one.

Points in $\mathscr{P}$ then correspond to results of merges, with the initial points (p_{l_i} with coordinates $(\delta_{ij})_j$)) corresponding to closing substitutions of the literals of the clause. Let p_{false} be the vertex whose coordinates are $(1,\dots,1)$, and p^i_{unit} be the vertices with coordinates $(1-\delta_{ij})_j$. Then, $s(p_{false})$ is the set of substitutions that, used to instantiate C, yield an empty clause, and $s(p^i_{unit})$ is the sets that yield unit clauses (the unit literal is l_i).

We want to compute the sets $s(p_{false})$ and $s(p^i_{unit})$. To do so, we define a function `get_subst_set` that computes the set of substitutions $s(p)$ at a given position p. Figure 19 sketches the algorithm. The computation takes place at most once per point, because results are cached. The function `get_subst_set` takes as arguments the memoizing map m, used to cache results, and a point $p \in \mathscr{P}$, and returns $s(p)$. Coordinates are here considered as bit fields, like integers in C (which is how they are actually represented). The function *split* is then defined over those bit fields, taking an integer and returning two integers. Assuming (l,h) are the indices of the lowest true bit in p, respectively the highest true bit in p (e.g., if $p = (0,1,1,1,0,1,0) = 0111010$, then $l = 1$ and $h = 5$), starting from 0, *split* divides it into two disjoint sets of bits p_{low} and p_{high} such that $p = p_{low}|p_{high}$ (bitwise or). The divide-and-conquer method defines p_{low} as p where only the true bits whose indexes are between l and $\left\lfloor \frac{l+h}{2} \right\rfloor$, and p_{high} as p where only the bits between $\left\lceil \frac{l+h}{2} \right\rceil$ and h are kept. For example, here is the computation of p_{low} and p_{high} for $p = 0011101101000$:

$$\begin{array}{c|cccc}
p & 00 & 1110 & 1101 & 000 \\
p_{low} & 00 & 0000 & 1101 & 000 \\
p_{high} & 00 & 1110 & 0000 & 000 \\
 & & \text{h} & \text{l} &
\end{array}$$

$$\begin{aligned}
\text{get\_subst\_set}(m,p) &\rightsquigarrow c(l_i) \text{ if } p = (\delta_{ij})_j \\
\text{get\_subst\_set}(m,p) &\rightsquigarrow m(p) \text{ if } m \text{ contains } p \\
\text{get\_subst\_set}(m,p) &\rightsquigarrow m[p] := s;\ \text{return } s \\
&\quad \text{where} \\
&\qquad s_{low} = \text{get\_subst\_set}(m, p_{low}) \\
&\qquad s_{high} = \text{get\_subst\_set}(m, p_{high}) \\
&\qquad (p_{low}, p_{high}) = split(p)
\end{aligned}$$

Figure 19: The `get_subst_set` function

Some Implementation Details

To avoid doing the same instantiations several times, in each clause, we keep the set of hashes of substitutions used so far with this clause (to avoid doing them again). If a substitution's hash is not in this set, then the clause is instantiated and the hash is put in the set. The hash is on 32 bits, so false positives can occur, but we believe they are not frequent enough to seriously threaten the success of the prover (which is already incomplete, anyway).

We also need an efficient representation of sets of substitutions. From this perspective, we represent such a set as a linked list of substitutions sorted by increasing *hash modulo E-graph*. The hash modulo E-graph is designed so that substitutions that are syntactically different, but that are the same in the current model, get hashed into the same value. It is computed by representing the substitution as an array of key/value where each pair is $(x, root[\sigma(x)])$ for $x \in dom(\sigma)$, sorting the array by increasing key, and hash it. This way, if two substitutions σ and σ' are such that $\forall x.\sigma(x) =_E \sigma'(x)$, then they have the same hash; since they will yield the same ground instance *in the current model* we can keep only one of them. Note that this relation between substitutions is also (potentially) only valid in the current model and therefore cannot be kept after the solver backtracks — therefore we do not use hashing modulo E-graph to remember which substitutions have been used to instantiate a clause.

Constraints

On many SMT problems, many non-ground axioms use theories. Since Terminator, like the superposition calculus, is based on syntactic equality of uninterpreted terms (hence the use of unification), it fails to find substitutions that close literals *modulo theories*.

Therefore, we need more general ways to find closing substitutions (e.g., when a clause contains a literal $a \simeq f(y)$, or a literal $a < f(y)$, purely syntactic matching will fail). In some cases, for instance a literal $f(x) \simeq g(x)$, we simply extend our way of finding closing substitutions by considering the set of terms T_f that match $f(x)$, the set of terms T_g that match $g(x)$, and taking the set of $\{x \to t\}$ for $t \in T_f \cap T_g$ whenever $f(t) \neq_E g(t)$ as answer. When it makes no sense to generate substitutions between a literal and the current model (such as the literal $x < y$: there are too many possibilities for binding x and y, and they are hidden in the current state of the arithmetic solver) we have to find another way of dealing with the literal.

So, when a literal is not "specific" enough (i.e., it has a variable directly in a $\simeq$ or $<$ relation) or it contains too many theory-specific terms (the E-matching algorithm we use will miss the possibility of matching $x + f(3)$ and a with $[a - f(3)/x]$) we put the literal apart and use it as a *constraint*. It will not appear in the list of literals, will not have a proper set of closing substitutions, but a substitution σ is only eligible for instantiation of a clause C if $\sigma(p)$ is false in the current model, for all constraints p of C. However, when the clause is instantiated, constraints are instantiated like any other literal, since they are initially part of the clause.

The problem with this approach is that the implementation gets very complicated and full of corner cases. Moreover, it can be too restrictive, i.e., it prevents some substitutions from being instantiated, although they would have been useful. We present a more general framework to describe instantiation mechanisms and the way they select substitutions.

5.3 Relational Instantiator

Terminator does not allow to use only a subterm of some literal as a pattern (in the case of an interpreted predicate or term, like $a \simeq f(g(x)) + 1$, $f(g(x))$ may be a good pattern), or if we only have constraints (which makes Terminator drop the clause, because it has no way of *generating* substitutions)? We sought more flexibility, and came up with a generalization of the Terminator algorithm. Because this algorithm borrows some concepts from relational databases, we call it *Relational Instantiator*. On purely non-interpreted problems (without theories) it behaves the same way as Terminator.

The basic idea is to define operators that work on sets of substitutions. Some operators create substitutions, some others filter or combine them. We want a set of operators that allows us to express the behavior of Terminator, but also to express other behaviors that may be more permissive, possibly as permissive as heuristic E-graph matching.

At this point, we can note an analogy between a set of substitutions which all have the same domain, and a set of tuples in the relational algebra used for databases [Cod70]. A substitution σ which domain is $\{x_1,\dots,x_n\}$ can be seen as a tuple $(\sigma(x_1),\dots,\sigma(x_n))$ for some fixed ordering of variables. A *join* on two set of tuples is then similar to the merge operation we defined; although we do not require equality of the elements of the tuples , but only equality modulo E-graph. Constraints are akin to *filters* (or *selection* with a *where* clause, in the SQL language). We define the following operators on sets of substitutions:

join ($\bowtie$): merges n sets of substitutions. We weaken the definition to allow substitutions in the result to be incompatible with at most one of the input set, as long as the domain of the resulting substitution is the union of the domains of the input sets of substitutions. We always flatten nested *joins*, to give a clear semantic to the "at most one". More formally, if $s_1,\dots,s_n$ are sets of substitutions,

$$\begin{aligned}\bowtie_{i=1\dots n} s_i \equiv \{\theta \in S | dom(\theta) = \bigcup_{i=1}^{n} dom(s_i) \wedge \\ \exists i_1,\dots,i_{n-1} \in \{1,\dots,n\}^{n-1}.\ \text{pairwise distinct}, \\ \exists \sigma_{i_1} \in s_{i_1},\dots,\sigma_{i_{n-1}}. \in s_{i_{n-1}}, \\ \forall x.\theta(x) =_E \sigma_{i_1}(x) =_E \dots =_E \sigma_{i_{n-1}}\}\end{aligned}$$

filter (φ): Given an input set of substitutions s, and a predicate p such that $vars(p) \subseteq \bigcap_{\sigma\in s} dom(\sigma)$, $\varphi(p,s)$ yields the set of substitutions containing only those which satisfy the given predicate p (i.e., such that $\sigma(p)$ holds in the current model),

generate (γ): Given a pattern t (a non-ground term), $\gamma(t)$ yields the set of substitutions obtained by E-matching t against the current (partial) model,

union ($\cup$): Given n sets of substitutions, it returns the union of the sets,

empty (ϵ): It returns the empty set, containing no substitution.

Using those operators, we can then build *queries*. A query is a tree in which every node is annotated with an operator, and possibly some parameters for the operator. We associate to each node N a set of variables $b(N)$ that we call the *bound* variables of the node, recursively defined as follows:

$$\begin{aligned} b(\epsilon) &= \emptyset \\ b(\varphi(p,N_1)) &= b(N_1) \\ b(\gamma(t)) &= vars(t) \\ b(N_1 \cup N_2) &= b(N_1) \cap b(N_2) \\ b(N_1 \bowtie N_2) &= b(N_1) \cup b(N_2) \end{aligned}$$

$b(N)$ is the set of variables that will be bound by any substitution yielded by the node. We then associate one such query with each clause in the Relational Instantiator, with the invariant that for any clause C with query q, the root node $root(q)$ satisfies $b(root(q)) \supseteq vars(C)$. Such a query is said to be a *grounding query* for the clause C. The Relational Instantiator uses the DAG associated with each clause to compute, in the current model, a set of grounding substitutions for this clause, and instantiate the clause using the substitutions that have not been used yet. Terminator, in its simplest version, without constraints, can be expressed as a simple n-ary *join* over *generate* nodes (one *generate* for each literal in the clause). Adding constraints corresponds to wrapping this join in nested *filter* nodes (one per constraint).

For instance, let C be the clause $p(x) \vee \neg q(y) \vee f(x,y) \simeq a$. Then, the straightforward query that we generate automatically would be $\gamma(\neg p(x)) \bowtie \gamma(q(y)) \bowtie \varphi(f(x,y) \not\simeq a,\ \gamma(f(x,y)))$. The query satisfies the bound variables constraint, as the *join* binds both x and y. A graphical representation of the query is shown in Figure 20.

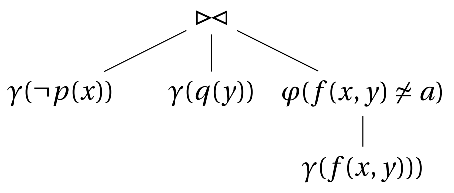

Figure 20: Relational Instantiation Query

When a non-ground clause is added to the Relational Instantiator (ground clauses are directly forwarded to the SMT solver), a grounding query is computed. The basic algorithm to compute the query is shown in Figure 21 as a function $c : \mathcal{T}(F,S,X) \to D$ where D is the set of possible queries. The function c takes a term and returns a DAG. Then, we "optimize" the DAG, in a way that is quite similar to query optimization in relational databases; the goal is to make the query faster to evaluate. An abstract version of the optimization algorithm is pictured in Figure 22. It consists of two functions o_1 and o_2. The first function, o_1, is used to make more queries *valid* (i.e., a grounding query) by shuffling filters at points where they make sense. For instance, the compilation phase may yield, for a clause $p(x) \vee x+1 \simeq a$, a query $\gamma(\neg p(x)) \bowtie \varphi(x+1 \not\simeq a, \epsilon \bowtie \epsilon)$; The first step of optimization will reduce this invalid DAG to $\varphi(x+1 \not\simeq a, \gamma(\neg p(x)))$ where the filter $\varphi(x+1 \not\simeq a,\ .)$ actually operates on a set of substitutions that bind x. Since $\bowtie$ and $\cup$ are commutative operators, we can shuffle their arguments. The main thing the second function, o_2, does is pushing filters as deep as possible in the tree, the rationale being that the sooner we reduce the size of the sets of substitutions, the easier it is to compute joins.

Whenever the intantiator is called, all clauses are examined sequentially.

Each clause has its own relational query (a DAG, in memory), and substitutions are computed top-down by evaluating the set of substitutions at leaves of the DAG (of kind γ or ϵ) and recursively to the root of the DAG. Joins are computed using the same divide-and-conquer algorithm that is used in Terminator. We can share similar leaves among clauses during a single instantiation round, because the state of the E-graph will be the same, and therefore the set of generated substitutions will also be the same.

$$\begin{array}{rcl}
c(a \vee b) & \rightsquigarrow & c(a) \bowtie c(b) \\
c(t_1 \simeq t_2) & \rightsquigarrow & \varphi(t_1 \not\simeq t_2, c(t_1) \bowtie c(t_2)) \\
c(t_1 \not\simeq t_2) & \rightsquigarrow & \varphi(t_1 \simeq t_2, c(t_1) \bowtie c(t_2)) \\
c(t) & \rightsquigarrow & \gamma(\neg t) \text{ where } t \text{ uninterpreted Boolean} \\
c(t) & \rightsquigarrow & \gamma(t) \text{ where } t \text{ uninterpreted term} \\
c(t_1 \circ t_2) & \rightsquigarrow & c(t_1) \bowtie c(t_2) \text{ where } \circ \text{ interpreted operator} \\
c(t) & \rightsquigarrow & \epsilon \text{ otherwise}
\end{array}$$

Figure 21: Compilation of a clause to a relational query

$$\begin{array}{rcl}
o_1(t_1 \cup t_2)) & \rightsquigarrow & o_1(t_1) \cup o_1(t_2) \\
o_1(\varphi(p,t)) & \rightsquigarrow & \varphi(p, o_1(t)) \\
o_1(\varphi(p,t) \bowtie t') & \rightsquigarrow & o_1(\varphi(p, t \bowtie t')) \text{ where } \mathit{vars}(p) \not\subseteq \mathit{vars}(t) \\
o_1(\epsilon \cup t) & \rightsquigarrow & o_1(t) \\
o_1(\epsilon \bowtie t) & \rightsquigarrow & o_1(t) \\
o_2(\varphi(p, t_1 \bowtie t_2)) & \rightsquigarrow & o_2(\varphi(p, t_1) \bowtie t_2) \text{ where } \mathit{vars}(p) \subseteq \mathit{vars}(t_1)
\end{array}$$

Figure 22: Optimization of query

Because interpreted literals are harder to match against, and may be over-filtered (because $\varphi(p,t)$ where p interpreted asks the E-graph for the truth value of $\sigma(p)$ for each candidate substitution σ, and the E-graph by itself may not have all the information), a second DAG is built from the first, by removing filters and replacing as many joins $\bowtie$ by unions $\cup$ without losing the grounding property. For instance, the query DAG in Figure 20 will be translated to a more *eager* DAG, as shown in Figure 23. However, since this eager DAG potentially produces much more substitutions, we limit the number of instantiations that it can trigger at every call to the instantiator, whereas the regular DAG is not limited.

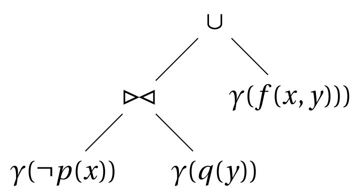

Figure 23: Eager Relational Instantiation Query

Soundness and Completeness

Both Terminator and the Relational Instantiator are obviously sound, because they perform no inference at all. The only thing they do is instantiating clauses that are theorems within the input axioms (whether they are axioms from the input, or consequences thereof derived by the superposition solver). Since the instantiations are ground (this property is ensured by the instantiation algorithms), they can be properly handled by the SMT solver. Therefore, YICES's soundness only depends on the soundness of the superposition prover and the soundness of the SMT solver.

In case the input contains interpreted theories (like arithmetic), the instantiation can miss some substitutions that would lead to the solution; if equalities appear, for instance in the problem $p(a), \neg p(f(f(a))), f(x) = x$, the instantiation will fail because it is not explicit that $f(a) \neq a$. Even in other cases, the ground terms present in the input may not be enough to trigger any instantiation, even though the problem is unsatisfiable. Therefore, we do not believe the instantiation mechanisms to be complete on any usual non-ground fragment of first-order logic.

For instance, on a non-ground problem $q(x) \vee r(x), q(x) \vee \neg r(x), \neg q(x) \vee r(x), \neg q(x) \vee \neg r(x)$, the instantiation mechanisms will fail because there are no ground terms in the DPLL trail, so no non-ground pattern ever gets instantiated (no instantiation would immediately get unit or false). This problem would however be solved by the superposition prover, e.g., by the resolution steps

$$\dfrac{\dfrac{q(x) \vee r(x) \qquad q(x) \vee \neg r(x)}{q(x)} \qquad \dfrac{\neg q(x) \vee \neg r(x) \qquad \neg q(x) \vee r(x)}{\neg q(x)}}{\square}$$

5.4 Implementation

We use a simple representation of sets of substitutions as sorted linked lists of arrays (allowing two substitutions equal modulo E-graph to appear only once in the set). Therefore, the join (or merge) of sets of substitutions is potentially very slow. We did not implement the P-representation of substitutions described in [AO83], nor the S-representation described in [MŁK08]. The reason for that was a lack of time. However, the selection of overly general patterns for E-matching seems to be worse a bottleneck in both algorithms.

In the case of the Relational Instantiator, possible workarounds for many cases that lead to combinatorial explosions would be to push filters directly into the *generate* leaves, and to drop non-necessary branches from big joins (branches which we can get rid of without reducing the domain of the resulting instantiations). Another possibility is to use iterators to compute sets of substitutions *lazily* at each node; this way, we may be able to avoid generating all matching substitutions at leaves of the queries. In this case, we would need a *fair* implementation of joins, one which evaluates all input iterators the same way (unlike the classical "nested loops" implementation, that will evaluate lazily the outermost iterator, but eagerly the inner ones).

Our simple, straightforward implementation also ignores some possible tricks for finding matching terms in interpreted terms. For instance, it cannot match $f(x+1)$ with $f(a)$. In addition, the pattern selection can behave poorly in arithmetic terms (say, $x+(k*(f(x)-1))$). We believe that the lack of fine-tuning and handling of special cases has a significant impact on the ability of YICES to solve some benchmarks (for instance, the *Simplify* benchmarks encode Booleans using uninterpreted symbols and integers, rather than using the notion of Booleans built in SMTLIB; this makes instantiation harder for our algorithms).

5.5 Putting it all together: Integration and Proof Search

The Terminator algorithm and the Relational Instantiators are only an *instantiation* mechanism. They will never do any *inferences*, only provide the SMT solver with new clauses that may help it prove the unsatisfiability of the problem. Similarly, the superposition solver by itself will most likely fail to refute any problem containing arithmetic. We therefore interleave the execution of the different components until we find a proof or run out of time.

The superposition prover is designed to allow step-by-step execution of the main loop. The main function of YICES combines the usual SMT ground decision procedure (denoted "`smt_solve`") one instantiation mechanism (called "`instantiate`") and the superposition solver (called "`superposition`") run for a given number of iterations each time it called.

Components are coordinated through a simple state machine. Each state of the automaton corresponds to the call to a component, and transitions are conditioned to the output of this call. A schema of the state machine is pictured in Figure 24. To ensure termination, we put a bound on the number of times the loop between superposition, instantiation and smt_solve is performed before returning "`unknown`".

This main loop gives control to the different components in turns. Each component has a specific role: The instantiation mechanism, using a set of non-ground clauses, try to participate in the ground search by instantiating the clauses; the SMT solver tries to refute or to find a model for the ground part of

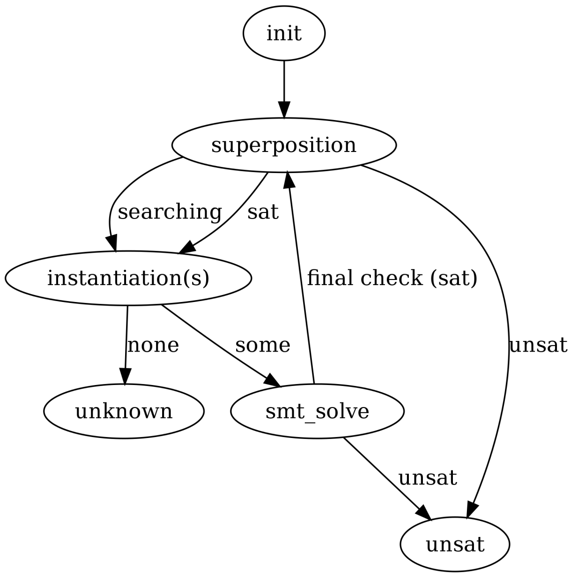

Figure 24: State machine for the main solving function

the problem; the superposition prover tries to refute the non-ground part of the problem, and in addition produces new clauses that are sent to the instantiation mechanism. Any clause that has been processed by the superposition prover is available for instantiation.

6 Experimental Results

6.1 Performance of the Superposition Solver

We have run several versions of the superposition prover (named `yices_tptp`, which is the superposition prover using a TPTP parser rather than the usual SMTLIB parser) against problems from the TPTP benchmarks [Sut09] [8] with a (short) timeout of 30s. For comparison, we have run SPASS on the same benchmarks. The results show, unsurprisingly, that SPASS is better than our prover; however our prover manages to solve several hundred of problems within a few seconds each. Considering the low amount of time that we dedicated to the prover heuristics, and the relative simplicity of the implementation (compared to a state of the art prover like SPASS), this result is encouraging. Results are shown in Table 1. The provers used for this benchmark are SPASS, SPASS with our clausification algorithm, and yices_tptp, all with a 30-second timeout. Note that some categories of problems, such as `CSR` (common sense reasoning) or `CAT` (category theory) are hard — lots of problems are not solved by any of the provers, and they are rated with a high difficulty level. That explains the low rate of success compared to the SMTLIB benchmarks, presented in the next section.

One important limitation of `yices_tptp` is that, since our implementation of substitution trees is, unfortunately, too slow, we use the simple indexing on the first symbol which does not scale. Indeed, operations on a substitution tree involve a matching or an unification at each node of the tree, which proves to be costly with our representation of substitutions and variables (an abstract machine like Dedam [NRV97] may be more suited to the problem). The implementation is quite complicated (roughly 1,000 lines of C code) and we did not have time to optimize it thoroughly.

prover	solved	unsolved	% success
yices_tptp	1729	3588	32.5
SPASS	2279	3038	42.9
SPASS + yices	2112	3205	39.7

Table 1: Experimental Results of the Superposition Prover

6.2 Results for First-Order YICES

We ran YICES with the Relational Instantiator and the superposition prover enabled on benchmarks of the AUFLIA and AUFLIRA[9] category of SMTLIB. We compare the results with the performance of YICES1 on the same benchmarks.

[8] which can be found at `http://www.cs.miami.edu/~tptp/`

[9] respectively: Array, Uninterpreted Functions and Linear Integer Arithmetic, and the same with Linear Real Arithmetic

We also show the number of benchmarks that are solved using internalization only (simplification then reduction to CNF).

The results are shown in Table 2. For each prover, the number of problems solved within different time ranges are indicated. We can see that YICES1 is better than our implementation, which we believe is caused by several factors:

1. the lack of fine-tuning of our algorithms,
2. the better E-matching (w.r.t. arithmetic terms, especially) of YICES1,
3. the absence of quantifier elimination of our implementation.

However, YICES2 solves some problems that YICES1 fails to solve. YICES2 with the Relational Instantiator solves 376 problems on which YICES2 fails, and YICES2 with the Terminator solves 473 such problems.

On the other hand, Terminator and the Relational Instantiator have quite comparable results; a more detailed analysis shows that 904 problems are solved by only one of them. It would be interesting to use both instantiators at the same time (assuming they share the set of clauses to instantiate, in order not to duplicate instantiations).

prover	≤ 1s	≤ 10s	≤ 100s	unsolved
YICES1	32420	237	66	1758
YICES2 with Rel. Inst.	25910	2801	1469	4301
YICES2 with terminator	25775	2781	1388	4537
internalizer	21254	0	0	13227

Table 2: Experimental Results of YICES

7 Conclusions and Future Work

Related Work

Handling of quantifiers via heuristic E-graph matching first appeared in Simplify [DNS05]. SPASS+T [PW06] is a black-box integration of an SMT solver in the saturation-based prover SPASS [WSH$^+$07]. DPLL(Γ) [dMB08b] is a tight integration of a superposition solver inside an SMT solver, in which the superposition prover can do inferences based on the current model, which may lead to the withdrawal of inferred clauses — our integration of the superposition prover is not as tight, which avoids the additional complexity of backtracking. SMELS [LT] uses superposition rules between non-ground clauses and *justified* literals from the partial model (the justification being similar to the process to build lemmas in usual SMT solvers). Instantiation-based theorem provers like iProver [Kor] use a ground procedure coupled with an instantiation mechanism.

Conclusion

In this thesis, we have presented our work to integrate non-trivial first-order reasoning techniques in YICES. A superposition prover is used to generate new clauses, its execution is interleaved with the ground procedure; an instantiation mechanism is used to try to refute the current ground model using the clauses processed by the superposition prover. The approach is a novel combination of SMT and Superposition; Empirical results show decent initial results although our implementation is not yet as good as state-of-the-art solvers.

We believe the main limitation of our current system is the limited handling of theories other than equality. Heuristic E-matching can choose a trigger and be lucky enough to perform instantiations that will lead it to a proof, but our attempt of limiting instantiation with fewer heuristic mechanisms is easily fooled by, say, complicated arithmetic terms.

Although the experimental results are mixed compared to state-of-the-art provers based on heuristic E-matching, we think that alternative instantiation mechanisms are worth exploring. The coupling of an SMT ground procedure and a superposition prover helps proving some problems without any instantiation, and contributes to solving other problems by generating new clauses (in particular, small clauses) that can be used for instantiation. We are quite happy with our choice of using the calculus of E ([Sch02]), as it allows fine tuning of heuristics — e.g., to favor the processing of ground clauses, or of unit clauses — and is simple enough we could implement it in a few months with additional, specialized rules. The handling of theories in first-order provers is an active research topic (there even is a new category in TPTP that is dedicated to first-order with arithmetic).

Future Work

The solver implementation could be extended in several ways, including improving the performance of the implementation of several components, or improving the algorithms themselves. Possible directions include:

- Improving the implementation of the superposition prover, notably by adding an efficient general-purpose term index (especially for unification) and a non-perfect index for subsumption (such as the *Feature Vector Indexing* [Sch04] that is used in E).

- Improving the heuristics of the superposition prover, especially w.r.t. the SMT procedure (e.g., use the VSIDS [MMZ+01] to influence the selection of the given clause).

- Improving the calculus of the superposition prover to enhance the way it deals with SMTLIB theories, such as arithmetic or arrays, perhaps using theory resolution [Sti85].

- Improving the instantiation algorithms by making them incremental (i.e., avoid recomputing all the E-matching substitutions for each literal of each clause) and improving the implementation (efficient representation of sets of substitutions).
- Implementing some quantifier elimination techniques in the simplification phase, before the reduction to CNF. By eliminating some quantified formulæ that contain interpreted symbols, this technique can help in cases where E-matching is incomplete (and simplifies the problem in many case).

References

[AO83] Grigoris Antoniou and Hans Jürgen Ohlbach. TERMINATOR. In *IJ-CAI'83*, pages 916–919, 1983.

[BS01] Franz Baader and Wayne Snyder. Unification theory. In Robinson and Voronkov [RV01], pages 445–532.

[BT07] Clark Barrett and Cesare Tinelli. CVC3. volume 4590 of *Lecture Notes in Computer Science*, pages 298–302, July 2007.

[Cod70] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13:377–387, June 1970.

[DDM06] B Dutertre and L De Moura. The Yices SMT solver. *Tool paper at http://yices.csl.sri.com/tool-paper.pdf*, pages 1–5, 2006.

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5:394–397, July 1962.

[dMB07] Leonardo de Moura and Nikolaj Bjørner. Efficient E-matching for SMT solvers. In *Conference on Automated Deduction (CADE), Bremen, Germany*, 2007.

[dMB08a] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In C. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *Lecture Notes in Computer Science*, chapter 24, pages 337–340. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2008.

[dMB08b] Leonardo de Moura and Nikolaj Bjørner. Engineering DPLL(T) + saturation. In *Proc. 4TH IJCAR*, 2008.

[dMDS07] Leonardo de Moura, Bruno Dutertre, and Natarajan Shankar. A tutorial on satisfiability modulo theories. In *Proceedings of the 19th international conference on Computer aided verification*, CAV'07, pages 20–36, Berlin, Heidelberg, 2007. Springer-Verlag.

[DNS05] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: A theorem prover for program checking. *J. ACM*, 52:365–473, May 2005.

[GS94] Peter Graf and Im Stadtwald. Substitution tree indexing. Technical Report MPI-I-94-251, Max-Planck-Institut Für Informatik, October 1994.

[Kor] Konstantin Korovin. System Description: iProver – An Instantiation-Based Theorem Prover for First-Order Logic.

[Löc06] Bernd Löchner. Things to know when implementing KBO. *J. Autom. Reason.*, 36:289–310, April 2006.

[LT] Christopher Lynch and Duc-Khanh Tran.

[McC90] William McCune. Experiments with discrimination-tree indexing and path indexing for term retrieval. *Journal of Automated Reasoning*, 9:9–2, 1990.

[McC95] William W. McCune. OTTER 3.0 Reference Manual and Guide, 1995.

[MŁK08] Michał Moskal, Jakub Łopuszański, and Joseph R. Kiniry. E-matching for fun and profit. *Electron. Notes Theor. Comput. Sci.*, 198:19–35, May 2008.

[MM82] Alberto Martelli and Ugo Montanari. An Efficient Unification Algorithm. *Transactions on Programming Languages and Systems (TOPLAS)*, 4(2):258–282, 1982.

[MMZ$^+$01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: engineering an efficient SAT solver. In *Proceedings of the 38th Annual Design Automation Conference*, DAC '01, pages 530–535, New York, NY, USA, 2001. ACM.

[NO78] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1978.

[NOT06] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT Modulo Theories: From an Abstract Davis-Putnam-Logemann-Loveland Procedure to DPLL(T). *Journal of the ACM*, 53:2006, 2006.

[NR99] Robert Nieuwenhuis and Albert Rubio. Paramodulation-based theorem proving. In Robinson and Voronkov [RV01].

[NRV97] Robert Nieuwenhuis, José Rivero, and Miguel Vallejo. Dedam: A kernel of data structures and algorithms for automated deduction with equality clauses. In William McCune, editor, *Automated Deduction—CADE-14*, volume 1249 of *Lecture Notes in Computer Science*, pages 49–52. Springer Berlin / Heidelberg, 1997. 10.1007/3-540-63104-6_5.

[NW01] Andreas Nonnengart and Christoph Weidenbach. Computing small clause normal forms. In Robinson and Voronkov [RV01], pages 335–367.

[Pel86] Francis Jeffry Pelletier. Seventy-Five Problems for Testing Automatic Theorem Provers. *J. Autom. Reasoning*, 2(2):191–216, 1986.

[PW06] Virgile Prevosto and Uwe Waldmann. SPASS + T. *Proceedings ESCoR: FLoC'06 Workshop on Empirically Successful Computerized Reasoning*, pages 18 – 33, 2006.

[RLT06] Silvio Ranise, Loria, and Cesare Tinelli. The SMT-LIB standard: Version 1.2, 2006.

[Rob] J. A. Robinson. A machine-oriented logic based on the resolution principle (1965).

[RSV01] I. V. Ramakrishnan, R. C. Sekar, and Andrei Voronkov. Term indexing. In Robinson and Voronkov [RV01], pages 1853–1964.

[RV] Alexandre Riazanov and Andrei Voronkov. System description: Vampire 1.0.

[RV01] John Alan Robinson and Andrei Voronkov, editors. *Handbook of Automated Reasoning (in 2 volumes)*. Elsevier and MIT Press, 2001.

[SB05] Stephan Schulz and Maria Paola Bonacina. On handling distinct objects in the superposition calculus. In *In Notes 5th IWIL Workshop*, pages 11–66, 2005.

[Sch02] Stephan Schulz. E - A Brainiac Theorem Prover, 2002.

[Sch04] Stephan Schulz. Simple and Efficient Clause Subsumption with Feature Vector Indexing. In *Proc. of the IJCAR-2004 Workshop on Empirically Successful First-Order Theorem Proving*. Elsevier Science, 2004.

[Sti85] Mark E. Stickel. Automated Deduction by Theory Resolution. *Journal of Automated Reasoning*, 1:333–355, 1985.

[Sut09] G. Sutcliffe. The TPTP Problem Library and Associated Infrastructure: The FOF and CNF Parts, v3.5.0. *Journal of Automated Reasoning*, 43(4):337–362, 2009.

[WSH$^+$07] Christoph Weidenbach, Renate Schmidt, Thomas Hillenbrand, Rostislav Rusev, and Dalibor Topic. System Description: SPASS Version 3.0. In Frank Pfenning, editor, *Automated Deduction – CADE-21*, volume 4603 of *Lecture Notes in Computer Science*, pages 514–520. Springer Berlin / Heidelberg, 2007. 10.1007/978-3-540-73595-3_38.