Prime Implicate Generation in Equational Logic (extended abstract)*

Mnacho Echenim[1], **Nicolas Peltier**[1], **Sophie Tourret**[2]
[1] CNRS, LIG, University of Grenoble Alpes, Grenoble, France
[2] Max Planck Institute for Informatics, Saarbrücken
[Mnacho.Echenim|Nicolas.Peltier]@univ-grenoble-alpes.fr, sophie.tourret@mpi-inf.mpg.de

Abstract

A procedure is proposed to efficiently generate sets of ground implicates of first-order formulas with equality. It is based on a tuning of the superposition calculus [Nieuwenhuis and Rubio, 2001], enriched with rules that add new hypotheses on demand during the proof search. Experimental results are presented, showing that the proposed approach is more efficient than state-of-the-art systems.

1 Introduction and Motivations

Abductive reasoning is the process of inferring, given a set of axioms $\mathcal{A}$ and a formula ϕ, a set of assertions $\mathcal{H}$ such that $\mathcal{A} \cup \mathcal{H} \models \phi$. The set $\mathcal{H}$ may be viewed as a set of hypotheses that are sufficient to ensure the validity of the entailment $\mathcal{A} \models \phi$, or as a set of explanations of ϕ. Such hypotheses must be economical and plausible, in particular $\mathcal{H}$ must be minimal w.r.t. logical entailment and $\mathcal{A} \cup \mathcal{H}$ must be satisfiable. Abductive reasoning has many applications in artificial intelligence, verification and debugging, e.g. for computing missing pre-conditions, spotting and correcting errors in a logical specification, or for dealing with approximative, incomplete or spurious information. The problem has been thoroughly investigated in propositional logic [Marquis, 2000], and very efficient algorithms have been proposed [Simon and Del Val, 2001; Previti *et al.*, 2015], but only a few approaches handle more expressive logics [Knill *et al.*, 1993; Marquis, 1991; Mayer and Pirri, 1993; Nabeshima *et al.*, 2010]. In particular, none of them is able to deal with the equality predicate in an efficient way.

In the present paper, we tackle the problem of generating such a set $\mathcal{H}$, when $\mathcal{A} \cup \{\phi\}$ is a set of first-order formulas with equality and $\mathcal{H}$ is a set of ground unit clauses. In this case, by duality, $\neg\mathcal{H}$ is a clausal logical consequence of $\mathcal{A} \cup \{\neg\phi\}$, i.e., an *implicate* of $\mathcal{A} \cup \{\neg\phi\}$, and the problems boils down to efficiently generating sets of (entailment-minimal) implicates or *prime* implicates.

We illustrate our approach by an example in verification. Consider the following toy program, redirecting the tail of a nonempty list l_1 to a list l_2 of length 1. Assume we want to check that l_1 is of length 2 after the program is executed. This post-condition does not hold, which may come as a surprise for some programmers. Indeed, if $l_1 = l_2$, then the tail of l_1 is redirected to l_1 yielding a cyclic list (hence $length(l_1)$ is either 1 or undefined, depending on the definition). The problem can be stated in first-order logic with equality as the following set of clauses S, where the tail function is represented by a function $tail : \texttt{heap} \times \texttt{list} \to \texttt{list}$, and where $h, h' : \texttt{heap}$ are constant symbols denoting the initial and final states of the heap and $x, y : \texttt{list}, z : \texttt{heap}$ are universally quantified variables.

Algorithm 1: `Example`(List l_1, List l_2)

requires $length(l_1) \neq 0$;
requires $length(l_2) = 1$;
let $l_1.tail = l_2$;
ensures $length(l_1) = 2$;

% Definition of the tail redirection operation
% The tail of x is redirected to y:
$$x \simeq nil \vee tail(SetTail(z, x, y), x) \simeq y$$
% The tail of the other lists is not affected by the redirection:
$$x \simeq nil \vee x' \simeq nil \vee x' \simeq x$$
$$\vee tail(SetTail(z, x, y), x') \simeq tail(z, x')$$
% Definition of $length$
$$length(z, nil) \simeq 0$$
$$x \simeq nil \vee length(z, x) \simeq s(length(z, tail(z, x)))$$
% Pre-conditions
$$length(h, l_1) \not\simeq 0$$
$$length(h, l_2) \simeq s(0)$$
% Negation of the post-condition
$$length(h', l_1) \not\simeq s(s(0))$$
% Translation of the program
$$h' = SetTail(h, l_1, l_2)$$

The set S is satisfiable, the problem is to show that S is unsatisfiable under the hypothesis $l_1 \not\simeq l_2$, or equivalently that $l_1 \simeq l_2$ is an implicate of S.

2 Normalized Ground Clauses

Terms, atoms, literals and clauses are defined inductively as usual over a set of function symbols Σ and variables $\mathcal{V}$. We

*This paper is an extended abstract of an article in the Journal of Artificial Intelligence Research [Echenim *et al.*, 2017].

assume, w.l.o.g., that the equality $\simeq$ is the only predicate symbol. We use the symbol $\bowtie$ to denote either $\simeq$ or $\not\simeq$. Substitutions are functions mapping variables to terms, extended to any expression (terms, atoms, clauses, etc.) in the natural way and written in postfix notation. We assume given a reduction order $\succ$, i.e., an order among terms that is closed under substitution and context embedding, and contains the subterm ordering (this entails that $\prec$ is well-founded). The order $\prec$ is extended to atoms or literals by interpreting them as multisets of terms, and to clauses by the usual multiset extension. For any expression E, we denote by E^c the nnf of the negation of E, formally defined as follows: $(s \simeq t)^c = s \not\simeq t$, $(s \not\simeq t)^c = t \simeq s$, $(\bigvee_{i=1}^n l_i)^c = \bigwedge_{i=1}^n {l_i}^c$ and $(\bigwedge_{i=1}^n l_i)^c = \bigvee_{i=1}^n {l_i}^c$.

Definition 1 *Given a clause C, we define the relation $\equiv_C$ on terms as follows: for all terms s, t, $s \equiv_C t$ iff $C^c \models s \simeq t$. The* C-representatives *of a term s, literal l and clause $l_1 \vee \cdots \vee l_n$ are respectively defined by:*

$$s_{\downharpoonleft C} \stackrel{def}{=} \min_{\prec}\{s \mid t \equiv_C s\}$$
$$(s \bowtie t)_{\downharpoonleft C} \stackrel{def}{=} s_{\downharpoonleft C} \bowtie t_{\downharpoonleft C}$$
$$(l_1 \vee \cdots \vee l_n)_{\downharpoonleft C} \stackrel{def}{=} l_{1\downharpoonleft C} \vee \cdots \vee l_{n\downharpoonleft C}$$

Definition 2 *A ground clause C is* normalized *if:*

1. *every literal l in C is such that $l_{\downharpoonleft C \setminus l} = l$;*
2. *there are no two distinct positive literals l, m in C such that $m_{\downharpoonleft l^c \vee C^-}$ is a tautology;*
3. *C contains no literal of the form $t \not\simeq t$.*

A conjunction of literals $\mathcal{X}$ is normalized if the clause $\mathcal{X}^c$ is normalized.

Proposition 3 *Any ground falsifiable clause C admits a unique equivalent normalized clause $C_{\downarrow}$, which is the $\prec$-smallest clause equivalent to C. Consequently, two clauses are equivalent iff either they are both tautological or they have the same normalized form.*

3 A Constrained Superposition Calculus

This section contains the definition of the calculus for generating implicates, named c$\mathcal{SP}$. It is based on the Superposition Calculus $\mathcal{SP}$ (see for instance [Nieuwenhuis and Rubio, 2001]) which is the most successful proof procedure for first-order logic with equality. The principle underlying c$\mathcal{SP}$ consists in applying the inference rules of $\mathcal{SP}$ to the set of clauses under consideration along with ground unit clauses that are added during proof search and that act as hypotheses. To keep track of the hypotheses that were used to derive a clause, the former are attached to the clauses as constraints. This yields the following definition:

Definition 4 *A* constraint *is a conjunction (or set) of ground literals. The empty (tautological) constraint is denoted by $\top$.*

A constrained clause *(or* c-clause*) is a pair $[C \mid \mathcal{X}]$ where C is a clause and $\mathcal{X}$ is a constraint. The c-clause $[C \mid \top]$ is simply represented as C, and referred to as a standard clause.*

Intuitively $\mathcal{X}$ denotes the set of hypotheses used to derive C. If C is empty, then $[C \mid \mathcal{X}]$ is equivalent to $\mathcal{X}^c$, and $\mathcal{X}^c$ is an implicate of the clause set under consideration.

c-Superposition $(i), (ii), (iii)$	$\dfrac{[t \simeq s \vee C \mid \mathcal{X}]\ [u[t'] \bowtie v \vee D \mid \mathcal{Y}]}{[u[s] \bowtie v \vee C \vee D \mid \mathcal{X} \cup \mathcal{Y}]\sigma}$
c-Factoring $(iv), (ix)$	$\dfrac{[t \simeq u \vee t' \simeq v \vee C \mid \mathcal{X}]}{[t \simeq u \vee u \not\simeq v \vee C \mid \mathcal{X}]\sigma}$
c-Reflection $(v), (ix)$	$\dfrac{[t \not\simeq t' \vee C \mid \mathcal{X}]}{[C \mid \mathcal{X}]\sigma}$
Positive Assertion $(vi), (vii)$	$\dfrac{[u[t] \bowtie v \vee C \mid \mathcal{X}]}{[u[s] \bowtie v \vee C \mid \mathcal{X} \wedge t' \simeq s]\sigma}$
Negative Assertion $(ii), (viii), (x)$	$\dfrac{[t \simeq s \vee C \mid \mathcal{X}]}{[u[s] \bowtie v \vee C \mid \mathcal{X} \wedge u[t'] \bowtie v]\sigma}$

where the following conditions hold:

For all rules: $\sigma = \mathrm{mgu}(t, t')$;

(i): $(u[t'] \bowtie v)\sigma \in \mathrm{sel}((u[t'] \bowtie v \vee D)\sigma)$ and $v\sigma \not\succ u[t']\sigma$;

(ii): $(t \simeq s)\sigma \in \mathrm{sel}((t \simeq s \vee C)\sigma)$ and $s\sigma \not\succ t\sigma$;

(iii): $\mathcal{X} \cup \mathcal{Y} \in \mathfrak{X}$;

(iv): $(t \simeq u)\sigma \in \mathrm{sel}((C \vee t \simeq u \vee t' \simeq v)\sigma)$;

(v): $(t \simeq t')\sigma \in \mathrm{sel}((t \simeq t' \vee C)\sigma)$;

(vi): $s \prec t'$, $v\sigma \not\succ u[t]\sigma$ and $(u[t] \bowtie v)\sigma \in \mathrm{sel}((u[t] \bowtie v \vee C)\sigma)$;

(vii): $\mathcal{X} \wedge t' \simeq s \in \mathfrak{X}$;

(viii): $v \prec u[t']$;

(ix): $\mathcal{X} \in \mathfrak{X}$; and ***(x)***: $\mathcal{X} \wedge u[t'] \bowtie v \in \mathfrak{X}$.

Figure 1: Inference Rules for c$\mathcal{SP}$.

The set of inference rules defining c$\mathcal{SP}$ are given in Figure 1. The notation $u[t]$ is used to denote a term containing a subterm t (u denotes the context). We may then write $u[s]$ to denote a term obtained from $u[t]$ by replacing t by s. The symbol $\bowtie$ ($\simeq$ or $\not\simeq$) must denote identical symbols in the premise and in the conclusion of the rule. The rules are parameterized by the order $\prec$ and by:

- A selection function sel that maps every clause C to a set of selected literals in C. The set $\mathrm{sel}(C)$ must contain all $\succ$-maximal literals or at least one negative literal.
- A set of normalized constraints $\mathfrak{X}$, closed under subset. Intuitively $\mathfrak{X}$ denotes the set of *abductible formulas*, i.e., formulas that are allowed to be added as hypotheses.

The use of $\succ$ and sel is standard, it aims at pruning the search space by restricting inferences, whereas the set $\mathfrak{X}$ allows one to control the addition of hypotheses into the search space. The reader may consult [Nieuwenhuis and Rubio, 2001] for missing definitions and more details about the superposition calculus.

A crucial feature of the Superposition calculus is the availability of a general criterion for detecting redundant clauses. In $\mathcal{SP}$, a clause is considered redundant if all its instances are entailed by $\prec$-smaller instances of existing clauses. The definition for c$\mathcal{SP}$ is similar, with two differences: first the constraints of the entailing clauses must be included in that of the considered clause, and second the literals occurring in this constraint can also be used in the entailment test, provided they are smaller than the considered clause. Formally:

Definition 5 *If $\mathcal{X}$ is a constraint and C is a clause, we denote by $\mathcal{X}|_{\preceq C}$ the set of literals in $\mathcal{X}$ that are smaller than or equal to C.*

A c-clause $[C \mid \mathcal{X}]$ is redundant *w.r.t. a set of c-clauses S if either $\mathcal{X}$ is unsatisfiable or for every ground substitution σ of the variables in C, there exist c-clauses $[D_i \mid \mathcal{Y}_i] \in S$ ($1 \leq i \leq n$) and ground substitutions θ_i ($1 \leq i \leq n$) such that:*

- *$\forall i \in \{1 \dots n\}$, $C\sigma \succeq D_i\theta_i$ and $\mathcal{Y}_i \subseteq \mathcal{X}$, and*
- *$\mathcal{X}|_{\preceq C\sigma}, D_1\theta_1, \dots, D_n\theta_n \models C\sigma$.*

A set of c-clauses S is c$\mathcal{SP}$*-saturated* if every c-clause deducible by the rules of c$\mathcal{SP}$ in one step is redundant w.r.t. S. A c$\mathcal{SP}$*-saturation* of a set of c-clauses S is a set of c-clauses S^* such that: (i) every c-clause in S is redundant w.r.t. S^*, (ii) every c-clause in S^* is obtained from those in S by a finite number of applications of the rules in c$\mathcal{SP}$, (iii) S^* is c$\mathcal{SP}$-saturated.

The following theorem states the soundness and deductive completeness of c$\mathcal{SP}$.

Theorem 6 *Let S be a set of standard clauses and S^* be a c$\mathcal{SP}$-saturation of S. For every $\mathcal{X} \in \mathfrak{X}$, $\mathcal{X}^c$ is an implicate of S iff S^* contains a c-clause of the form $[\square \mid \mathcal{X}']$ with $\mathcal{X}' \subseteq \mathcal{X}$.*

4 On the Storage of Implicates

Sets of implicates are huge, thus being able to store those sets in a compact way and to detect redundant implicates efficiently is a critical feature. We begin by devising an efficient algorithm for testing entailment between ground equational clauses.

Definition 7 *Let C, D be two falsifiable ground clauses. The clause D* E-subsumes *C, written $D \leq_{\mathrm{E}} C$, iff the two following conditions hold:*

1. *for every negative literal $t \not\simeq s \in D$, $t_{\downarrow C^-} = s_{\downarrow C^-}$;*
2. *for every positive literal $l \in D$, there exists a positive literal $m \in C$ such that $m_{\downarrow C^- \vee l^c}$ is a tautology.*

Theorem 8 *Let C and D be two ground clauses. If C and D are falsifiable, then $D \models C$ iff $D \leq_{\mathrm{E}} C$.*

Based on the entailment test of Definition 7, we define a trie-like data structure to store implicates and algorithms to remove redundancy inspired from [De Kleer, 1992]. A *clausal tree* is a tree whose edges are labeled by literals and whose leaves are labeled by either $\bot$ or $\top$. Each path from the root of the tree to a leaf can be associated with a clause defined as the disjunction of all the literals labeling the edges along the path. The set of clauses associated with the tree is the set of clauses associated with a path from the root to a $\top$-leaf. This representation ensures that the prefixes of the clauses will be shared, thus reducing the amount of consumed memory. Note that the edges pointing to a $\bot$-leaf are useless and may be deleted (a node with no successor may be labeled by $\bot$).

We consider a total ordering $<_\pi$ on ground literals, defined as follows: if l_1 is a negative literal and l_2 is a positive literal then $l_1 <_\pi l_2$, and if l_1, l_2 are literals with same polarity then $l_1 <_\pi l_2$ iff $l_1 \prec l_2$. To enforce maximal sharing and minimize the size of the tree, we assume that the literals occurring along a path in the tree are ordered w.r.t. $<_\pi$, i.e., for any path $e_1, \dots, e_n$ in the tree, if $e_1, \dots, e_n$ are edges labeled by literals $l_1, \dots, l_n$ respectively, then $l_1 <_\pi \dots <_\pi l_n$.

When a new implicate C is generated, we need to test whether it is a consequence of a previously generated implicate (forward subsumption). If this is the case, then C is redundant and can be dismissed, otherwise we need to delete in the clausal tree all branches corresponding to an implicate that is a logical consequence of C (backward subsumption). We briefly sketch algorithms for performing these two tasks. The formal definitions of the algorithms can be found in [Echenim *et al.*, 2017] together with detailed proofs of their properties.

Forward Subsumption. Beside the new generated clause C and the clausal tree T, we also consider two input clauses N and M that are both initially empty. The clause N contains the literals occurring in the parent nodes in the recursive calls and the clause M permits to keep track of the negative literals of C in recursive calls after having used them a first time to rewrite literals in the tree.

The algorithm is based on a depth-first traversal of the tree which is best described as a non-deterministic recursive algorithm. The algorithm returns true if the tree contains a single node labeled by $\top$ (representing the empty clause) and backtracks if the tree is not $\top$ and $C = \square$ (base cases). Otherwise, an edge e starting from the root of the tree is chosen. Let l be the literal labeling e and let T' be the subtree pointed by e:

- If it is clear that $l \models M \vee C$, then we add l into N and we proceed to T'. This entailment condition is tested by checking that $l_{\downarrow M}$ is a contradiction (if l is negative) or that $C_{\downarrow M \vee l^c}$ contains a tautological literal (if l is positive).
- If the relation between l and $M \vee C$ is not currently determined (because l is $<_\pi$-greater than the literal currently considered in C and thus may entail literals that remain to be examined), then the minimal literal of C is added to M before restarting the exploration of the branch corresponding to e.
- Lastly, if it is clear that $l \not\models C$, which is the case e.g. when l is $\leq_\pi$-smaller than all the literals in C, then the algorithm backtracks.

Backward Subsumption. This handles the removal from a clausal tree T of all the clauses D that are E-subsumed by a given c-clause C, under the assumption that C itself is not subsumed by a clause stored in T. As the previous one, the algorithm performs a depth-first traversal of T with a rewriting of literals. The main difference is that the roles of C and T are switched: the literals of a branch of T are rewritten using the negative literals of C. When the subsumption test succeeds, the algorithm cuts the corresponding branch in T before exploring the remaining branches.

5 Experimental Results

Our prime implicate generation method has been implemented in a research prototype called `cSP`, written in OCaml, based on the `LogTk` library [Cruanes, 2014] for term ordering and congruence closure. We also implemented another version of the program, called `cSP_flat`, that only handles *flat* clauses, i.e., clauses not containing function symbols of an arity strictly greater than 0, with additional refinements that only apply to this fragment.

We compared `cSP` and `cSP_flat` with the most efficient available systems for generating implicates of logical formulas, namely `primer` [Previti *et al.*, 2015] and `Zres` [Simon and Del Val, 2001], two propositional prime implicate generation tools based on satisfiability encoding and resolution respectively, and `SOLAR` [Iwanuma *et al.*, 2009], a prime implicate generation tool for first order logic based on semantic tableaux. Equality is not built-in in these systems, thus the equality axioms had to be added into the considered formula: reflexivity, commutativity, transitivity and substitutivity. For `Zres` and `primer`, these axioms must also be grounded and transformed into propositional logic by flattening.

We compared the systems on two randomly generated sets of (flat and non-flat respectively) equational ground clauses of small size. Figure 2 compares the execution time of `cSP` on the random flat benchmark with that of `cSP_flat` and `primer`. It shows that `cSP_flat` is the most suitable tool to handle such problems.

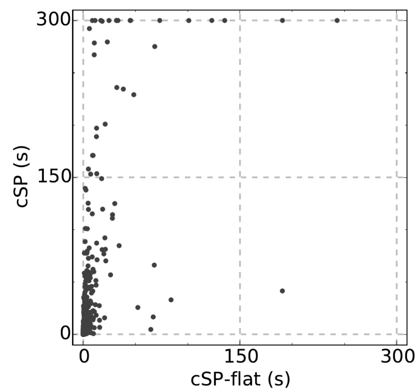
	successes	SOLAR successes				Zres successes			
		fail.	time(s)	inf.	PIs	fail.	time(s)	inf.	PIs
SOLAR	15%	0%	11.842	663190	506	74%	13.608	767160	455
Zres	52%	13%	0.695	X	2986	0%	12.474	X	13804
primer	53%	13%	0.794	X	2986	0%	3.770	X	13847
cSP_flat	63%	4%	6.622	2275	74	2%	0.500	1737	159
cSP	76%	0%	0.042	99	21	2%	3.576	755	49

	primer successes				cSP_flat successes			
	fail.	time(s)	inf.	PIs	fail.	time(s)	inf.	PIs
SOLAR	75%	13.608	767160	455	76%	12.360	694523	460
Zres	2%	12.474	X	13804	19%	7.073	X	10405
primer	0%	8.016	X	18949	17%	7.590	X	15779
cSP_flat	2%	1.480	2347	180	0%	14.290	6730	348
cSP	2%	4.296	851	49	3%	7.556	1004	62

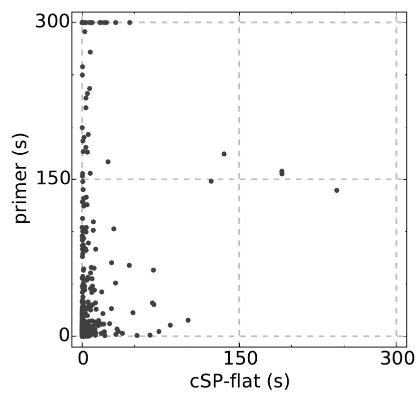
	cSP successes				timeouts	
	fail.	time(s)	inf.	PIs	inf.	PIs?
SOLAR	79%	11.842	663191	506	2452908	28152
Zres	33%	10.391	X	11338	X	X
primer	31%	7.671	X	16687	X	X
cSP_flat	19%	8.795	5418	313	X	X
cSP	0%	10.193	1209	79	14714	538

Table 1: Test results summary; random non-flat benchmark

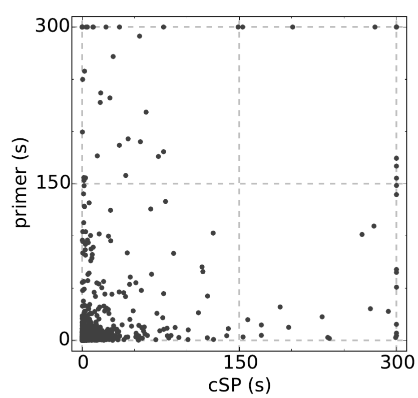
Table 1 compares `Zres`, `primer` and `SOLAR` with `cSP_flat` and `cSP` on the random non-flat benchmark. For each system, we isolate the benchmarks for which the system is able to generate all implicates, and we specify for all systems the failure rate on those benchmarks (i.e., the percentage of formulas for which the system is not capable of generating all implicates in the allocated time, namely 5 min.), the time needed to generate all prime implicates in case of success, the number of inferences and the number of implicates generated. As shown in the 'successes' column, `cSP` is the obvious winner in terms of the number of tests handled before timeout.



(a) `cSP` vs. `cSP_flat`



(b) `primer` vs. `cSP_flat`



(c) `primer` vs. `cSP`

Figure 2: Time comparison of `cSP`, `cSP_flat` and `primer`; random flat benchmark

More detailed experimental results are presented in [Echenim *et al.*, 2017].

References

[Cruanes, 2014] Simon Cruanes. Logtk: a logic toolkit for automated reasoning and its implementation. In *4th Work-*

shop on Practical Aspects of Automated Reasoning., 2014.

[De Kleer, 1992] J. De Kleer. An improved incremental algorithm for generating prime implicates. In *Proceedings of the National Conference on Artificial Intelligence*, pages 780–780. John Wiley & Sons ltd, 1992.

[Echenim *et al.*, 2017] Mnacho Echenim, Nicolas Peltier, and Sophie Tourret. Prime Implicate Generation in Equational Logic. *Journal of Artificial Intelligence Research*, 60:827–880, 2017.

[Iwanuma *et al.*, 2009] Koji Iwanuma, Hidetomo Nabeshima, and Katsumi Inoue. Toward an efficient equality computation in connection tableaux: a modification method without symmetry transformation—a preliminary report—. *Proceedings of the international workshop on First-order Theorem Proving*, 556:19, 2009.

[Knill *et al.*, 1993] E. Knill, P. T. Cox, and T. Pietrzykowski. Equality and abductive residua for Horn clauses. *Theoretical Computer Science*, 120(1):1–44, November 1993.

[Marquis, 1991] P. Marquis. Extending abduction from propositional to first-order logic. In *Proceedings of the International Workshop on Fundamentals of Artificial Intelligence Research*, pages 141–155. Springer, 1991.

[Marquis, 2000] Pierre Marquis. Consequence finding algorithms. In *Handbook of Defeasible Reasoning and Uncertainty Management Systems*, pages 41–145. Springer, 2000.

[Mayer and Pirri, 1993] Marta Cialdea Mayer and Fiora Pirri. First order abduction via tableau and sequent calculi. *Logic Journal of the IGPL*, 1(1):99–117, 1993.

[Nabeshima *et al.*, 2010] Hidetomo Nabeshima, Koji Iwanuma, Katsumi Inoue, and Oliver Ray. SOLAR: an automated deduction system for consequence finding. *AI Communications*, 23(2):183–203, 2010.

[Nieuwenhuis and Rubio, 2001] Robert Nieuwenhuis and Albert Rubio. Paramodulation-based theorem proving. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, pages 371–443. Elsevier and MIT Press, 2001.

[Previti *et al.*, 2015] A. Previti, A. Ignatiev, A. Morgado, and J. Marques-Silva. Prime compilation of non-clausal formulae. In *Proceedings of the 24th International Conference on Artificial Intelligence*, pages 1980–1987. AAAI Press, 2015.

[Simon and Del Val, 2001] L. Simon and A. Del Val. Efficient consequence finding. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence*, pages 359–370, 2001.