From Search to Computation: Redundancy Criteria and Simplification at Work

Thomas Hillenbrand[1], Ruzica Piskac[2], Uwe Waldmann[1], and Christoph Weidenbach[1]

[1] Max-Planck-Institut für Informatik
Campus E1.4, 66123 Saarbrücken, Germany

[2] École Polytechnique Fédérale de Lausanne (EPFL),
Station 14, CH-1015 Lausanne, Switzerland

Abstract. The concept of redundancy and simplification has been an ongoing theme in Harald Ganzinger's work from his first contributions to equational completion to the various variants of the superposition calculus. When executed by a theorem prover, the inference rules of logic calculi usually generate a tremendously huge search space. The redundancy and simplification concept is indispensable for cutting down this search space to a manageable size. For a number of subclasses of first-order logic appropriate redundancy and simplification concepts even turn the superposition calculus into a decision procedure. Hence, the key to successfully applying first-order theorem proving to a problem domain is to find those simplifications and redundancy criteria that fit this domain and can be effectively implemented.
We present Harald Ganzinger's work in the light of the simplification and redundancy techniques that have been developed for concrete problem areas. This includes a variant of contextual rewriting to decide a subclass of Euclidean geometry, ordered chaining techniques for Church-Rosser and priority queue proofs, contextual rewriting and history-dependent complexities for the completion of conditional rewrite systems, rewriting with equivalences for theorem proving in set theory, soft typing for the exploration of sort information in the context of equations, and constraint inheritance for automated complexity analysis.

1 Introduction

Theorem proving methods such as resolution or superposition aim at deducing a contradiction from a set of formulae by recursively deriving new formulae from given ones according to some logic calculus. A theorem prover computes one of the possible inferences of the current set of formulae and adds its conclusion to the current set, until a contradiction is found, or until a "closed" (or "saturated") set is reached, where the conclusion of every inference is already contained in the set.

Usually the inference rules of the calculus generate an infinite search space. For any serious application of saturation theorem provers, it is therefore indispensable to cut down the search space, and preferably, to turn undirected

search into goal-directed computation. Simplification and redundancy detection are the key techniques to reduce the search space of a saturation-based prover. Abstractly stated, a redundant formula is a formula that is known to be unnecessary for deriving a contradiction and can be discarded, and a simplification is a process that makes a formula redundant, possibly by adding other (simpler) formulas. To be useful in practice, however, these abstract properties have to be approximated by concrete simplifications and redundancy criteria that fit the given problem domain and can be effectively and preferably also efficiently implemented. The importance of efficiency should not be underestimated here: current theorem provers can easily spend more than 90 % of their runtime on simplification and redundancy detection.

The concept of redundancy and simplification has been an ongoing theme in Harald Ganzinger's work from his first contributions to equational completion in the mid 1980's to the various variants of the superposition calculus. We give examples of the work of Harald Ganzinger, his students, and members of his group, that illustrate simplification and redundancy techniques and their application for concrete problem areas. This includes a variant of contextual rewriting to decide a subclass of Euclidean geometry, ordered chaining techniques for Church-Rosser and priority queue proofs, contextual rewriting and history-dependent complexities for the completion of conditional rewrite systems, rewriting with equivalences for theorem proving in set theory, soft typing for the exploration of sort information in the context of equations, and constraint inheritance for automated complexity analysis.

2 Preliminaries

We start this section by briefly summarizing the foundations of first-order logic, term rewriting, and refutational theorem proving. For a more detailed presentation we refer to [2] and [9].

We consider terms and formulas over a set of function symbols Σ, a set of predicate symbols Π, and a set of variables X, where Σ, Π, and X are disjoint. Every function and predicate symbol comes with a unique arity $n \geq 0$. The set of terms over Σ and X is the least set containing x whenever $x \in X$, and containing $f(t_1, \ldots, t_n)$ whenever each t_i is a term and $f \in \Sigma$ has arity n. The set of atoms over Π, Σ, and X contains $P(t_1, \ldots, t_n)$ whenever each t_i is a term and $P \in \Pi$ has arity n. We assume that Π contains a binary predicate symbol $\approx$ (equality), written in infix notation. An atom $t \approx t'$ is also called an equation. Formulas are constructed from atoms and the constants $\top$ (true) and $\bot$ (false) using the usual connectives $\neg$, $\vee$, $\wedge$, $\Rightarrow$, $\Leftrightarrow$ and the quantifiers $\forall$ and $\exists$. Throughout this survey we assume that function and predicate symbols are declared appropriately such that all syntactic objects (terms, atoms, etc.) are well-formed.

The set of variables occurring in a syntactic object Q is denoted by $\mathrm{Var}(Q)$. If $\mathrm{Var}(Q)$ is empty, then Q is called ground. We require that there exists at least one ground term.

A literal is either an atom A (also called a positive literal) or a negated atom $\neg A$ (also called a negative literal). A clause is either the symbol $\bot$ (empty clause) or a disjunction of literals. We identify clauses with finite multiset of literals. The symbol $[\neg]\ A$ denotes either A or $\neg\ A$. Instead of $\neg\ t \approx t'$, we sometimes write $t \not\approx t'$. If no explicit quantifiers are specified, variables in a clause are implicitly universally quantified.

A substitution σ is a mapping from X into the set of terms over Σ and X. Substitutions are homomorphically extended to terms, and likewise to atoms, literals, or clauses. We use postfix notation for substitutions and write $t\sigma$ instead of $\sigma(t)$; $\sigma\sigma'$ is the substitution that maps every x to $(x\sigma)\sigma'$. A syntactic object Q' is called an instance of an object Q, if $Q\sigma = Q'$ for some substitution σ. For a set N of clauses, the set of all ground instances of clauses in N is denoted by $\bar{N}$. A substitution that maps the variables $x_1, \ldots, x_n$ to the terms $t_1, \ldots, t_n$, respectively, is denoted by $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$.

An interpretation $\mathcal{A}$ for Σ and Π consists of a non-empty set U, called the domain of $\mathcal{A}$, and a mapping that assigns to every n-ary function symbol $f \in \Sigma$ a function $f^{\mathcal{A}} : U^n \to U$ and to each n-ary predicate symbol $P \in \Pi$ an n-ary relation $P^{\mathcal{A}} \subseteq U^n$. An $\mathcal{A}$-assignment α is a mapping from the set of variables X into the domain of $\mathcal{A}$. Assignments can be homomorphically extended to terms over Σ and X. An atom $P(t_1, \ldots, t_n)$ is called true with respect to $\mathcal{A}$ and α if $(\alpha(t_1), \ldots, \alpha(t_n)) \in P^{\mathcal{A}}$, otherwise it is called false.

The extension to arbitrary formulas happens in the usual way. In particular, a negative literal $\neg\ A$ is true with respect to $\mathcal{A}$ and α if and only if A is not true, and a clause C is true with respect to $\mathcal{A}$ and α if at least one of its literals is true. An interpretation $\mathcal{A}$ is a model of a formula, if it is true with respect to $\mathcal{A}$ and α for every $\mathcal{A}$-assignment α. It is a model of a set N of formulas, if it is a model of every formula in N. If $\mathcal{A}$ is a model of N, we also say that it satisfies N. A set of formulas is called satisfiable if it has a model. Obviously every set of formulas containing $\bot$ is unsatisfiable.

In refutational theorem proving, one is primarily interested in the question whether or not a given set of universally quantified clauses is satisfiable. For this purpose we may confine ourselves to term-generated interpretations, that is, to interpretations $\mathcal{A}$ where every element of U is the image of some ground term.[3] We may even confine ourselves to Herbrand interpretations, that is, to term-generated interpretations whose domain is the set of ground terms, and where every ground term is interpreted by itself: A set of clauses has a model, if and only if it has a term-generated model, if and only if it has a Herbrand model.

As long as we restrict ourselves to term-generated models we may think of a non-ground clause as a finite representation of the set of all its ground instances: A term-generated interpretation is a model of a clause C if and only it is a model of all ground instances of C.

When one uses the equality symbol $\approx$ in a logical language, one is commonly interested in interpretations $\mathcal{A}$ in which $\approx^{\mathcal{A}}$ is not an arbitrary binary

[3] Recall that we require that there is at least one ground term.

relation but actually the equality relation on the domain of $\mathcal{A}$. We refer to such interpretations as normal. If we want to recover the intuitive semantics of the equality symbol while working with Herbrand interpretations, we have to encode the intended properties of the equality symbol explicitly using the equality axioms reflexivity, symmetry, transitivity, and congruence. If N is a set of clauses, then an interpretation that is a model of N and of the equality axioms is called an equality model of N. A set of clauses has a normal model, if and only if it has a term-generated normal model, if and only if it has an equality Herbrand model. If N and N' are sets of clauses and every equality Herbrand model of N is a model of N', we say that N entails N' modulo equality and denote this by $N \models N'$.

For simplicity, we will usually assume that equality is the only predicate symbol. This does not restrict the expressivity of the logic: A predicate P different from $\approx$ can be coded using function symbols p and *true*, so that $P(t_1, \ldots, t_n)$ is to be taken as an abbreviation for $p(t_1, \ldots, t_n) \approx \mathit{true}$. Under these circumstances, every Herbrand interpretation is completely characterized by the interpretation $\approx^{\mathcal{A}}$ of the equality predicate. For any set $E_{\mathcal{A}}$ of ground equations there is exactly one Herbrand interpretation $\mathcal{A}$ in which the equations in $E_{\mathcal{A}}$ are true and all other ground equations are false. We will usually identify $\mathcal{A}$ and $E_{\mathcal{A}}$. A positive ground literal A is thus true in $E_{\mathcal{A}}$, if $A \in E_{\mathcal{A}}$; a negative ground literal $\neg A$ is true in $E_{\mathcal{A}}$, if $A \notin E_{\mathcal{A}}$.

In the rest of the paper, we will almost exclusively work with (equality) Herbrand interpretations and models, or more precisely, with the set $E_{\mathcal{A}}$ of equations corresponding to a Herbrand interpretation $\mathcal{A}$. For simplicity, we will usually drop the attribute "Herbrand".

We describe theorem proving calculi using inference rules of the form

$$\mathcal{I}\ \frac{C_1\ \ldots\ C_n}{\begin{array}{c} D_1 \\ \vdots \\ D_m \end{array}}$$

and reduction rules of the form

$$\mathcal{R}\ \frac{C_1\ \ldots\ C_n}{\begin{array}{c} D_1 \\ \vdots \\ D_m \end{array}}$$

Both kinds of rules are used to derive new formulas $D_1, \ldots, D_m$ (conclusions) from given formulas $C_1, \ldots, C_n$ (premises) that are contained in some set N; an application of an inference rule *adds* the conclusions to N; an application of a reduction rule *replaces* the premises by the conclusions in N. (The list of conclusions can be empty; in this case the reduction rule simply deletes the premises.)

To prove the completeness of calculi, we have to construct Herbrand interpretations and to check whether a given equation is contained in such an interpretation. Rewriting techniques are our main tool for this task. The rest of this subsection serves mainly to fix the necessary notation; for more detailed information about rewrite systems we refer the reader to [2].

As usual, positions (also known as occurrences) of a term are denoted by strings of natural numbers. $t[t']_o$ is the result of the replacement of the subterm at the position o in t by t'. We write $t[t']$ if o is clear from the context.

A binary relation $\rightarrow$ is called a rewrite relation, if it is stable under substitutions and contexts, that is, if $t_1 \rightarrow t_2$ implies $t_1\sigma \rightarrow t_2\sigma$ and $s[t_1]_o \rightarrow s[t_2]_o$ for all terms t_1, t_2, and s, and for all substitutions σ.

A rewrite rule is a pair (t, t') of terms, usually written as $t \rightarrow t'$. A rewrite system is a set of rewrite rules. If R is a rewrite system, then the rewrite relation $\rightarrow_R$ associated with R is the smallest rewrite relation containing $t \rightarrow_R t'$ for every rule $t \rightarrow t' \in R$.

For a binary relation $\rightarrow$, we commonly use the symbol $\leftarrow$ for its inverse relation, $\leftrightarrow$ for its symmetric closure, $\rightarrow^+$ for its transitive closure, and $\rightarrow^*$ for its reflexive-transitive closure (and thus $\leftrightarrow^*$ for its reflexive-symmetric-transitive closure).

A binary relation $\rightarrow$ is called noetherian (or terminating), if there is no infinite chain $t_1 \rightarrow t_2 \rightarrow t_3 \rightarrow \cdots$. We say that t is a normal form (or irreducible) with respect to $\rightarrow$ if there is no t' such that $t \rightarrow t'$; t is called a normal form of s if $s \rightarrow^* t$ and t is a normal form. We say that $\rightarrow$ is confluent if for every t_0, t_1, t_2 such that $t_1 \leftarrow^* t_0 \rightarrow^* t_2$ there exists a t_3 such that $t_1 \rightarrow^* t_3 \leftarrow^* t_2$. A terminating and confluent relation is called convergent.

A transitive and irreflexive binary relation $\succ$ is called an ordering. An ordering on terms is called a reduction ordering, if it is a noetherian rewrite relation. Well-known examples of reduction orderings are polynomial orderings, lexicographic path orderings (LPO), recursicve path orderings (RPO), and Knuth-Bendix orderings (KBO) [2].

A well-founded ordering on a set S generates a well-founded ordering on finite multisets over S. We use this construction to lift a term ordering $\succ$ to a literal and a clause ordering: We assign the multiset $\{s, t\}$ to a positive literal $s \approx t$ and the multiset $\{s, s, t, t\}$ to a negative literal $s \not\approx t$. The literal ordering $\succ_L$ compares these multisets using the multiset extension of $\succ$. The clause ordering $\succ_C$ compares clauses by comparing their multisets of literals using the multiset extension of $\succ_L$. (The subscripts L and C are often omitted.)

We use the symbol $\succeq$ to denote the reflexive closure of an ordering $\succ$. If $(S_0, \succ)$ is an ordered set, $S \subseteq S_0$, and $s \in S_0$, then $S^{\prec s}$ is an abbreviation for $\{\, t \in S \mid t \prec s \,\}$.

3 CEC – Conditional Equational Completion

After having worked on compiler generation and abstract data types for about ten years, Harald Ganzinger started to work on term rewriting and completion

of algebraic specification in the mid 1980's. From 1986 to 1990, most of his work [21, 22, 20, 27, 24, 23] centered around the CEC system, serving both as a testbed to evaluate the usefulness of theoretical results and as an inspiration for further developments.

Knuth-Bendix completion [34] is a method that attempts to convert a set of equations into an equivalent convergent rewrite system. The completion procedure is based on two main operations. The first one is orientation: Equations $s \approx t$ are turned into rewrite rules $s \to t$ if s is larger than t according to some reduction ordering $\succ$. The second one is critical pair computation: If two rewrite rules $s \to t$ and $l \to r$ overlap, that is, if a non-variable subterm u of s at the position o can be unified with l, these two rewrite rules generate a new equation $(s[r]_o)\sigma \approx t\sigma$ (where σ is the most general unifier of u and l). These two main operations are supplemented by techniques to simplify (or discard) equations and rules.

The CEC ("Conditional Equational Completion") system [13], implemented in Prolog by Hubert Bertling, Harald Ganzinger and Renate Schäfers, generalizes Knuth-Bendix completion to conditional equations of the form

$$u_1 \approx v_1 \ \wedge \ \ldots \ \wedge \ u_n \approx v_n \Rightarrow s \approx t.$$

In previous approaches to completion of conditional equations, one had only considered *reductive* equations e, that is, conditional equations where the left-hand side s of the conclusion is larger than every other term occurring in e and, in particular, contains all variables of e. It is clear that this condition is quite restrictive and in fact makes most applications of conditional equations impossible. CEC, on the other side, does not require reductivity, and in fact even permits conditional equations containing extra variables, that is, variables that appear in the conditions or in the right-hand side of the conclusion, but not in the left-hand side.

One method CEC uses to deal with such conditional equations is to declare them as non-operational [20, 23]. This is the predecessor of a technique that should later become known as selection of negative literals in the superposition calculus: Instead of computing overlaps with the term s of a conditional equation $\Gamma \Rightarrow s \approx s'$, conditional rewrite rules are superposed on one selected equation of Γ, yielding new conditional equations. If the resulting conditional equations can be proved to be convergent, then the non-operational conditional equation is also convergent; it is irrelevant for the computation of normal forms (unless the specification is extended).

Moreover, CEC makes it possible to use quasi-reductive conditional equations in a Prolog-like manner. A conditional rewrite rule $u_1 \approx v_1 \ \wedge \ \ldots \ \wedge \ u_n \approx v_n \Rightarrow s \to t$ is called *quasi-reductive* [24], if it is deterministic, that is, $\mathrm{Var}(t) \subseteq \mathrm{Var}(s) \cup \bigcup_{j=1}^{n} (\mathrm{Var}(u_j) \cup \mathrm{Var}(v_j))$ and $\mathrm{Var}(u_i) \subseteq \mathrm{Var}(s) \cup \bigcup_{j=1}^{i-1} (\mathrm{Var}(u_j) \cup \mathrm{Var}(v_j))$ for every $1 \leq i \leq n$, and if $u_j\sigma \succeq v_j\sigma$ for $1 \leq j \leq i$ implies $s\sigma \succ_{\mathrm{st}} u_{i+1}\sigma$ and $u_j\sigma \succeq v_j\sigma$ for all $1 \leq j \leq n$ implies $s\sigma \succ t\sigma$.[4] Intuitively, this condition

[4] $\succ_{\mathrm{st}}$ is the transitive closure of the union of $\succ$ and the strict subterm ordering.

means that the instantiation of s yields the instantiation of u_1, normalizing any instantiated u_i and matching the result against v_i yields the instantiation of u_{i+1}, and normalizing every u_i yields the instantiation of t.

CEC also implements rewriting and completion modulo AC to deal with associative and commutative operators. An interesting feature from the user perspective is the ability to specify the term ordering incrementally during the completion process.

In contrast to an automated theorem prover, a completion procedure like CEC may fail if it encounters a conditional equation that can neither be oriented nor discarded. Powerful techniques for simplifying critical pair peaks are therefore extremely important for a successful completion procedure, and CEC contains a large repertoire of such techniques. A conditional equation $\Gamma \Rightarrow s \approx s'$ can be eliminated if there exists a proof of $\Gamma \Rightarrow s \approx s'$ that is simpler than $\Gamma \Rightarrow s \approx s'$ itself. Rewriting in contexts is a method to simplify conditional equations, where skolemized oriented condition equations are employed to reduce terms in the conclusion. CEC also uses non-reductive equations $\Gamma \Rightarrow s \to s'$ for simplification: if $\Gamma\sigma$ is a subset of Δ, then $s\sigma \to s'\sigma$ is available to simplify $\Delta \Rightarrow t \approx t'$. Finally, CEC makes it possible to make the complexity of a conditional equation history-dependent: complexities of input formulas can be arbitrarily chosen, whereas the origin of a derived conditional equation determines the complexity of the latter [21].

The completeness proof for the procedure implemented in CEC extends the proof ordering technique of Leo Bachmair, Nachum Dershowitz and Jieh Hsiang [4]: while Bachmair, Dershowitz and Hsiang use linear proofs and define an ordering on them as the multiset extension of the ordering of proof steps, one needs now tree-like proofs, represented as proof terms, which are compared using an RPO with an ordering on proof operators as precedence. As in [4], completion inferences lead to smaller proof terms.

CEC has been used, for instance, for the correctness proof for a code generator (Giegerich [29]) and for semi-functional translation of modal logics (Ohlbach [43]). It is able to deal with order-sorted specifications [24], including specifications with non-sort-decreasing rules, for which the procedure of Gnaedig, Kirchner and Kirchner [30] fails. Other examples include the specification of a maximum function over a total ordering, including the transitivity axiom [23].

4 Saturate

Among all techniques developed to deal with equality in first-order theorem proving, the paramodulation calculus of George Robinson and Larry Wos [47] has been the most influential. The paramodulation rule embodies the ideas of the resolution calculus and the operation of "replacing equals by equals" that is fundamental for term rewriting. Whenever a clause contains a positive literal $t \approx t'$, the paramodulation rule permits to rewrite a subterm t occurring in some literal $[\neg]\ A[t]$ of another clause to t'. For non-ground clauses, equality is replaced by unifiability, so that the resulting rule is essentially a combination of

non-ground resolution and Knuth-Bendix completion.[5]

$$\mathcal{I}\ \frac{D' \vee t \approx t' \qquad\qquad C' \vee [\neg]\, s[w] \approx s'}{(D' \vee C' \vee [\neg]\, s[t'] \approx s')\sigma}$$

where σ is a most general unifier of t and w.[6]

Both resolution and completion are (or can be) subject to ordering restrictions with respect to some syntactical ordering $\succ$ on atoms or terms: In the Knuth-Bendix completion procedure,[7] only overlaps at non-variable positions between the maximal sides of two rewrite rules produce a critical pair. Similarly, the resolution calculus remains a semidecision procedure if inferences are computed only if each of the two complementary literals is maximal in its premise. It is natural to ask whether paramodulation may inherit the ordering restrictions of both its ancestors. More precisely: Let a paramodulation inference between clauses $D = D' \vee t \approx t'$ and $C = C' \vee [\neg]\, s[w] \approx s'$ be given as above, and let $\succ$ be a reduction ordering that is total on ground terms. Does the calculus remain refutationally complete if we require, as in completion, that (i) w is not a variable, (ii) $t\sigma \not\preceq t'\sigma$, (iii) $(s[w])\sigma \not\preceq s'\sigma$, and, as in ordered resolution, that (iv) $(t \approx t')\sigma$ is maximal in $D\sigma$, and (v) $(s[w] \approx s')\sigma$ is maximal in $C\sigma$?

A first result in this direction was obtained by Gerald Peterson [44], who showed the admissibility of restrictions (i) and (ii). It was extended to (i), (ii), (iii) for positive literals, and (v) by Michaël Rusinowitch [48], and to (i), (ii), (iv), and (v) by Jieh Hsiang and Michaël Rusinowitch [31]. The final answer was given by Leo Bachmair and Harald Ganzinger [5–7]: All five restrictions may be imposed on the paramodulation rule (which is named *superposition* then), however, an additional inference rule becomes necessary to cope with certain non-Horn clauses: either the *merging paramodulation* rule, which appeared first in (Bachmair and Ganzinger [5, 6]), or the *equality factoring* rule, which is due to Robert Nieuwenhuis [36]. The resulting inference system is the basis of the superposition calculus; it consists of the rules *superposition*, *equality resolution* (i. e., ordered resolution with the reflexivity axiom), and either *equality factoring* or *ordered factoring* and *merging paramodulation*.

The "model construction" technique developed by Bachmair and Ganzinger to prove the refutational completeness of superposition is based on an earlier idea by Zhang and Kapur [56]. Let N be saturated and let $\bar{N}$ be the set of all ground instances of clauses in N. We inspect all clauses in $\bar{N}$ in ascending order and construct a sequence of interpretations, starting with the empty interpretation. If a clause $C \in \bar{N}$ is false in the current interpretation $\mathcal{I}_C$ generated by clauses smaller than C and has a positive and strictly maximal literal A, and if some

[5] Essentially the same rule (usually restricted to equational unit or Horn clauses) occurs in narrowing calculi (Fay [18]) used for theory unification.

[6] We use the letters $\mathcal{I}$ and $\mathcal{R}$ to distinguish between *inference rules*, whose premises are kept after the conclusions have been added to the given set of clauses, and *reduction rules*, whose premises are replaced by the conclusions.

[7] Or rather: in its unfailing variant (Bachmair [3]), which is a semidecision procedure for unit equational logic.

additional conditions are satisfied, then a new interpretation is created extending the current one in such a way that A becomes true. We say that the clause is productive. Otherwise, the current interpretation is left unchanged. One can then show that, if N is saturated and does not contain the empty clause $\bot$, then every clause C is either true in $\mathcal{I}_C$ or productive, so that every clause of N becomes true in the limit interpretation $\mathcal{I}_N$ (also known as the perfect model of N).

The model construction method is the foundation of general notion of redundancy [7]. Essentially, every ground formula C that is true in $\mathcal{I}_C$ is useless to show that a set of formulas is not saturated. We call such formulas *weakly redundant*. Unfortunately, a formula that is weakly redundant in a set of formulas N may lose this property if we compute inferences from formulas in N and add the conclusions to N. For this reason, it is usually better to work with (strong) redundancy: Let $\bar{N}^{\prec C}$ be the set of all formulas in $\bar{N}$ that are smaller than C. We say that the formula C is *(strongly) redundant* with respect to N if $\bar{N}^{\prec C} \models C$, and that an inference with conclusion C is redundant with respect to N if $\bar{N}^{\prec D} \models C$, where D is the maximal premise of the inference.[8] Non-ground formulas and inferences are called redundant, if all their ground instances are redundant. Usual strategies for resolution-like calculi such as tautology deletion or clause subsumption are encompassed by this definition, just as the simplification steps and critical-pair criteria [3] that can be found in completion procedures. Superposition can also be enhanced by selection functions, so that hyperresolution-like strategies become applicable.

The SATURATE system [26] has been the first superposition-based theorem prover. The implementation was originally started by Robert Nieuwenhuis and Pilar Nivela and later continued by Harald Ganzinger. Written in Prolog, it lacks the inference speed of later superposition provers, such as E, SPASS, or VAMPIRE; it is still remarkable, though, for the huge number of calculi it uses, such as constraint superpostion, chaining, and lazy CNF transformation, and the sophisticated redundancy checks and simplification techniques enabled in this way. In the rest of this section we present four concrete applications of these simplification techniques.

4.1 Automatic Complexity Analysis

The automated complexity analysis technique of David Basin and Harald Ganzinger [12] is based on the concept of order locality. A set of clauses (without equality) is called local with respect to a term ordering $\succ$ if, for every ground clause C, $N \models C$ implies that there is a proof of C from those instances of N in which every term is smaller than or equal to some term of C. As a special case, defining $\succ$ as the subterm ordering yields the notion of subterm locality that had been previously investigated by David McAllester [35].

[8] Note that "redundancy" is called "compositeness" in [7]. In later papers the standard terminology has changed.

If the ordering $\succ$ has the property that for every ground term there are only finitely many smaller terms, then locality with respect to $\succ$ implies complexity bounds for the decision problem $N \models C$. More precisely, if for every clause of size n there exist $O(f(n))$ terms that are smaller than or equal to some term in the clause and that can be enumerated in time $g(n)$, and if a set N of Horn clauses is local with respect to $\succ$, then $N \models C$ is decidable in time $O(f(m)^k + g(m))$, where m is the size of C. The constant k depends only on N, it is at most the maximum of the number of variables of each clause in N. For instance, one obtains polynomial complexity bounds if one takes the subterm ordering as $\succ$; a Knuth-Bendix ordering yields exponential bounds and a polynomial ordering doubly exponential bounds.

Order locality is closely related to saturation with respect to ordered resolution: If N is a saturated set of clauses with respect to an atom ordering $\succ'$, then N is local with respect to some term ordering $\succ$, provided that certain compatibility requirements for $\succ$ and $\succ'$ are satisfied.

The SATURATE system [26] has been used both to prove saturation (and hence locality) and to transform sets of clauses into equivalent saturated and local sets. Formulas like the transitivity axiom show up rather frequently in such clause sets, and for such clauses, inheritance of ordering constraints is useful to show saturation. The technique is due to Nivela and Nieuwenhuis [41]: In contrast to a normal clause C, which can be taken as a representative of all its ground instances, a constrained clause $\Theta \parallel C$ represents only those ground instances of C that satisfy the constraint Θ, where Θ may be a conjunction of ordering and equality literals between terms or atoms. In particular, a clause whose constraint Θ is unsatisfiable is redundant, since it does not represent any ground instance.

Both the ordering restrictions of an inference and the constraints of its premises are propagated to its conclusion, so we obtain inference rules like the following for constraint resolution:

$$\mathcal{I}\ \frac{\Theta_1 \parallel D \vee A \qquad\qquad \Theta_2 \parallel C \vee \neg B}{\Theta_1 \wedge \Theta_2 \wedge A = B \wedge \Theta \parallel D \vee C}$$

Here, Θ_1 and Θ_2 are the constraints of the premises which are propagated to the conclusion, $A = B$ is the equality constraint of the inference, and Θ is the ordering constraint of the inference stating that the literals A and B are (strictly) maximal in their respective clauses.

Examples of theories that have been successfully saturated using the SATURATE system are the congruence closure axioms, the theory of tree embedding, and the theory of partial orderings.

4.2 Church–Rosser Theorems for the λ-Calculus

The λ-calculus, originally conceived by Alonzo Church and Stephen Kleene [33, 16] around 1935, is a model of computability that is based on the notions of function definition, function application, and recursion. It operates on λ-terms, which are the closure of a given set of variables $x, y, \ldots$ under application $(t_1\ t_2)$

and abstraction $\lambda x.t$. A notion of variable substitution is defined recursively over the term structure. Since free variables in a substituted expression must not be bound after substitution, renaming of variables may become necessary. The result of replacing x in t by s this way is denoted by $t[s/x]$.

The calculus comes with conversion rules which capture when two λ-terms denote the same function: α-conversion models that the actual names of bound variables do not matter; β-conversion $(\lambda x.t)s \leftrightarrow_\beta t[s/x]$ corresponds to function application; and η-conversion covers extensional equality of functions: $\lambda x.(tx) \leftrightarrow_\eta t$ unless x is free in t. In order to ease the management of variables when manipulating λ-terms, Nicolaas Govert de Bruijn [15] suggested to consider, instead of $x, y, \ldots$, natural numbers in a fixed order, thereby getting rid of names altogether, and of α-conversion as well. The remaining conversions, if applied in left-to-right direction only, constitute reduction systems. A key property of the λ-calculus is that β-reduction $\rightarrow_\beta$ enjoys a Church–Rosser theorem: Any two $\leftrightarrow_\beta^*$-convertible terms are $\rightarrow_\beta^*$-reducible to a common successor. The same applies to $\rightarrow_\eta$, and to the union of the two reduction systems.

Tobias Nipkow [39] formalized this family of Church–Rosser theorems within the Isabelle/HOL system [40], which is a proof assistant for higher-order logic. Though interactive, Isabelle also features automation of subproofs via a term rewriting engine and a tableaux prover for predicate logic. Nipkow reported that the success of the latter depended on the right selection of lemmas supplied as parameters. For arithmetic goals arising from de Bruijn indices, he added a special tactic based on Fourier–Motzkin elimination. The proof development followed the lines of [15, Chapter 3], with an excursus to the approach of [51] via parallel reductions.

Each of the propositions that Nipkow showed with Isabelle/HOL encapsulates a single induction or is already deductive, at least modulo the arithmetic reasoning in the background; and in the former case the induction scheme was explicitly given. Therefore the question whether these propositions could be demonstrated automatically with a first-order theorem prover constituted a real challenge, and would set a landmark for the applicability of such systems if answered in the affirmative. This is what Harald Ganzinger and his student Sebastian Winkel set out for at the end of the 1990's. Their key idea was to integrate a fragment of arithmetic into the first-order axiomatization itself. They used a Peano-style formulation, postulated a total ordering, and related the latter to the successor and to the predecessor operation. Such theories fall into the domain of the chaining calculus [8], which specializes resolution and superposition for transitive resolution, and which is implemented in the SATURATE system [26]. Notably SATURATE managed to prove all the propositions, within the scope of some minutes.

Just to give an impression of the kind of reasoning in this domain, the first-order axiomatization on top of the approximation of numbers is shown now. A variable with de Bruijn number i is denoted by $\mathrm{var}(i)$; furthermore $\mathrm{abs}(s)$ denotes an abstraction, and $\mathrm{app}(s, t)$ an application. In formalizing substitutions, a function $\mathrm{lift}(t, i)$ is needed that increments all free variables in t that are greater

than i or equal to i. SATURATE is supplied with a case-split definition according to the structure of t. This definition corresponds to the usual recursive one, but its semantics for SATURATE is purely first-order. All variables are universally quantified.

$$\begin{aligned}
&\neg(i < k) \vee \mathrm{lift}(\mathrm{var}(i), k) \approx \mathrm{var}(i)\\
&\neg(k \leq i) \vee \mathrm{lift}(\mathrm{var}(i), k) \approx \mathrm{var}(s(i))\\
&\qquad\qquad \mathrm{lift}(\mathrm{app}(s,t), k) \approx \mathrm{app}(\mathrm{lift}(s,k), \mathrm{lift}(t,k))\\
&\qquad\qquad \mathrm{lift}(\mathrm{abs}(s), k) \approx \mathrm{abs}(\mathrm{lift}(s, \mathrm{succ}(k)))
\end{aligned}$$

In a similar fashion, SATURATE is provided with a definition of $\mathrm{subst}(t,s,k)$, which shall amount to $t[s/k]$. Note that within $t[s/k]$, all free variables of t above k are decremented, for application within β-reduction and η-reduction.

$$\begin{aligned}
&\neg(k < i) \vee \mathrm{subst}(\mathrm{var}(i), s, k) \approx \mathrm{var}(\mathrm{pred}(i))\\
&\qquad\qquad \mathrm{subst}(\mathrm{var}(k), s, k) \approx s\\
&\neg(i < k) \vee \mathrm{subst}(\mathrm{var}(i), s, k) \approx \mathrm{var}(i)\\
&\qquad\qquad \mathrm{subst}(\mathrm{app}(t,u), s, k) \approx \mathrm{app}(\mathrm{subst}(t,s,k), \mathrm{subst}(u,s,k))\\
&\qquad\qquad \mathrm{subst}(\mathrm{abs}(t), s, k) \approx \mathrm{abs}(\mathrm{subst}(t, \mathrm{lift}(s,0), \mathrm{succ}(k)))
\end{aligned}$$

One of the inductive propositions that Nipkow proved in Isabelle about substitution is the identity $t[i/i] = t[i/i+1]$. As to SATURATE, Harald Ganzinger and Sebastian Winkel first introduced a predicate for the induction hypothesis:

$$\mathrm{P}(t,i) \equiv \neg(0 \leq i) \vee \mathrm{subst}(t, \mathrm{var}(i), i) \approx \mathrm{subst}(t, \mathrm{var}(i), \mathrm{succ}(i))$$

Then SATURATE was able to discharge the conjunction of the following proof obligations in a single run, which correspond to the base case respectively the two step cases of the induction:

$$\begin{aligned}
&\mathrm{P}(\mathrm{var}(j), i)\\
\neg\mathrm{P}(t,k) &\vee \mathrm{P}(\mathrm{abs}(t), i)\\
\neg\mathrm{P}(t,i) \vee \neg\mathrm{P}(u,i) &\vee \mathrm{P}(\mathrm{app}(t,u), i)
\end{aligned}$$

The following clauses correspond to the base case respectively the two step cases of the induction:

$$\begin{aligned}
&\mathrm{P}(\mathrm{var}(j), i)\\
\neg\mathrm{P}(t,k) &\vee \mathrm{P}(\mathrm{abs}(t), i)\\
\neg\mathrm{P}(t,i) \vee \neg\mathrm{P}(u,i) &\vee \mathrm{P}(\mathrm{app}(t,u), i)
\end{aligned}$$

In the end SATURATE discharged the conjunction of these proof obligations in a single run, retaining no more than 57 clauses as non-redundant. The standard parameter setting was employed.

Transitivity and other ordering axioms play a vital role in the problem description. Transitive relations are known to be detrimental to the efficiency of standard theorem provers. SATURATE contains an implementation of the chaining calculus (Bachmair and Ganzinger [8]) with makes it possible to avoid explicit inferences with transitivity axioms.

Given a transitive relation R and a well-founded ordering $\succ$ on ground terms and literals, the chaining calculus has the following inference rules:

$$\mathcal{I}\ \frac{C \vee R(s,t) \qquad D \vee R(u,v)}{(C \vee D \vee R(s,v))\sigma}$$

$$\mathcal{I}\ \frac{C \vee R(t,s) \qquad D \vee \neg R(u,v)}{(C \vee D \vee \neg R(s,v))\sigma}$$

$$\mathcal{I}\ \frac{C \vee R(s,t) \qquad D \vee \neg R(v,u)}{(C \vee D \vee \neg R(s,v))\sigma}$$

where, for all rules, σ is an mgu of t and u. Moreover, the chaining rules are equipped with ordering restrictions similar to the superposition calculus; in particular the inferences only need to be performed if positive R-literals are strictly maximal in the respective premises, negative R-literals are maximal in the respective premises, and $s\sigma \not\succeq t\sigma$ and $v\sigma \not\succeq u\sigma$.

In order to assess the merits of the chaining calculus for this proof problem, one may want to compare the 57 clauses kept by SATURATE with the corresponding number for a theorem prover like SPASS that implements the standard superposition calculus without chaining. With default setting and the same reduction ordering, SPASS has to develop 947 non-redundant clauses until a proof is found. If splitting is turned off, then this number reduces to 342. Seemingly the presence of the two intertwined transitive relations $<$ and $\leq$ is a menace to SPASS. If the axiomatization is rephrased in terms of $<$ only, then with a set-of-support strategy and increased variable weight one gets down to 184 clauses.

Via `http://isabelle.in.tum.de/dist/library/HOL/Lambda` the proof development within Isabelle is available. The SATURATE distribution can be obtained from `http://www.mpi-inf.mpg.de/SATURATE/Saturate.html` and contains all the mentioned first-order proof formulations.

4.3 Lazy CNF Transformation

Practically all automated theorem provers in use today are based on clausal logic. The input is preprocessed to obtain clause normal form (CNF), this includes the replacement of equivalences by conjunctions of implications and the elimination of existential quantifiers by Skolemization.

Very often it is useful to already exploit properties at the formula level via appropriate deduction mechanisms. Consider the following example: Suppose we have an equivalence $P \Leftrightarrow (Q \wedge Q')$, where P, Q, Q' are propositional formulas. Translation to CNF yields the three clauses $P \Rightarrow Q$, $P \Rightarrow Q'$, $Q \wedge Q' \Rightarrow P$. If $P \succ Q, Q'$ and C is a clause $R \vee R' \vee P$, then the two resolution steps deriving $R \vee R' \vee Q$ and $R \vee R' \vee Q'$ from C constitute a simplification of C: C follows from the three smaller clauses $Q \wedge Q' \Rightarrow P$, $R \vee R' \vee Q$ and $R \vee R' \vee Q'$. This fact is somewhat hidden within the set of clauses, though, whereas it was rather obvious considering the original equivalence. The situation is even worse if one side of the equivalence contains additional quantified variables, which are skolemized

away during the clausification of one of the two directions of the equivalence. For example,

$$\forall A.\forall B.(A \subseteq B \;\Leftrightarrow\; \forall x.(x \in A \;\Rightarrow\; x \in B))$$

is turned into

$$\begin{gathered}\neg\, A \subseteq B \;\vee\; \neg\, x \in A \;\vee\; x \in B \\ f(A,B) \in A \;\vee\; A \subseteq B \\ \neg\, f(A,B) \in B \;\vee\; A \subseteq B.\end{gathered}$$

To overcome this problem, Harald Ganzinger and Jürgen Stuber [28] introduced a variant of the superposition calculus with equivalences between formulas and lazy quantifier elimination. Its positive superposition rule

$$\mathcal{I}\;\frac{D \;\vee\; u \approx v \qquad\qquad C \;\vee\; s[u'] \approx t}{(D \;\vee\; C \;\vee\; s[v] \approx t)\sigma}$$

where σ is a most general unifier of u and u' is applicable both to term equations and to equivalences, that is, equations between formulas. This allows reasoning with equivalences as they usually arise from definitions in a natural way: If the larger side of an equivalence is an atomic formula, it can be used in a positive superposition; such an inference is a simplification for instance if D is empty and $u'\sigma = u$. If the larger side of an equivalence $u \approx v$ is not atomic, the equivalence can be eliminated using rules like

$$\mathcal{R}\;\frac{C \;\vee\; u \approx v}{C \;\vee\; u \approx \bot \;\vee\; v \approx \top} \qquad\qquad \mathcal{R}\;\frac{C \;\vee\; u \approx v}{C \;\vee\; u \approx \top \;\vee\; v \approx \bot}$$

whose results are then simplified by tableau-like expansion rules such as

$$\mathcal{R}\;\frac{C \;\vee\; (u_1 \wedge u_2 \approx \top)}{C \;\vee\; u_1 \approx \top} \qquad\qquad \mathcal{R}\;\frac{C \;\vee\; (u_1 \wedge u_2 \approx \top)}{C \;\vee\; u_2 \approx \top}.$$

Bound variables are encoded by de Bruijn indices, so a formula $\exists x \forall y\, f(x,y) \approx y$ is written as $(\exists\, \forall\, (p(2,1) \approx 1)) \approx \top$, and quantified formulas are handled by γ and δ expansion rules

$$\mathcal{R}\;\frac{C \;\vee\; (\exists\, u \approx \top)}{C \;\vee\; u(f(x_1,\ldots,x_n)) \approx \top}$$

where $x_1,\ldots,x_n$ are the free variables in u and f is a fresh Skolem function, and

$$\mathcal{R}\;\frac{C \;\vee\; (\forall\, u \approx \top)}{C \;\vee\; u(z) \approx \top}$$

where z is a fresh variable. Since de Bruijn indices may be replaced by arbitrarily large terms, they must have greater precedence than all other function symbols in the recursive path ordering used to compare terms and formulas.

The calculus is implemented in the SATURATE system [26]. In applications like set theory, that are dominated by complex definitions, the number of inferences that SATURATE performs can be several orders of magnitude smaller

compared to more conventional provers, such as VAMPIRE, or E. Motivated by the experiments with SATURATE, in SPASS a definition detection and expansion algorithm has been integrated that can simulate the above reasoning for many practical cases [1]. The fact that many equivalence transformations are now simplifications reduces the search space significantly; for certain examples the derivation of SATURATE is completely deterministic and terminates after few steps whereas other provers do not find a solution within any reasonable time limit.

4.4 Priority Queues

Runtime result checking is a method to ensure software reliability that has been proposed by Hal Wasserman and Manuel Blum [52]. In this approach, a checker program runs in parallel to the program to be checked and monitors its inputs and outputs. The checker either confirms correctness of the program's output or reports an error. It does not verify the correctness of the program, though; in fact it does not look into the program code at all. It only verifies that the output was correct on a given input.

A priority queue is a data structure that maintains a set of real numbers under the operations *insert_element* and *delete_and_return_minimal_element*. It can be implemented in such a way that each operation needs logarithmic time. A checker for priority queues has been developed by Ulrich Finkler and Kurt Mehlhorn [19]. It runs in parallel to the original priority queue algorithm and associates a lower bound with every member of the priority queue. The lower bound of an element e is defined as the maximum of all values that the priority queue returned as minimal since e was inserted. In the case that the priority queue would return a non-minimal element, the lower bound of the current minimal element will be greater than the element itself. When this element will be retrieved, the checker will report an error. The checker is time-efficient, but an off-line checker; this means that, when the priority queue is incorrectly implemented, i.e. returns a non-minimal element, this error will not be noticed immediately, but only at the moment when one of the smaller elements is returned in a later step.

A formal correctness proof for Finkler's and Mehlhorn's priority queue checker has been given by Ruzica Piskac using the SATURATE system. She showed that, if during the run of the priority queue the checker does not report any error until the queue is empty, then all returned minimal elements are correct [45]. The verification was done in two stages: in the first stage the correctness of the algorithm used for the checker was proved, while in the second stage a framework following more closely the concrete implementation and data structures was developed (de Nivelle and Piskac [42]).

The problem description defining the behavior of priority queues and of the checker contained more than 50 formulas (cf. Figure 1). In order to find the proof, SATURATE needed some additional lemmas (which again needed to be proved by the theorem prover, sometimes making further lemmas necessary). At the end more than 80 lemmas were used for the complete proof.

quasi-ordered set with bottom element:

$p_1 \leq p_2 \wedge p_2 \leq p_3 \Rightarrow p_1 \leq p_3$.

$p_1 \leq p_2 \vee p_2 \leq p_1$.

$p \leq p$.

$\text{bottom} \leq p$.

$(p_1 < p_2) \Leftrightarrow (p_1 \leq p_2 \wedge \neg p_2 \leq p_1)$.

priority queues:

$\neg\text{contains\_pq}(\text{create\_pq}, p)$.

$\text{contains\_pq}(\text{insert\_pq}(pq, p_1), p_2) \Leftrightarrow (\text{contains\_pq}(pq, p_2) \vee p_1 \approx p_2)$.

$\text{remove\_pq}(\text{insert\_pq}(pq, p), p) \approx pq$.

$\neg p_1 \approx p_2 \wedge \text{contains\_pq}(pq, p_2)$
$\Rightarrow \text{remove\_pq}(\text{insert\_pq}(pq, p_1), p_2) \approx \text{insert\_pq}(\text{remove\_pq}(pq, p_2), p_1)$.

$\text{contains\_pq}(pq, p) \wedge (\forall p_1.\text{contains\_pq}(pq, p_1) \Rightarrow p \leq p_1)$
$\Rightarrow \text{find\_min\_pq}(pq, p) \approx p$.

$\text{contains\_pq}(pq, p) \wedge (\forall p_1.\text{contains\_pq}(pq, p_1) \Rightarrow p \leq p_1)$
$\Rightarrow \text{remove\_min\_pq}(pq, p) \approx \text{remove\_pq}(pq, p)$.

lower bounds:

$\neg\text{contains\_s}(\text{create\_s}, p)$.

$\text{contains\_s}(\text{assign\_s}(s, \text{pair}(p_1, r)), p_2)$
$\Leftrightarrow (\text{contains\_s}(s, p_2) \vee p_1 \approx p_2)$.

$\neg\text{pair\_in\_s}(\text{create\_s}, p, r)$.

$\text{pair\_in\_s}(\text{assign\_s}(s, \text{pair}(p_1, r_1)), p_2, r_2)$
$\Leftrightarrow (\text{pair\_in\_s}(s, p_2, r_2) \vee (p_1 \approx p_2 \wedge r_1 \approx r_2))$.

$\text{remove\_s}(\text{assign\_s}(s, \text{pair}(p, r)), p) \approx s$.

$\neg p_1 \approx p_2 \wedge \text{contains\_s}(s, p_2)$
$\Rightarrow \text{remove\_s}(\text{assign\_s}(s, \text{pair}(p_1, r)), p_2)$
$\approx \text{assign\_s}(\text{remove\_s}(s, p_2), \text{pair}(p_1, r))$.

$\text{lookup\_s}(\text{assign\_s}(s, \text{pair}(p, r)), p) \approx r$.

$\neg p_1 \approx p_2 \wedge \text{contains\_s}(s, p_2)$
$\Rightarrow \text{lookup\_s}(\text{assign\_s}(s, \text{pair}(p_1, r)), p_2)$
$\approx \text{lookup\_s}(s, p_2)$.

$\text{update\_s}(\text{create\_s}, p) \approx \text{create\_s}$.

$r < p_2$
$\Rightarrow \text{update\_s}(\text{assign\_s}(s, \text{pair}(p_1, r)), p_2)$
$\approx \text{assign\_s}(\text{update\_s}(s, p_2), \text{pair}(p_1, p_2))$.

$p_2 \leq r$
$\Rightarrow \text{update\_s}(\text{assign\_s}(s, \text{pair}(p_1, r)), p_2)$
$\approx \text{assign\_s}(\text{update\_s}(s, p_2), \text{pair}(p_1, r))$.

Fig. 1. Excerpt of the problem description.

Six of the formulas to be proved make heavy use of ordering axioms, and, as in Sect. 4.2, the chaining inference rule was crucial for the success of SATURATE for these formulas. Being implemented in Prolog, SATURATE is in general much slower than provers like SPASS, VAMPIRE, or E. In those cases, however, where the chaining rule makes it possible to avoid explicit inferences with the transitivity axiom, SATURATE can be orders of magnitude faster. We have repeated the experiments with the same lemmas using SPASS, version 3.0, where chaining is not implemented. The running times of both theorem provers are shown in Figure 2.

benchmark	SATURATE	SPASS
lemma_not_min_elem_not_check	00:07.98	03:31.46
lemma_not_ok_persistence	00:03.24	–
lemma_contains_s_I_remove	00:06.50	–
remove_min_02_1	00:03.55	03:08.31
tmp_not_check_02	00:02.92	–
tmp_not_check_03	00:04.50	59:21.52

Fig. 2. Table shows time in format min:sec that SATURATE and SPASS spent on the problem. The symbol "–" indicates that a prover did not terminate after two hours of running.

All the experiments with SATURATE were done using the standard settings. It includes tautology deletion, forward subsumption, forward and backward reduction, and simplification by totality resolution.

5 Spass

The development of the theorem prover SPASS started in 1994 (Weidenbach et al. [55, 54]). Using memory-efficient data structures and specific indexing techniques, SPASS has been the first high speed implementation of the superposition calculus, followed by E [49] and VAMPIRE [46]. SPASS also features an advanced CNF transformation module, equivalence-based definition extraction and expansion technology, a large collection of simplification methods, a special treatment of monadic predicates ("sorts"), and a tableau-like splitting rule for dealing with clauses that can be written as disjunctions of variable-disjoint subclauses.

5.1 Euclidean Geometry

Philippe Balbiani [10, 11] introduced a convergent and terminating conditional term rewriting system for a subtheory of Euclidean geometry. Lacking a powerful general proof procedure, Balbiani developed a Prolog-based proof procedure just in order to establish the properties of this term rewriting system. Christof Brinker and Christoph Weidenbach showed that SPASS plus a specific form of

contextual rewriting can also be used to produce a complete system for the Balbiani rule set [14].

The conditional equations in Figure 3 formalize properties of Euclidean geometry on the basis of straight lines, indicated by the letter d, and points, indicated by the letter p. For convenience, uppercase variable letters represent points and lowercase variable letters straight lines (sets of points). Then $d(X, Y)$ formalizes the line through X and Y and in case $X = Y$ an arbitrary but fixed line through X, $f_{dp}(x, X)$ the line parallel to x through X, $f_{pd}(X, x)$ the projection from X to x, $f_{dd}(x, y)$ the perpendicular in the intersection of x and y and, in case x and y are parallel, an arbitrary but fixed perpendicular on y, and $p(X, x)$ the perpendicular from X on x.

$$
\begin{aligned}
\mathsf{d}(\mathsf{X},\mathsf{Y}) &\approx \mathsf{d}(\mathsf{Y},\mathsf{X}) && (\mathsf{RGO}_0)\\
\mathsf{f_{dp}}(\mathsf{x},\mathsf{f_{pd}}(\mathsf{X},\mathsf{x})) &\approx \mathsf{x} && (\mathsf{RGO}_1)\\
\mathsf{f_{pd}}(\mathsf{X},\mathsf{f_{dp}}(\mathsf{x},\mathsf{X})) &\approx \mathsf{X} && (\mathsf{RGO}_2)\\
\mathsf{f_{dd}}(\mathsf{y},\mathsf{f_{dd}}(\mathsf{x},\mathsf{y})) &\approx \mathsf{y} && (\mathsf{RGO}_3)\\
\mathsf{f_{dp}}(\mathsf{p}(\mathsf{X},\mathsf{x}),\mathsf{X}) &\approx \mathsf{p}(\mathsf{X},\mathsf{x}) && (\mathsf{RGO}_4)\\
\mathsf{f_{dd}}(\mathsf{p}(\mathsf{X},\mathsf{x}),\mathsf{x}) &\approx \mathsf{p}(\mathsf{X},\mathsf{x}) && (\mathsf{RGO}_5)\\
\mathsf{f_{pd}}(\mathsf{X},\mathsf{d}(\mathsf{X},\mathsf{Y})) &\approx \mathsf{X} && (\mathsf{RGO}_6)\\
\mathsf{p}(\mathsf{f_{pd}}(\mathsf{X},\mathsf{x}),\mathsf{f_{dd}}(\mathsf{y},\mathsf{x})) &\approx \mathsf{x} && (\mathsf{RGO}_7)\\
\\
\mathsf{f_{pd}}(\mathsf{f_{pd}}(\mathsf{X},\mathsf{x}),\mathsf{x}) &\approx \mathsf{f_{pd}}(\mathsf{X},\mathsf{x}) && (\mathsf{RGO}_8)\\
\mathsf{f_{pd}}(\mathsf{X},\mathsf{p}(\mathsf{X},\mathsf{x})) &\approx \mathsf{X} && (\mathsf{RGO}_9)\\
\mathsf{f_{dp}}(\mathsf{f_{dp}}(\mathsf{x},\mathsf{X}),\mathsf{X}) &\approx \mathsf{f_{dp}}(\mathsf{x},\mathsf{X}) && (\mathsf{RGO}_{10})\\
\mathsf{f_{dp}}(\mathsf{d}(\mathsf{X},\mathsf{Y}),\mathsf{X}) &\approx \mathsf{d}(\mathsf{X},\mathsf{Y}) && (\mathsf{RGO}_{11})\\
\mathsf{f_{dd}}(\mathsf{f_{dd}}(\mathsf{x},\mathsf{y}),\mathsf{y}) &\approx \mathsf{f_{dd}}(\mathsf{x},\mathsf{y}) && (\mathsf{RGO}_{12})\\
\mathsf{f_{dd}}(\mathsf{x},\mathsf{p}(\mathsf{X},\mathsf{x})) &\approx \mathsf{x} && (\mathsf{RGO}_{13})\\
\mathsf{p}(\mathsf{X},\mathsf{f_{dd}}(\mathsf{x},\mathsf{f_{dp}}(\mathsf{y},\mathsf{X}))) &\approx \mathsf{f_{dp}}(\mathsf{y},\mathsf{X}) && (\mathsf{RGO}_{14})\\
\mathsf{p}(\mathsf{X},\mathsf{f_{dd}}(\mathsf{x},\mathsf{p}(\mathsf{X},\mathsf{y}))) &\approx \mathsf{p}(\mathsf{X},\mathsf{y}) && (\mathsf{RGO}_{15})\\
\mathsf{p}(\mathsf{X},\mathsf{f_{dd}}(\mathsf{x},\mathsf{d}(\mathsf{X},\mathsf{Y}))) &\approx \mathsf{d}(\mathsf{X},\mathsf{Y}) && (\mathsf{RGO}_{16})\\
\mathsf{p}(\mathsf{f_{pd}}(\mathsf{X},\mathsf{f_{dd}}(\mathsf{x},\mathsf{y})),\mathsf{y}) &\approx \mathsf{f_{dd}}(\mathsf{x},\mathsf{y}) && (\mathsf{RGO}_{17})\\
\mathsf{p}(\mathsf{f_{pd}}(\mathsf{X},\mathsf{p}(\mathsf{Y},\mathsf{x})),\mathsf{x}) &\approx \mathsf{p}(\mathsf{Y},\mathsf{x}) && (\mathsf{RGO}_{18})\\
\mathsf{p}(\mathsf{f_{pd}}(\mathsf{X},\mathsf{x}),\mathsf{p}(\mathsf{Y},\mathsf{x})) &\approx \mathsf{x} && (\mathsf{RGO}_{19})\\
\mathsf{p}(\mathsf{X},\mathsf{p}(\mathsf{Y},\mathsf{f_{dp}}(\mathsf{x},\mathsf{X}))) &\approx \mathsf{f_{dp}}(\mathsf{x},\mathsf{X}) && (\mathsf{RGO}_{20})\\
\mathsf{p}(\mathsf{X},\mathsf{p}(\mathsf{Y},\mathsf{p}(\mathsf{X},\mathsf{x}))) &\approx \mathsf{p}(\mathsf{X},\mathsf{x}) && (\mathsf{RGO}_{21})\\
\mathsf{p}(\mathsf{X},\mathsf{p}(\mathsf{Y},\mathsf{d}(\mathsf{X},\mathsf{Z}))) &\approx \mathsf{d}(\mathsf{X},\mathsf{Z}) && (\mathsf{RGO}_{22})\\
\\
\mathsf{f_{pd}}(\mathsf{X},\mathsf{x}) \not\approx \mathsf{f_{pd}}(\mathsf{Y},\mathsf{x}) \Rightarrow\ \mathsf{d}(\mathsf{f_{pd}}(\mathsf{X},\mathsf{x}),\mathsf{f_{pd}}(\mathsf{Y},\mathsf{x})) &\approx \mathsf{x} && (\mathsf{RGO}_{23})\\
\\
\mathsf{X} \not\approx \mathsf{f_{pd}}(\mathsf{Y},\mathsf{f_{dp}}(\mathsf{x},\mathsf{X})) \Rightarrow \mathsf{d}(\mathsf{X},\mathsf{f_{pd}}(\mathsf{Y},\mathsf{f_{dp}}(\mathsf{x},\mathsf{X}))) &\approx \mathsf{f_{dp}}(\mathsf{x},\mathsf{X}) && (\mathsf{RGO}_{24})\\
\mathsf{X} \not\approx \mathsf{f_{pd}}(\mathsf{Y},\mathsf{p}(\mathsf{X},\mathsf{x})) \Rightarrow\ \mathsf{d}(\mathsf{X},\mathsf{f_{pd}}(\mathsf{Y},\mathsf{p}(\mathsf{X},\mathsf{x}))) &\approx \mathsf{p}(\mathsf{X},\mathsf{x}) && (\mathsf{RGO}_{25})\\
\mathsf{X} \not\approx \mathsf{f_{pd}}(\mathsf{Y},\mathsf{d}(\mathsf{X},\mathsf{Z})) \Rightarrow\ \mathsf{d}(\mathsf{X},\mathsf{f_{pd}}(\mathsf{Y},\mathsf{d}(\mathsf{X},\mathsf{Z}))) &\approx \mathsf{d}(\mathsf{X},\mathsf{Z}) && (\mathsf{RGO}_{26})
\end{aligned}
$$

Fig. 3. Euclidean Conditional Rewrite System

In its general form, *Contextual Rewriting* is the reduction rule between a clauses C and D from a clause set N given below. The term $N_C\tau$ for a substi-

tution τ grounding C denotes the set of all ground clauses generated from N by instantiating variables with ground terms from the range of τ that are smaller than $C\tau$.

Let $C = C' \vee s \approx t$, $D = D' \vee [\neg]\, u[s']_p \approx v$ be two clauses in N. The reduction

$$\mathcal{R}\ \frac{C' \vee s \approx t \qquad\qquad D' \vee [\neg]\, u[s'] \approx v}{\begin{array}{c} C' \vee s \approx t \\ D' \vee [\neg]\, u[t\sigma]_p \approx v \end{array}}$$

where (i) $s\sigma = s'$, (ii) $s\sigma \succ t\sigma$, (iii) $D \succ C\sigma$, (iv) τ is a Skolem substitution replacing the variables in $C\sigma$ and D by new Skolem constants, (v) $N_C\tau \models D''\tau \vee \neg A\tau$ for all negative equations A in $C'\sigma$ where D'' are the negative equations in D', and (vi) $N_C\tau \models \neg A\tau \vee D'''\tau$ for all positive equations A in $C'\sigma$ where D''' are the positive equations in D' is called *contextual rewriting*.

This general form of contextual rewriting can be effectively computed, but is very expensive. For example, given the Skolem constants for $C\sigma$ and D there are exponentially many possibilities in the number of variables to instantiate a clause from N_C by these constants. Furthermore, Harald Ganzinger implemented the rule in the SATURATE system [41] and his experiments showed that the complexity shows up in practice. There were examples where the prover spent hours on the applicability of a single contextual rewriting application of the above form.

A detailed study of Balbiani's conditional rewrite system and proof procedure yielded that considering the context $N_C\tau$ is not necessary for termination of the saturation process. It is sufficient to study contextual rewriting with respect to the involved clauses and standard reduction of the generated clauses $D''\tau \vee \neg A\tau$ and $\neg A\tau \vee D'''\tau$. The result is the local contextual rewriting rule.

Let $C = C' \vee s \approx t$, $D = D' \vee [\neg]\, u[s']_p \approx v$ be two clauses in N. The reduction

$$\mathcal{R}\ \frac{C' \vee s \approx t \qquad\qquad D' \vee [\neg]\, u[s'] \approx v}{\begin{array}{c} C' \vee s \approx t \\ D' \vee [\neg]\, u[t\sigma]_p \approx v \end{array}}$$

where (i) $s\sigma = s'$, (ii) $s\sigma \succ t\sigma$, (iii) $D \succ C\sigma$, (iv) τ is a Skolem substitution replacing the variables in $C\sigma$ and D by new Skolem constants, (v) $\models D''\tau \vee \neg A\tau$ for all negative equations A in $C'\sigma$ where D'' are the negative equations in D', and (v) $\models \neg A\tau \vee D'''\tau$ for all positive equations A in $C'\sigma$ where D''' are the positive equations in D' is called *Local Contextual Rewriting*.

The applicability of local contextual rewriting can be decided in polynomial time[9], because the semantic tautology checks for $\models D''\tau \vee \neg A\tau$ and $\models \neg A\tau \vee D'''\tau$ can be decided by the well-known congruence closure algorithm [17].

We get the *Semantic Tautology Rule*

$$\mathcal{R}\ \frac{C}{}$$

[9] If the used ordering $\prec$ is decidable in polynomial time.

if $\models C$ for free, because we need to decide semantic tautologies for the applicability of local contextual rewriting anyway.

Now with these two extra rules local contextual rewriting and semantic tautology deletion the Balbiani system can be finitely saturated. SPASS needs less than one second to generate the saturated system consisting of 40 conditional equations. Except for the commutativity of $d(X, Y)$, all clauses have a single maximal oriented equation containing all variables in the left-hand side of the equation. Therefore, the saturated system can be effectively used to decide any ground query, i.e., universally quantified conjecture.

5.2 Soft Typing

The general redundancy notion of superposition is given with respect to the perfect, minimal (Herbrand) model $\mathcal{I}_N$, generated by a (saturated) clause set N. Any clause C that is true in the perfect model generated by all clauses of the actual clause set N smaller than C, written $\mathcal{I}_C \models C$, is weakly redundant (see Section 4). One consequence of this result is that actually any clause C that is implied by smaller clauses from N can be deleted, i.e., if $N^{\prec C} \models C$ then C can be deleted. This is the foundation for most of all practically used redundancy and simplification notions, e.g., rewriting or subsumption.

The model-based redundancy notion has two problems in practice. First, it is dynamic. As long as the set N is not saturated and new clauses are derived the interpretation $\mathcal{I}_N$ changes and therefore, clauses must not be deleted but can only be blocked for inferences (see Section 4). Second, the properties $\mathcal{I}_N \models C$ and $\mathcal{I}_C \models C$ are undecidable in general, because they constitute a second-order property by considering validity in a minimal model of a set of first-order clauses N.

One solution to this problem is to define an upper approximation $\mathcal{I}'_N$ of $\mathcal{I}_N$ that is (i) stable under inferences in N and for which (ii) the problem $\mathcal{I}'_N \models C$ becomes decidable. An (Herbrand) interpretation $\mathcal{I}'_N$ is an upper approximation of $\mathcal{I}_N$, written $\mathcal{I}_N \subseteq \mathcal{I}'_N$, if for all predicates P: $P^{\mathcal{I}_N} \subseteq P^{\mathcal{I}'_N}$ and the two interpretations agree on the interpretation of all function symbols. Then such an approximation can be used to simplify reasoning on N. A first application is the detection and deletion of redundant clauses. Consider a clause $\neg A_1 \vee \ldots \vee \neg A_n \vee C$ out of a clause set N such that for any grounding substitution σ the atoms $A_i\sigma$ are false: $\mathcal{I}'_N \not\models A_1\sigma \wedge \ldots \wedge A_n\sigma$. Then the clause $\neg A_1 \vee \ldots \vee \neg A_n \vee C$ is a tautology and can be deleted. This technique is called *soft typing*. There are applications where soft typing is key to finitely saturate a clause set [25].

In order to obtain effectively an upper approximation $\mathcal{I}'_N$ the idea is to actually approximate the clause set N by a (consistent) clause set N' such that eventually $\mathcal{I}_N \subseteq \mathcal{I}_{N'}$ and N' belongs to a decidable clause class. This way, a second application is to prove properties of N by considering N'. If Φ is a universally closed conjunction of atoms and $N' \models \Phi$, then $N \models \Phi$. Thus, if Φ is provable from N', which is decidable by construction, then we need not to consider validity with respect to N, which is undecidable, in general.

So we need to find an expressive, decidable sublanguage of first-order clause logic that can serve as the range class for an approximation. Monadic Horn clause sets are a natural and powerful candidate [37, 32, 53]. There exist several decidability results and we will show in this section that there also exist natural and powerful approximations into the class. Monadic Horn clause classes are typically used to describe sorts or types and serve as a theoretical basis in other contexts like programming languages or abstract interpretation, supporting the name *soft typing*.

The theoretical background for the application of approximation functions given below was developed by Harald Ganzinger, Christoph Meyer, and Christoph Weidenbach [25]. It was implemented by Enno Keen via the DFG2DFG tool, part of the SPASS distribution since 2001. All rules can be applied separately but exhaustively and can be actually composed to obtain different overall approximations. Note that application of the rules may turn a consistent clause set into an inconsistent one due to the upper approximation of predicates. So checking consistency of the approximated clause set is mandatory for the approach to work.

The *Horn Rule* transforms a non-Horn clause into a set of Horn clauses:

$$\mathcal{R}\ \frac{C \vee A_1 \vee \ldots \vee A_n}{\begin{array}{c} C \vee A_1 \\ \vdots \\ C \vee A_n \end{array}}$$

where $n \geq 2$ and $A_1 \vee \ldots \vee A_n$ are equality or non-equality atoms and no more positive atoms are in C.

The next two rules constitute alternative transformations from non-monadic non-equality literals into monadic literals. Note that equality literals are not transformed.

The *Monadic Term Encoding Rule* transforms an n-ary predicate into a monadic atom using by moving the predicate to the function level.

$$\mathcal{R}\ \frac{C \vee [\neg]P(t_1, \ldots, t_n)}{C \vee [\neg]T(p(t_1, \ldots, t_n))}$$

where $n \geq 2$, p is a new function corresponding to the predicate P and T is a special fixed predicate. Applied to a given clause set, all occurrences of P in the clause set are transformed into the same function p.

The *Monadic Projection Encoding Rule* transforms an n-ary predicate into several monadic atoms by argument projection:

$$\mathcal{R}\ \frac{C \vee [\neg]P(t_1, \ldots, t_n)}{\begin{array}{c} C \vee [\neg]P_1(t_1) \\ \vdots \\ C \vee [\neg]P_n(t_n) \end{array}}$$

where $n \geq 2$, and $P_1, \ldots, P_n$ are new monadic predicates. All occurrences of P in the clause set are transformed into the same predicates $P_1, \ldots, P_n$.

So far a combination of the rules enables the transformation from an arbitrary clause set into a monadic Horn clause set. From now on we assume all clause sets to be Horn and monadic. The following rules approximate a monadic Horn clause set into a monadic Horn clause set with a decidable entailment problem by relaxing the term structure. There are several candidates for such clause sets, relying on linearity and shallowness. A term is called *linear* if it contains no repeated variables. A term is called *shallow* if it is of the form $f(x_1, \ldots, x_n)$. The *Linear Approximation Rule* given below reduces the number of non-linear variable occurrences in a Horn clause by replacing a variable x repeated within an atom by some new variable x'. Note that the transformation is not applicable to clauses containing equality or non-monadic literals.

The *Linear Approximation Rule* eliminates non-linear variable occurrences in atoms of monadic Horn clauses:

$$\mathcal{R}\ \frac{C \vee P(t)[x]_{p,q}}{C\{x \mapsto x'\} \vee C \vee P(t)[x'/x]_p}$$

where $p \neq q$, and x' is a new variable.

Finally, nested terms are transformer into shallow terms by the *Shallow Approximation Rule*

$$\mathcal{R}\ \frac{C \vee P(t[s]_p)}{\begin{array}{c}\neg S(x) \vee C \vee P(t[x/s]_p) \\ C \vee S(s)\end{array}}$$

where s is a complex term at non-top position p in t, and x is a new variable and S a new predicate.

The rule can be further refined by considering all occurrences of s in t simultaneously and by filtering C with respect to variable dependencies with s.

The transformation rules Horn Transformation, Monadic Projection Encoding, Linear Approximation, Shallow and Relaxed Shallow produce upper approximations of the original clause set.

Eventually the rules can be combined to obtain a decidable approximation for a given clause set N. A typical sequence is the transformation to Horn clauses, transformation to monadic literals, linear transformation, and finally shallow transformation resulting in an approximation Horn clause set N' in the above sense. In practice, the challenge is to find approximations that lead to consistent and non-trivial approximations.

6 Conclusions

Simplification and redundancy detection are the key techniques to reduce the search space of a theorem prover. Harald Ganzinger has developed the fundamental abstract concept of redundancy and simplification for superposition-like

calculi together with Leo Bachmair. His theoretical work, however, has always been supplemented by the urge to make it practically useful – by developing concrete, effective redundancy and simplification criteria to be implemented in current theorem provers in order to make them beneficial for various application domains. In this survey, we have tried to give a few representative examples of this practical side of his scientific work.

References

1. Bijan Afshordel, Thomas Hillenbrand, and Christoph Weidenbach. First-order atom definitions extended. In Robert Nieuwenhuis and Andrei Voronkov, eds., *Logic for Programming, Artificial Intelligence, and Reasoning, LPAR 2001*, 2001, *LNAI 2250*, pp. 309–319. Springer-Verlag.
2. Franz Baader and Tobias Nipkow. *Term rewriting and all that.* Cambridge University Press, New York, NY, USA, 1998.
3. Leo Bachmair. *Canonical Equational Proofs.* Birkhäuser, Boston, MA, USA, 1991.
4. Leo Bachmair, Nachum Dershowitz, and Jieh Hsiang. Orderings for equational proofs. In *[First Annual] Symposium on Logic in Computer Science*, Cambridge, Massachusetts, USA, June 16–18, 1986, pp. 346–357. IEEE Computer Society Press.
5. Leo Bachmair and Harald Ganzinger. Completion of first-order clauses with equality by strict superposition (extended abstract). In Stéphane Kaplan and Mitsuhiro Okada, eds., *Conditional and Typed Rewriting Systems, 2nd International Workshop*, Montreal, Canada, June 11–14, 1990, LNCS 516, pp. 162–180. Springer-Verlag.
6. Leo Bachmair and Harald Ganzinger. On restrictions of ordered paramodulation with simplification. In Mark E. Stickel, ed., *10th International Conference on Automated Deduction*, Kaiserslautern, Germany, July 24–27, 1990, LNAI 449, pp. 427–441. Springer-Verlag.
7. Leo Bachmair and Harald Ganzinger. Rewrite-based equational theorem proving with selection and simplification. *Journal of Logic and Computation*, 4(3):217–247, 1994.
8. Leo Bachmair and Harald Ganzinger. Ordered chaining calculi for first-order theories of transitive relations. *Journal of the ACM*, 45(6):1007–1049, 1998.
9. Leo Bachmair and Harald Ganzinger. Resolution theorem proving. In Alan Robinson and Andrei Voronkov, eds., *Handbook of Automated Reasoning*, vol. 1, ch. 2, pp. 19–99. Elsevier, 2001.
10. Philippe Balbiani. Equation solving in geometrical theories. In Nachum Dershowitz and Naomi Lindenstrauss, eds., *Proceedings of the 4th International Workshop on Conditional and Typed Rewriting Systems*, 1995, *LNCS 968*. Springer-Verlag.
11. Philippe Balbiani. Mécanisation de la géométrie : incidence et orthogonalité. *Revue d'intelligence artificielle*, 11:179–211, 1997.
12. David Basin and Harald Ganzinger. Automated complexity analysis based on ordered resolution. *Journal of the ACM*, 48(1):70–109, 2001.
13. Hubert Bertling, Harald Ganzinger, and Renate Schäfers. CEC: A system for the completion of conditional equational specifications. In Harald Ganzinger, ed., *ESOP '88, 2nd European Symposium on Programming*, Nancy, France, 1988, *LNCS 300*, pp. 378–379. Springer-Verlag.

14. Christof Brinker. Geometrisches Schließen mit SPASS. Diplomarbeit, Universität des Saarlandes and Max-Planck-Institut für Informatik, Saarbrücken, Germany, 2000. Supervisors: H. Ganzinger, C. Weidenbach.
15. Nicolaas Govert de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church–Rosser theorem. *Indagationes Mathematicae*, 34(5):381–392, 1972.
16. Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58:345–363, 1936.
17. Peter J. Downey, Ravi Sethi, and Robert Endre Tarjan. Variations on the common subexpression problem. *Journal of the ACM*, 27(4):758–771, 1980.
18. M. Fay. First-order unification in an equational theory. In *Fourth Workshop on Automated Deduction*, Austin, TX, USA, February 1979, pp. 161–167. Academic Press.
19. Ulrich Finkler and Kurt Mehlhorn. Checking priority queues. In *Proceedings of the 10th annual ACM-SIAM symposium on Discrete algorithms (SODA'99)*, 1999, pp. 901–902. Society for Industrial and Applied Mathematics.
20. Harald Ganzinger. A completion procedure for conditional equations. In Stéphane Kaplan and Jean-Pierre Jouannaud, eds., *Conditional Term Rewrite Systems*, Orsay, France, July 1987, *LNCS 308*, pp. 62–83. Springer-Verlag.
21. Harald Ganzinger. Completion with history-dependent complexities for generated equations. In Donald Sannella and Andrzej Tarlecki, eds., *Recent Trends in Data Type Specification, 5th Workshop on Abstract Data Types*, Gullane, Scotland, UK, 1987, *LNCS 332*, pp. 73–91. Springer-Verlag.
22. Harald Ganzinger. Ground term confluence in parametric conditional equational specifications. In Franz J. Brandenburg, Guy Vidal-Naquet, and Martin Wirsing, eds., *4th Annual Symposium on Theoretical Aspects of Computer Science*, Passau, Federal Republic of Germany, February 19–21, 1987, *LNCS 247*, pp. 286–298. Springer-Verlag.
23. Harald Ganzinger. A completion procedure for conditional equations. *Journal of Symbolic Computation*, 11:51–81, 1991.
24. Harald Ganzinger. Order-sorted completion: the many-sorted way. *Theoretical Computer Science*, 89:3–32, 1991.
25. Harald Ganzinger, Christoph Meyer, and Christoph Weidenbach. Soft typing for ordered resolution. In *Proceedings of the 14th International Conference on Automated Deduction, CADE-14*, Townsville, Australia, 1997, *LNAI 1249*, pp. 321–335. Springer-Verlag.
26. Harald Ganzinger, Robert Nieuwenhuis, and Pilar Nivela. The Saturate system, 1994. System available from `http://www.mpi-sb.mpg.de/SATURATE`.
27. Harald Ganzinger and Renate Schäfers. System support for modular order-sorted horn clause specifications. In *12th International Conference on Software Engineering*, Nice, France, 1990, pp. 150–159. IEEE Computer Society Press.
28. Harald Ganzinger and Jürgen Stuber. Superposition with equivalence reasoning and delayed clause normal form transformation. *Information and Computation*, 199:3–23, 2005.
29. Robert Giegerich. Specification and correctness of code generators – an experiment with the CEC-system. In Jürgen Müller and Harald Ganzinger, eds., *1st German Workshop "Term Rewriting: Theory and Applications"*, 1989, SEKI-Report 89/02. Universität Kaiserslautern.
30. Isabelle Gnaedig, Claude Kirchner, and Hélène Kirchner. Equational completion in order-sorted algebras. *Theoretical Computer Science*, 72(2&3):169–202, May 1990.

31. Jieh Hsiang and Michaël Rusinowitch. Proving refutational completeness of theorem-proving strategies: The transfinite semantic tree method. *Journal of the ACM*, 38(3):559–587, July 1991.
32. Florent Jacquemard, Christoph Meyer, and Christoph Weidenbach. Unification in extensions of shallow equational theories. In Tobias Nipkow, ed., *Rewriting Techniques and Applications, 9th International Conference, RTA-98*, 1998, *LNCS 1379*, pp. 76–90. Springer-Verlag.
33. Stephen Kleene. A theory of positive integers in formal logic. *American Journal of Mathematics*, 57:153–173 and 219–244, 1935.
34. Donald E. Knuth and Peter B. Bendix. Simple word problems in universal algebras. In J. Leech, ed., *Computational Problems in Abstract Algebra*, pp. 263–297. Pergamon Press, Oxford, United Kingdom, 1970. Reprinted in Siekmann and Wrightson [50], pp. 342–376.
35. David A. McAllester. Automatic recognition of tractability in inference relation. *Journal of the ACM*, 40(2):284–303, April 1993.
36. Robert Nieuwenhuis. First-order completion techniques. Technical report, UPC-LSI, 1991. Cited in Nieuwenhuis and Rubio [38].
37. Robert Nieuwenhuis. Basic paramodulation and decidable theories (extended abstract). In *Proceedings 11th IEEE Symposium on Logic in Computer Science, LICS'96*, 1996, pp. 473–482. IEEE Computer Society Press.
38. Robert Nieuwenhuis and Albert Rubio. Basic superposition is complete. In Bernd Krieg-Brückner, ed., *ESOP'92, 4th European Symposium on Programming*, Rennes, France, February 26–28, 1992, LNCS 582, pp. 371–389. Springer-Verlag.
39. Tobias Nipkow. More Church–Rosser proofs (in Isabelle/HOL). *Journal of Automated Reasoning*, 26:51–66, 2001.
40. Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, *LNCS 2283*. Springer-Verlag, 2002.
41. Pilar Nivela and Robert Nieuwenhuis. Saturation of first-order (constrained) clauses with the *Saturate* system. In Claude Kirchner, ed., *Rewriting Techniques and Applications, 5th International Conference, RTA-93*, Montreal, Canada, June 16–18, 1993, *Lecture Notes in Computer Science, LNCS*, vol. 690, pp. 436–440. Springer-Verlag.
42. Hans de Nivelle and Ruzica Piskac. Verification of an off-line checker for priority queues. In Bernhard K. Aichernig and Bernhard Beckert, eds., *Third IEEE International Conference on Software Engineering and Formal Methods (SEFM 2005)*, Koblenz, 2005, pp. 210–219. IEEE.
43. Hans Jürgen Ohlbach. Translation methods for non-classical logics – an overview. *Bulletin of the IGPL*, 1(1):69–90, 1993.
44. Gerald E. Peterson. A technique for establishing completeness results in theorem proving with equality. *SIAM Journal on Computing*, 12(1):82–100, February 1983.
45. Ruzica Piskac. Formal correctness of result checking for priority queues. Master's thesis, Universität des Saarlandes, February 2005.
46. Alexandre Riazanov and Andrei Voronkov. The design and implementation of vampire. *AI Communications*, 15(2-3):91–110, 2002.
47. G[eorge] Robinson and L[arry] Wos. Paramodulation and theorem-proving in first-order theories with equality. In Bernard Meltzer and Donald Michie, eds., *Machine Intelligence 4*, ch. 8, pp. 135–150. Edinburgh University Press, Edinburgh, United Kingdom, 1969. Reprinted in Siekmann and Wrightson [50], pp. 298–313.
48. Michaël Rusinowitch. Theorem-proving with resolution and superposition. *Journal of Symbolic Computation*, 11(1&2):21–49, January/February 1991.

49. Stephan Schulz. E – A Brainiac Theorem Prover. *Journal of AI Communications*, 15(2/3):111–126, 2002.
50. Jörg Siekmann and Graham Wrightson, eds. *Automation of Reasoning*, vol. 2: Classical Papers on Computational Logic 1967–1970. Springer-Verlag, Berlin, Germany, 1983.
51. Masako Takahashi. Parallel reductions in λ-calculus. *Information and Computation*, 118(1):120–127, 1995.
52. Hal Wasserman and Manuel Blum. Software reliability via run-time result-checking. *Journal of the ACM*, 44(6):826–849, 1997.
53. Christoph Weidenbach. Towards an automatic analysis of security protocols in first-order logic. In Harald Ganzinger, ed., *16th International Conference on Automated Deduction, CADE-16*, 1999, *LNAI 1632*, pp. 378–382. Springer-Verlag.
54. Christoph Weidenbach, Uwe Brahm, Thomas Hillenbrand, Enno Keen, Christian Theobalt, and Dalibor Topic. SPASS version 2.0. In Andrei Voronkov, ed., *Automated Deduction – CADE-18*, July 27-30 2002, *LNCS 2392*, pp. 273–277. Springer-Verlag.
55. Christoph Weidenbach, Bernd Gaede, and Georg Rock. SPASS & FLOTTER, version 0.42. In Michael A. McRobbie and John K. Slaney, eds., *Automated Deduction – CADE-13, 13th International Conference on Automated Deduction*, New Brunswick, NJ, USA, July 30–August 3, 1996, LNAI 1104, pp. 141–145. Springer-Verlag.
56. Hantao Zhang and Deepak Kapur. First-order theorem proving using conditional rewrite rules. In Ewing Lusk and Ross Overbeek, eds., *9th International Conference on Automated Deduction*, Argonne, Illinois, USA, May 23–26, 1988, LNCS 310, pp. 1–20. Springer-Verlag.