The Theory of SPASS Version 2.0

Christoph Weidenbach

Max-Planck-Institut für Informatik
Stuhlsatzenhausweg 85
66123 Saarbrücken, Germany.
weidenb@mpi-sb.mpg.de
http://www.mpi-sb.mpg.de/~weidenb

Contents

1. What This Article is (not) About

This article is about the design, the implementation and the use of SPASS Version 2.0 [Weidenbach, Afshordel, Brahm, Cohrs, Engel, Keen, Theobalt and Topic 1999], a saturation-based automated theorem prover for first-order logic with equality. SPASS is unique due to the combination of the superposition calculus with specific inference/reduction rules for sorts (types) and a splitting rule for case analysis motivated by the β-rule of analytic tableau and the case analysis employed in the Davis-Putnam procedure [Davis and Putnam 1960]. Furthermore, SPASS provides a sophisticated clause normal form translation [Nonnengart, Rock and Weidenbach 1998, Nonnengart and Weidenbach 2001]. This chapter is not about completeness/soundness proofs for saturation-based first-order logic calculi. For this we refer to Bachmair and Ganzinger [2001], Nieuwenhuis and Rubio [2001] and the corresponding references in this article. Nevertheless, this documentation introduces a variety of inference/reduction rules that are implemented in SPASS and that form a basis for various first-order calculi.

At the heart of SPASS is a first-order calculus. It consists of inference rules that generate new clauses and reduction rules that reduce the number of clauses or transform clauses into simpler ones. In SPASS we introduced a great variety of clause set[1] based inference and reduction rules that can be composed to various sound and complete first-order calculi. The clause store data structure together with such a calculus are the basis for most of today's theorem proving systems, like Otter, E or Vampire(see Appendix C). SPASS goes one step further by introducing a splitting rule that supports explicit case analysis. This generalizes the standard clause store based approach to a clause store collection[2] approach where different clause stores represent the different cases. Therefore, the splitting rule introduces a second dimension in saturation-based automated theorem proving.

The third dimension we consider in SPASS are constraints, extra information attached to a clause restricting its semantics and/or usage with respect to the calculus. Well-known constraints are ordering constraints, forcing substituted terms to satisfy the attached ordering restrictions, basicness constraints, forbidding paramodulation inferences on certain terms in the clause, or type constraints guaranteeing that instantiations for variables conform to the attached type of the variable. From an abstract implementation point of view the handling of constraints is always the same. The information is attached to a clause, it is maintained during the inference/reduction application process and it is exploited by constraint specific algorithms/deduction mechanisms to restrict inferences/reductions or to even eventually delete a clause. We implemented sort constraints, specific type constraints for variables where the type (sort) theory is itself expressed by clauses.

A software project like SPASS is always a compromise between different goals like maintainability, efficiency, flexibility, readability, short development time, modularity, etc. For SPASS, the most important goals are maintainability, flexibility, readability and modularity of the design (code). This does not mean that SPASS is inefficient, but whenever there is a conflict between efficiency and, e.g., a modular design, we prefer the latter. Best evidence that SPASS really meets its design goals is the fact that its code is used by several

[1] From an implementation point of view we consider clause multisets, called *clause stores*.
[2] A *clause store collection* is a multiset of clause stores.

research groups as a basis for code development and that the project has at the time of this writing run successfully for eight years. We also view this prover as a tool(box), so even for users that don't want to spend effort in implementation work it offers great flexibility, rich documentation and a number of indispensable extra tools like syntax translators or a proof checker.

We believe that a sophisticated calculus, a "good theory", has the highest impact on the performance of a prover. Therefore, we won't study the implementation of provers at the level of data structures, object hierarchies or module design. Instead, we will discuss the needs for an efficient implementation of the various inference/reduction rules and the impacts that the top-level search algorithms have on an actual implementation. This together with a specific design goal decision can then lead to a design concept for a real prover like SPASS.

Heuristics are also not in focus of SPASS, although they can play an important rôle in automated theorem proving. For example, the heuristic that chooses the next clause for inferences inside a typical "main loop" of a saturation based prover (see Table 1 on page 11) can have a great impact on the success/non-success of a search attempt for a proof. However, it is the nature of heuristics that they are sometimes useful and sometimes make things even worse. In the context of automated theorem proving, it is often not predictable what will be the case as long as we don't restrict our attention to specific problems (problem classes). Therefore again, the main focus of SPASS is on inference/simplification/reduction techniques. For these techniques we know, e.g., that they can be composed to decision procedures for a variety of syntactically identifiable subclasses of first-order logic [Bachmair, Ganzinger and Waldmann 1993, Nieuwenhuis 1996, Jacquemard, Meyer and Weidenbach 1998, Weidenbach 1999]. Our level of abstraction is often lower compared to papers that solely are concerned with theory, because we want to emphasize on the implementation relevant aspects of inference/simplification/reduction techniques. Hence, we always refrain from "more elegant" formulations in order to make the consequences for an (efficient) implementation more explicit.

The design concepts introduced in SPASS and discussed here are not necessarily original contributions of the author. For example, the combination of saturation and splitting is original, but the use of indexing techniques [Graf 1996] is a widely used method. Many of the design ideas introduced in SPASS are "common knowledge" among the developers of first-order saturation based theorem provers and are regularly discussed among these. Thus it is hard to say where the origin of some idea comes from and I refer to my colleagues listed in the acknowledgments.

In this chapter I frequently use the notion *in practice* to argue for design decisions. This refers to the problem domains we have been interested in so far: Problems resulting from the analysis/verification of software [Fischer, Schumann and Snelting 1998], from the area of automatic type inference [Frühwirth, Shapiro, Vardi and Yardeni 1991, Charatonik, McAllester, Niwinski, Podelski and Walukiewicz 1998], from the analysis of security protocols [Heintze and Clarke 1999, Weidenbach 1999], planning problems [Kautz and Selman 1996], modal logic problems [Hustadt and Schmidt 1997], and problems from the TPTP problem library [Sutcliffe and Suttner 1998]. If we say that some technique/design/calculus is preferred over some other technique/design/calculus *in practice*, this is always meant with respect to the above mentioned problem domains.

SPASS is freely available from the SPASS homepage at

http://spass.mpi-sb.mpg.de/

After a section on notation and notions (Section 2), an introduction to major design aspects of saturation-based provers (Section 3), we discuss the inference/reduction rules (Section 4) of SPASS. For each rule we provide a formal definition and explain specific aspects of its pragmatics and implementation. In Section 5 we evolve the global design of a prover from all these rules. Finally, the appendix establishes links between all mentioned design concepts, inference/reduction rules and the user interface of SPASS as well as its source code.

2. Foundations

A *multiset* over a set A is a function M from A to the natural numbers. Intuitively, $M(a)$ specifies the number of occurrences of a in M. We say that a is an element of M if $M(a) > 0$. The union, intersection, and difference of multisets are defined by the identities $(M_1 \cup M_2)(x) = M_1(x) + M_2(x)$, $(M_1 \cap M_2)(x) = \min(M_1(x), M_2(x))$, and $(M_1 \setminus M_2)(x) = max(0, M_1(x) - M_2(x))$. We use a set-like notation to describe multisets.

A first-order language is constructed over a signature $\Sigma = (\mathcal{F}, \mathcal{R})$, where $\mathcal{F}$ and $\mathcal{R}$ are non-empty, disjoint, in general infinite sets of function and predicate symbols, respectively. Every function or predicate symbol has some fixed arity. Function and predicate symbols with arity one are called *monadic*. In addition to these sets that are specific for a first-order language, we assume a further, infinite set $\mathcal{X}$ of variable symbols disjoint from the symbols in Σ. Then the set of all *terms* $\mathcal{T}(\mathcal{F}, \mathcal{X})$ is recursively defined by: (i) every function symbol $c \in \mathcal{F}$ with arity zero (a *constant*) is a term, (ii) every variable $x \in \mathcal{X}$ is a term and (iii) whenever $t_1, \ldots, t_n$ are terms and $f \in \mathcal{F}$ is a function symbol with arity n, then $f(t_1, \ldots, t_n)$ is a term. A term not containing a variable is a *ground term*. If $t_1, \ldots, t_n$ are terms and $R \in \mathcal{R}$ is a predicate symbol with arity n, then $R(t_1, \ldots, t_n)$ is an *atom*. An atom or the negation of an atom is called a *literal*. Disjunctions of literals are *clauses* where all variables are implicitly universally quantified. Clauses are often denoted by their respective multisets of literals where we write multisets in usual set notation. A clause consisting of exactly one literal is called a *unit*.

The set of *free* variables of an atom (term) ϕ denoted by $vars(\phi)$ is defined as follows: $vars(P(t_1, \ldots, t_n)) = \cup_i vars(t_i)$ and $vars(f(t_1, \ldots, t_n)) = \cup_i vars(t_i)$, $vars(x) = \{x\}$. The function naturally extends to literals, clauses and (multi)sets of terms (literals, clauses).

A *substitution* σ is a mapping from the set of variables to the set of terms such that $x\sigma \neq x$ for only finitely many $x \in \mathcal{X}$. We define the *domain* of σ to be $dom(\sigma) = \{x \mid x\sigma \neq x\}$ and the co-domain of σ to be $cdom(\sigma) = \{x\sigma \mid x\sigma \neq x\}$. Hence, we can denote a substitution σ by the finite set $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ where $x_i\sigma = t_i$ and $dom(\sigma) = \{x_1, \ldots, x_n\}$. A *ground substitution* σ has no variable occurrences in its co-domain, $vars(cdom(\sigma)) = \emptyset$. An injective substitution σ where $cdom(\sigma) \subset \mathcal{X}$ is called a *variable renaming*. The application of substitutions to terms is given by $f(t_1, \ldots, t_n)\sigma = f(t_1\sigma, \ldots, t_n\sigma)$ for all $f \in \mathcal{F}$ with arity n. We extend the application of substitutions to literals and clauses as usual: $P(t_1, \ldots, t_n)\sigma = P(t_1\sigma, \ldots, t_n\sigma)$ (accordingly for literals)

and $\{L_1, \ldots, L_n\}\sigma = \{L_1\sigma, \ldots, L_n\sigma\}$.

Given two terms (atoms) s, t, a substitution σ is called a *unifier* for s and t if $s\sigma = t\sigma$. It is called a *most general unifier* (mgu) if for any other unifier τ of s, t there exists a substitution λ with $\sigma\lambda = \tau$. A substitution σ is called a *matcher* from s to t if $s\sigma = t$. The notion of a mgu is extended to atoms, literals in the obvious way. We say that σ is a unifier for a sequence of terms (atoms, literals) $t_1, \ldots, t_n$ if $t_i\sigma = t_j\sigma$ for all $1 \leq i, j \leq n$ and σ is a mgu if in addition for any other unifier τ of $t_1, \ldots, t_n$, there exists a substitution λ with $\sigma\lambda = \tau$.

A *position* is a word over the natural numbers. The set $pos(f(t_1, \ldots, t_n))$ of positions of a given term $f(t_1, \ldots, t_n)$ is defined as follows: (i) the empty word ϵ is a position in any term t and $t|_\epsilon = t$, (ii) if $t|_\pi = f(t_1, \ldots, t_n)$, then $\pi.i$ is a position in t for all $i = 1, \ldots, n$, and $t|_{\pi.i} = t_i$. We write $t[s]_\pi$ for $t|_\pi = s$. With $t[\pi/s]$, where $\pi \in pos(t)$, we denote the term (atom) obtained by replacing $t|_\pi$ by s at position π in t. The *length* of a position π is defined by $length(\epsilon) = 0$ and $length(i.\tau) = 1 + length(\tau)$. The notion of a position can be extended to atoms, literals and even formulae in the obvious way.

As an alternative to the already mentioned multiset notation of clauses, we also write clauses in the form $\Theta \parallel \Gamma \rightarrow \Delta$ where Θ is a multiset of monadic atoms[3] and Γ, Δ are multisets containing arbitrary atoms. Logically, the atoms in Θ and Γ denote negative literals while the atoms in Δ denote the positive literals in the clause. The empty clause $\Box$ denotes $\bot$ (falsity). The multiset Θ is called the *sort constraint* of $\Theta \parallel \Gamma \rightarrow \Delta$. A sort constraint Θ is *solved* in a clause $\Theta \parallel \Gamma \rightarrow \Delta$ if it does not contain non-variable terms and $vars(\Theta) \subseteq vars(\Gamma \cup \Delta)$. If the clause is determined by the context, we simply say that a sort constraint is solved. In case we are not interested in a separation of the negative literals in a clause, we write clauses in the form $\Gamma \rightarrow \Delta$. We often abbreviate disjoint set union with sequencing, e.g., we write $\Theta \parallel \Gamma \rightarrow \Delta, R(t_1, \ldots, t_n)$ for $\Theta \parallel \Gamma \rightarrow \Delta \cup \{R(t_1, \ldots, t_n)\}$. Equality atoms are written $l \approx r$ and are mostly distinguished from non-equality atoms. The latter are named A, B. In case we don't want to distinguish these two different kinds of atoms we use the letter E (possibly indexed) to denote an arbitrary atom. Inferences and reductions where equations are involved are applied with respect to the symmetry of $\approx$.

A clause $\Theta_1 \parallel \Gamma_1 \rightarrow \Delta_1$ *subsumes* a clause $\Theta_2 \parallel \Gamma_2 \rightarrow \Delta_2$ if $\Theta_1\sigma \subseteq \Theta_2$, $\Gamma_1\sigma \subseteq \Gamma_2$ and $\Delta_1\sigma \subseteq \Delta_2$ for some matcher σ. The relation "is subsumed by" between clauses is a quasi-ordering on clauses. Please recall that we consider clauses to be multisets. Hence, e.g., the clause $\{P(x), P(y)\}$ (also possibly written $\rightarrow P(x), P(y)$) does not subsume the clause $\{P(x)\}$ (possibly written $\rightarrow P(x)$).

The function $size$ maps terms, atoms, literals to the number of symbols they are built from, e.g., $size(t) = |pos(t)|$. In case of a literal, we don't consider the negation symbol for its size. The *depth* of a term, literal is the maximal length of a position in the term, literal, e.g., $depth(t) = max(\{length(\pi) \mid \pi \in pos(t)\})$. The depth of a clause is the maximal depth of its literals. The size of clause is the sum of its literal sizes.

For the definition of our inference/reduction rules we shall often need the notion of an ordering to compare terms. This notion is then lifted to tuples, sets, clauses and (multi)sets of clauses. A *partial order* is a reflexive, transitive and antisymmetric relation. A *strict order* is a transitive and irreflexive relation. Every partial order $\succeq$ induces a strict order $\succ$

[3] These are atoms with a monadic predicate as their top symbol that form the sort constraint.

by $t \succ s$ iff $t \succeq s$ and $t \neq s$. The lexicographic extension $\succ^{lex}$ on tuples of some strict order $\succ$ is defined by $(t_1, \ldots, t_n) \succ^{lex} (s_1, \ldots, s_n)$ if for some $1 \leq i \leq n$ we have $t_i \succ s_i$ and for all $1 \leq j < i$ it is the case that $t_i = s_i$. The multiset extension $\succ^{mul}$ is defined by $M \succ^{mul} N$ if $N \neq M$ and for all $n \in N \setminus M$ there exists an $m \in M \setminus N$ with $m \succ n$. A *reduction ordering* $\succ$ is a well-founded, transitive relation satisfying for all terms t, s, l, positions $p \in pos(l)$ and substitutions σ that whenever $s \succ t$ then $l[p/s\sigma] \succ l[p/t\sigma]$. For the purpose of this article, we are mainly interested in reduction orderings that are total on ground terms, possibly up to some congruence on the ground terms. Any (reduction) ordering $\succ$ on terms (atoms) can be extended to clauses in the following way. We consider clauses as multisets of *occurrences* of equations and atoms. The occurrence of an equation $s \approx t$ in the antecedent is identified with the multiset $\{\{s, t\}\}$, the occurrence of an atom A in the antecedent is identified with the multiset $\{\{A, \top\}\}$, the occurrence of an equation in the succedent is identified with the multiset $\{\{s\}, \{t\}\}$ and the occurrence of an atom in the succedent is identified with the multiset $\{\{A\}, \{\top\}\}$. We always assume that $\top$ is the minimal constant with respect to $\succ$. Now we overload $\succ$ on literal occurrences to be the twofold multiset extension of $\succ$ on terms (atoms) and $\succ$ on clauses to be the multiset extension of $\succ$ on literal occurrences. If $\succ$ is well-founded (total) on terms (atoms), so are the multiset extensions on literals and clauses.

Observe that an occurrence of an equation $s \approx t$ (an atom) in the antecedent is strictly bigger than an occurrence of $s \approx t$ in the succedent. The atoms in the sort constraint will not be subject to ordering restrictions but will be processed by specific inference/reduction rules.

An antecedent or succedent occurrence of an equation $s \approx t$ (an atom A) is *maximal* in a clause $\Theta \parallel \Lambda \rightarrow \Pi$ if there is no occurrence of an equation or atom in $\Lambda \rightarrow \Pi$ that is strictly greater than the occurrence $s \approx t$ (the atom A) with respect to $\succ$. An antecedent or succedent occurrence of an equation $s \approx t$ is *strictly maximal* in a clause $\Theta \parallel \Lambda \rightarrow \Pi$ if there is no occurrence of an equation in $\Lambda \rightarrow \Pi$ that is greater or equal than the occurrence $s \approx t$ with respect to $\succ$. A clause $\Theta \parallel \Lambda \rightarrow \Pi, s \approx t$ (clause $\Theta \parallel \Lambda \rightarrow \Pi, A$) is *reductive* for the equation $s \approx t$ (the atom A), if $s \approx t$ (the atom A) is a strictly maximal occurrence of an equation (atom) and $t \not\succ s$.

For the specific sort constraint approach introduced here, monadic Horn theories are of particular importance. Such theories provide a natural representation of sort/type information (see Section 4.2). A *Horn clause* is a clause with at most one positive literal. A *monadic Horn theory* is a set of Horn clauses where all occurring predicates are monadic. A *declaration* is a clause $S_1(x_1), \ldots, S_n(x_n) \rightarrow S(t)$ with $\{x_1, \ldots, x_n\} \subseteq vars(t)$. It is called a *term declaration* if t is not a variable and a *subsort declaration* otherwise. A subsort declaration is called *trivial* if $n = 0$. A term t is called *shallow* if t is a variable or is of the form $f(x_1, \ldots, x_n)$ where the x_i are not necessarily different variables. A term t is called *linear* if every variable occurs at most once in t. It is called *semi-linear* if it is a variable or of the form $f(t_1, \ldots, t_n)$ such that every t_i is semi-linear and whenever $vars(t_i) \cap vars(t_j) \neq \emptyset$ we have $t_i = t_j$ for all i, j. A term declaration is called *shallow* (*linear*, *semi-linear*) if t is shallow (linear, semi-linear). Note that shallow term declarations don't include arbitrary ground terms. However, any ground term declaration can be equivalently represented, with respect to the minimal model semantics, by finitely many shallow term declarations. For example, the ground term declaration $\rightarrow S(f(a))$ can be

represented by the shallow declarations $T(x) \rightarrow S(f(x)), \rightarrow T(a)$. A *sort theory* is a finite set of declarations. It is called *shallow* (*linear*, *semi-linear*) if all term declarations are shallow (linear, semi-linear).

A *clause store* is a multiset of clauses. A *clause store collection* is a multiset of clause stores. The inference and reduction rules discussed in this chapter operate on clauses occurring in a clause store of a clause store collection. There are inference rules

$$\mathcal{I}\frac{\Theta_1 \parallel \Gamma_1 \rightarrow \Delta_1 \quad \ldots \quad \Theta_n \parallel \Gamma_n \rightarrow \Delta_n}{\Psi \parallel \Pi \rightarrow \Lambda}$$

reduction rules

$$\mathcal{R}\frac{\Theta_1 \parallel \Gamma_1 \rightarrow \Delta_1 \quad \ldots \quad \Theta_n \parallel \Gamma_n \rightarrow \Delta_n}{\begin{array}{c}\Psi_1 \parallel \Pi_1 \rightarrow \Lambda_1 \\ \vdots \\ \Psi_k \parallel \Pi_k \rightarrow \Lambda_k\end{array}}$$

and splitting rules.

$$\mathcal{S}\frac{\Theta \parallel \Gamma \rightarrow \Delta}{\begin{array}{c|c}\Psi_{1,1} \parallel \Pi_{1,1} \rightarrow \Lambda_{1,1} & \Psi_{1,2} \parallel \Pi_{1,2} \rightarrow \Lambda_{1,2} \\ \vdots & \vdots \\ \Psi_{n,1} \parallel \Pi_{n,1} \rightarrow \Lambda_{n,1} & \Psi_{m,2} \parallel \Pi_{m,2} \rightarrow \Lambda_{m,2}\end{array}}$$

The clauses $\Theta_i \parallel \Gamma_i \rightarrow \Delta_i$ are called the *parent clauses* or *premises* of the splitting (reduction, inference) rule and the clauses $\Psi_{i(,j)} \parallel \Pi_{i(,j)} \rightarrow \Lambda_{i(,j)}$ the *conclusions*. A rule is applied to a clause store collection P by selecting a clause store N out of P such that the premises of an inference (reduction, splitting) rule are contained in N. In this case, N is called the *current* clause store. If an inference is performed, the conclusion of the inference is *added* to N. If a reduction is performed, the premises are *replaced* in N by the conclusions. As a special case, if no conclusion is present, the premises are deleted from N. If a splitting rule is applied, the current store N is *replaced* in P by two stores

$$N \setminus \{\Theta \parallel \Gamma \rightarrow \Delta\} \cup \{\Psi_{j,1} \parallel \Pi_{j,1} \rightarrow \Lambda_{j,1} \mid 1 \leq j \leq n\}$$
$$N \setminus \{\Theta \parallel \Gamma \rightarrow \Delta\} \cup \{\Psi_{j,2} \parallel \Pi_{j,2} \rightarrow \Lambda_{j,2} \mid 1 \leq j \leq m\}$$

One can think of more general splitting rules but the above schema is sufficient for a general understanding of the implementation consequences caused by such a rule and is actually implemented in SPASS (see Section 4.5). Semantically, clause stores represent conjunctions of their clauses whilst clause store collections represent disjunctions of their contained clause stores. So a clause store collection P represents a disjunction (clause stores) of conjunctions (of universally quantified clauses) of disjunctions (of literals).

A clause store N is *saturated* with respect to a set of inference and reduction rules (no splitting rules), if any conclusion of an inference rule application to N yields a clause that can eventually be deleted by a sequence of reduction rule applications. This definition of saturation provides an operational point of view.

3. A First Simple Prover

In this section, we discuss the implementation of a simple resolution based calculus. Although the calculi implemented by SPASS are much more sophisticated than the simple resolution calculus considered here, some important design decisions can already be explained on the basis of such a simple example. The resolution calculus consists of the inference rules resolution, factoring and the reduction rules subsumption deletion and tautology deletion

Resolution

$$\mathcal{I}\frac{\Gamma_1, A \to \Delta_1 \quad \Gamma_2 \to \Delta_2, B}{(\Gamma_1, \Gamma_2 \to \Delta_1, \Delta_2)\sigma}$$

Factoring Right

$$\mathcal{I}\frac{\Gamma \to \Delta, A, B}{(\Gamma \to \Delta, A)\sigma}$$

Subsumption Deletion

$$\mathcal{R}\frac{\Gamma_1 \to \Delta_1 \quad \Gamma_2 \to \Delta_2}{\Gamma_1 \to \Delta_1}$$

Factoring Left

$$\mathcal{I}\frac{\Gamma, A, B \to \Delta}{(\Gamma, A \to \Delta)\sigma}$$

Tautology Deletion

$$\mathcal{R}\frac{\Gamma, A \to \Delta, A}{}$$

where σ is a most general unifier (mgu) of the atoms A and B for the rules resolution, factoring and in order to apply subsumption, the clause $\Gamma_1 \to \Delta_1$ must subsume the clause $\Gamma_2 \to \Delta_2$.

For the resolution rule to be complete, it is required that the parent clauses $\Gamma_1, A \to \Delta_1$ and $\Gamma_2 \to \Delta_2, B$ have no variables in common. Actual implementations of the rule satisfy this requirement in different ways. They all have in common that variables are represented by (natural) numbers, so this is our assumption for the rest of this paragraph. The first solution explicitely renames the clauses such that they have no variables in common. The second solution accepts clauses that share variables, but when running the unification algorithm the variables are separated by adding an offset to the variables of one clause.[4] A typical offset is the value of the maximal, with respect to number greater, variable of the other clause. The third solution also accepts clauses that share variables and solves the problem by employing two substitutions, one for each clause. This requires some modifications to the standard unification algorithms, because the terms of the different atoms need to be explicitely separated. This is the solution implemented in SPASS. In order to test applicability of the resolution rule, it is sufficient to explicitely or implicitly rename the variables of the considered atoms, not the overall clause.

For the factoring rule there is an extra variant for positive (Factoring Right) and negative literals (Factoring Left). We could have presented both variants in one rule, by denoting clauses as disjunctions of literals. However, our representation is closer to the implementation of the rule. All clause data structures used in well-known provers explicitely separate positive from negative literals. The reason is efficiency and already becomes obvious for factoring: Whenever we search for a partner literal for a positive literal it does

[4] Please recall that we assume variables to be represented by naturals.

not make sense to consider negative literals at all. Similar situations arise for other inference/reduction rules. Therefore, the decision in this article is always to distinguish positive and negative literals when presenting inference/reduction rules.

Now let us compose the inference/reduction rules to an actual prover. The input of the prover is a clause store containing clauses without equality and the output on termination is a proof or a saturated clause store. The above resolution calculus is complete, so we also want our search procedure to be complete in the sense that if resources don't matter and our procedure is called with an unsatisfiable clause store then it will eventually find a proof (the empty clause). In order to achieve this goal, we have to guarantee that the considered clause set is saturated in the limit. This includes that all inferences between clauses have been performed. An easy way to remember which inferences have already been performed is to split the input clause store in a set Wo of clauses (Worked off clauses) where all inferences between clauses in this set already took place and a set Us of clauses (Usable clauses) which still have to be considered for inferences. Then a main loop iteration of the prover consists of selecting a clause from the Us set, moving it to the Wo set and then adding all inferences between the selected clause and the clauses in Wo to the Us set. If the selection is fair, i.e., no clause stays arbitrarily long in the Us set without being selected, this results in a complete procedure. It remains to build reductions into this loop. The idea for this loop is due to the Otter theorem prover and its predecessors [McCune and Wos 1997].

The reduction rules tautology deletion and subsumption deletion decrease the number of clauses in the clause store while the inference rules increase the number of clauses. Hence, exhaustive application of the reduction rules terminates and produces smaller clause stores. In practice, small clause sets are preferred over large ones, hence reductions are preferred over inferences. This consideration together with the idea of the main-loop introduced above leads to $ResolutionProver1$ depicted in Table 1. Note that subsumption and tautology deletion are independent in the sense that once all tautologies have been removed, subsumption does not generate new tautologies. Analyzing such dependencies between reductions is one key for an efficient implementation.

For the description of theorem proving procedures we use the following abbreviations: $fac(C)$ is the set of all factoring inference conclusions (left and right) from the clause C, $res(C, D)$ is the set of all resolution inference conclusions between two clauses C and D, $taut(N)$ is the set N after exhaustive application of tautology deletion and $sub(N, M)$ is the set of all clauses from N that are not subsumed by a clause in M. We overload sub for one argument, where $sub(N)$ denotes the set N after exhaustive application of subsumption deletion to the clauses in N. We overload res by defining $res(C, N)$ to be the set of all resolution inferences between the clause C and a clause in N. The function $choose$ selects and removes a clause from its argument clause store and returns the selected clause as well as the updated argument clause store.

As already motivated, the procedure $ResolutionProver1$ operates on two clause stores: Wo and Us. The store Wo holds all clauses that have already been selected for inferences, while the store Us contains all candidate clauses to generate inferences. The prover $ResolutionProver1$ is called with a finite clause store N and tests those for unsatisfiability. Lines 2 and 3 initialize the sets Wo and Us. Note that Us is not initialized with N, but its completely inter-reduced equivalent. This step is called *input reduction*. The search for

```
 1 ResolutionProver1(N)
 2   Wo := ∅;
 3   Us  := taut(sub(N));
 4   While (Us ≠ ∅ and □ ∉ Us)   {
 5     (Given, Us):= choose(Us);
 6     Wo          := Wo ∪ {Given};
 7     New         := res(Given, Wo) ∪ fac(Given);
 8     New         := taut(sub(New));
 9     New         := sub(sub(New, Wo), Us);
10     Wo          := sub(Wo, New);
11     Us          := sub(Us, New) ∪ New;
12   }
13   If (Us = ∅) then print "Completion Found";
14   If (□ ∈ Us) then print "Proof Found";
```

Table 1: A First Resolution Based Prover

the empty clause (a saturation) is implemented by the lines 4–12. The while-loop starting at line 4 terminates if the empty clause is found or the set Us is empty. We will argue below that this implies that the set Wo is saturated. If Us is not empty and the body of the while-loop is entered, the function $choose$ selects at line 5 a clause out of the usable set. The function is fair, if no clause stays in Us for an infinite number of iterations through the while loop. A widely used, fair implementation (heuristic) of $choose$ is to select a lightest clause that is a clause of smallest size. This selection function is fair, because there are only finitely many different clauses with respect to subsumption having less than k symbols, for any constant k.[5] Many refinements of the $choose$ function are possible: using different weights for variable and signature symbols, preferring clauses with more/fewer variables, preferring clauses that contain certain atoms/term structures or considering in addition the depth of a clause in the search space. The depth of a clause in the search space is zero for all input clauses and every conclusion of an inference has the maximal depth of their parent clauses plus one. Many provers use a combination of weight and depth selection, e.g., choosing four times clauses by minimal weight and every fifth time by minimal depth. This combination again goes back to Otter where the ratio can be controlled by the *pick-given* ratio parameter. The parameter is also available in SPASS.

Then the clause $Given$ is selected, removed from Us and added to Wo (lines 5, 6). Next (line 7) all resolution inference conclusions between $Given$ and Wo and all factoring

[5] Note that since the input set N is finite, the relevant signature is finite, too.

inference conclusions from $Given$ are stored in New. Note that since $Given$ is already contained in Wo these inferences include self resolution inferences. The clauses generated so far are called *derived* clauses. The lines 8–11 are devoted to reduction. First, all tautologies and subsumed clauses are removed from New. Then all clauses that are subsumed by a clause in Wo or Us are deleted from New. This operation is called *forward subsumption*. Clauses remaining in New are then used for *backward subsumption*, the subsumption of clauses in the sets Wo and Us by clauses from New. Finally, the clauses from New are added to Us. These clauses are usually called *kept* clauses.

There are two invariants that hold each time line 4 is executed:

- Any resolution inference conclusion from two clauses in Wo (factoring inference conclusion from a clause in Wo) is either contained in Wo, Us or is subsumed by a clause in Wo, Us or is a tautology.
- The sets Wo and Us are completely inter-reduced:
  $Wo \cup Us = taut(Wo \cup Us)$ and
  $Wo \cup Us = sub(Wo \cup Us)$.

A consequence of these invariants to hold is that if the procedure stops then the set Wo is saturated. Furthermore, if the function $choose$ is fair, then the *ResolutionProver1* is complete.

In case that for the set N a satisfiable subset N' is known, e.g., if the clauses represent a proof attempt of a conjecture with respect to some theory that is known to be satisfiable, we could also initialize the sets by $Wo := N'$ and $Us := (N \setminus N')$, obtaining the so called set of support (SOS) strategy [Wos, Robinson and Carson 1965]. The SOS strategy preserves completeness.

Many other saturation based provers (e.g., Otter, SPASS, Waldmeister, see Appendix C) have a search algorithm based on two sets of clauses.[6] SPASS implements $ResolutionProver1$ on a Unix system by the shell invocation

`SPASS -Auto=0 -ISRe -ISFc -RTaut -RFSub -RBSub` <*file*>

where the option `-Auto=0` turns off the automatic mode of SPASS. In this mode SPASS decides on the basis of the input problem the set of inference and reduction rules. If the automatic mode is turned off, no inference/reduction rules are activated. All options starting with an `I` (de)activate inference rules, options starting with an `R` (de)activate reduction rules. So the above call to SPASS activates the inference rules standard resolution (`-ISRe`), standard factoring (`-ISFc`) and the reduction rules tautology deletion (`-RTaut`), forward subsumption(`-RFSub`) and backward subsumption (`-RBSub`). An inference/reduction rule option is activated by setting it to 1 (the default) and deactivated by setting it to 0. For further details consider Appendix A.

For example, we simulate a run of $ResolutionProver1$ on the clauses

$$\begin{array}{ll} 1: & \rightarrow P(f(a)) \\ 2: & P(f(x)) \rightarrow P(x) \\ 3: & P(f(a)), P(f(x)) \rightarrow \end{array}$$

shown in Table 2. For each while-loop iteration, we show the content of the Wo and Us set at line 4, the selected $Given$ clause and the content of New before execution of line 8.

[6] However, they use different names for the sets. So don't be confused.

Iteration 1

$Wo = \emptyset$ $Us = \{1, 2, 3\}$
$Given =$ 1: $\rightarrow P(f(a))$
$New = \emptyset$

Iteration 2 ↓

$Wo = \{1\}$ $Us = \{2, 3\}$
$Given =$ 2: $P(f(x)) \rightarrow P(x)$
$New =$ {4:[Res:1.1,2.1] $\rightarrow P(a)$,
5:[Res:2.1,2.2] $P(f(f(x))) \rightarrow P(x)$ }

Iteration 3 ↓

$Wo = \{1, 2\}$ $Us = \{3, 4, 5\}$
$Given =$ 4: $\rightarrow P(a)$
$New = \emptyset$

Iteration 4 ↓

$Wo = \{1, 2, 4\}$ $Us = \{3, 5\}$
$Given =$ 3: $P(f(a)), P(f(x)) \rightarrow$
$New =$ { 6:[Res:1.1,3.1] $P(f(x)) \rightarrow$,
7:[Res:1.1,3.2] $P(f(a)) \rightarrow$,
8:[Res:2.2,3.1] $P(f(f(a))), P(f(x)) \rightarrow$,
9:[Res:2.2,3.2] $P(f(a)), P(f(f(x))) \rightarrow$,
10:[Fac:3.1,3.2] $P(f(a)) \rightarrow$ }

Iteration 5 ↓

$Wo = \{1, 4\}$ $Us = \{6\}$
$Given =$ 6: $P(f(x)) \rightarrow$
$New =$ {11:[Res:1.1,6.1] $\square$ }

Table 2: A Run of $ResolutionProver1$

Newly generated clauses are printed in full detail while we refer to a clause in the sets Wo and Us only by its unique clause number. The function $choose$ selects lightest clauses.

Every box in Table 2 represents one while-loop iteration. For newly generated clauses we also show the applied inference rule and parent clauses/literals. Here *Res* indicates a resolution inference, *Fac* a factoring inference and the notion *n.m* refers to literal *m* of clause *n*. So, for example, clause 7 is generated by a resolution inference between the first literal of clause 1 and the second literal of clause 3 where literals are counted from left to right. Iteration 4 shows already some common phenomena of saturation based calculi. First, these calculi are typically redundant in the sense that the very same clause can be

generated in various, different ways. For example, clause 7 and clause 10 are logically identical, although the former is generated by a resolution inference while the latter is the result of a factoring application. As a consequence, subsumption is indispensable for saturation based calculi to cut down the number of kept clauses. The situation gets even more dramatic in the context of equality, where a single loop iteration can already cause an explosion in the number of newly generated clauses. This will be discussed in more detail in Section 4. Coming back to our run, note that in the reduction part of while-loop iteration 4, the clauses 2, 3, 5, 7–10 are all subsumed by clause 6. Second, even for this simple example, it happened that the selection of the $Given$ clause is not always unique when choosing lightest clauses. During iteration 4, the clauses 3 and 5 have both size 6, but choosing clause 5 instead of clause 3 would have caused an additional while loop iteration before the empty clause is derived. Of course, the function $choose$ could be refined and we will in fact discuss such refinements, but in practice it happens (and must happen) frequently that several clauses have the same precedence with respect to $choose$. Then selecting the right clause (by accident) can enable a prover to find a proof where it gets lost in the search space by selecting a different one. This phenomenon is common to all theorem provers and can be observed at the yearly CADE CASC system competitions (e.g., see [Sutcliffe and Suttner 1999]), where the performance of provers varies depending on the ordering of the input problem clauses.

If $ResolutionProver1$ is ran on non-trivial examples, the Us set rapidly gets much larger than the Wo set. It easily happens that after some iterations the size increases by a factor of 1000. In particular, it is common in the context of problems containing equality. Therefore, at least with respect to the number of clauses that have to be considered, the subsumption tests with respect to the Us set are the most expensive parts of the algorithm. Typical runs of $ResolutionProver1$ show a behavior where more than 95% of the overall time is spent for subsumption checks. This motivates the design of $ResolutionProver2$ shown in Table 3.

$ResolutionProver2(N)$ does not perform any subsumption tests with respect to the Us set and back subsumption is only performed with respect to the actually selected given clause. The two invariants for $ResolutionProver2$ are

- Any resolution inference conclusion from two clauses in Wo (factoring inference conclusion from a clause in Wo) is either contained in Wo, Us or is subsumed by a clause in Wo or is a tautology.
- The set Wo is completely inter-reduced:
  $Wo = taut(Wo)$ and
  $Wo = sub(Wo)$.

These two invariants are still strong enough to guarantee that if the while loop terminates, the Wo set is saturated. Note that although New is always reduced with respect to Wo at line 11, the set Us is in general not reduced with respect to Wo, i.e., $Us \neq sub(Us, Wo)$.

If we assume that $choose$ selects light clauses there is a further motivation to leave out subsumption tests with respect to the Us set. If a clause C subsumes a clause D, then $size(C) \leq size(D)$. So small clauses have a higher probability to subsume other clauses than larger clauses. Therefore, because we always select the lightest given clause, the hope is that not too many clauses that could have been subsumed stay in the Us set. In prac-

```
 1 ResolutionProver2(N)
 2   Wo          := ∅;
 3   Us          := taut(sub(N));
 4   While (Us ≠ ∅ and □ ∉ Us)   {
 5     (Given, Us):= choose(Us);
 6     if (sub({Given}, Wo) ≠ ∅)   {
 7       Wo          := sub(Wo, {Given});
 8       Wo          := Wo ∪ {Given};
 9       New         := res(Given, Wo) ∪ fac(Given);
10       New         := taut(sub(New));
11       New         := sub(New, Wo);
12       Us          := Us ∪ New;
13     }
14   }
15   If (Us = ∅) then print "Completion Found";
16   If (□ ∈ Us) then print "Proof Found";
```

Table 3: A Second Resolution Based Prover

tice, *ResolutionProver2* saves about 10% of the time spent for reductions (subsumption) compared to *ResolutionProver1*. For the simple resolution calculus we studied so far, *ResolutionProver2* is mostly in favor of *ResolutionProver1* when run in practice. As soon as our reduction techniques include rules that produce lighter clauses (see Section 4) the choice is no longer obvious in general. There are examples where an overall interreduction easily yields the empty clause, but for a *ResolutionProver2* style algorithm sophisticated heuristics are needed to still find a proof.

Running *ResolutionProver2* on the example clause store, the result is similar to the run of *ResolutionProver1* (Table 2). The first three iterations are identical, but at iteration 4, the clauses 2, 3, 5, 7–10 are not subsumed but stay in their respective sets. Then, in iteration 5, where clause 6 is selected as given clause, the clauses 2, 3 are removed from the Wo set (line 7 of *ResolutionProver2*, Table 3) and the empty clause is derived.
SPASS implements $ResolutionProver2$ by the call

```
SPASS -Auto=0 -FullRed=0 -ISRe -ISFc -RTaut -RFSub -RBSub
                        <file>
```

where the option `-FullRed=0` deactivates reduction with respect to the Us set and modifies the algorithm accordingly. For further details consider Appendix A.

There are many possible alternatives, variations, refinements for the two loops suggested here. Let us discuss some aspects. First, concerning factoring, any clause store can be finitely saturated with respect to factoring, since a factor has strictly fewer literals than its parent. So one could get the idea to keep the Wo set always saturated with respect to factoring. The disadvantage of this approach is that the number of factors that can be generated out of one clause grows worst case exponentially in the number of literals. The prover Bliksem allows a user to prefer factors (see Appendix C).

Second, concerning resolution and the selection of the given clause, we could also a priori built for each loop iteration all one step resolvents between the clauses in the Us set and between one parent from the Us set and one parent from the Wo set. Then instead of picking a $Given$ clause, we pick one resolvent, use it for (back and/or forward) reduction and finally add it to the Wo set. This approach results in a more fine grained development of the search space. This design for a proof search is closely related to clause graph resolution [Eisinger 1991].

Third, on the implementation side, if we once decide to implement *ResolutionProver2*, the only information we need for the clauses in Us are their properties with respect to the $choose$ function and how these clauses can be generated. For all clauses except the input clauses it suffices to store references for the parents and the used inference. This way it is possible to store all Us clauses in a compact way. In practice constant space suffices for any clause. This dramatically decreases memory consumption and results in an extra speed up. The necessary regeneration of clauses once they are selected, plays no rôle concerning performance. The Waldmeister prover follows this approach. Fourth, another way to keep the Us set small is to throw away clauses with respect to certain weight or complexity restrictions on the newly generated clauses. Either these clauses are just thrown away resulting in an incomplete procedure, this is supported by Otter, SPASS and Vampire (see Appendix C), or the restrictions can be set in a way such that only finitely many clauses can pass the restriction test and once the search results in such a saturated

set, the restrictions are adjusted and the search is restarted. This design is supported by Bliksem, SPASS and Fiesta. In SPASS the resource restriction strategy is controlled by the flags `BoundMode` specifying the resource type where 0 means no resource restriction, 1 means clause size restriction and 2 means clause depth restriction, the flag `BoundStart` specifying the initial start value to restrict the selected resource type and `BoundLoops` determines how often a saturation caused by resource restrictions is restarted with adjusted restrictions. So the call

`SPASS -BoundMode=1 -BoundStart=5 -BoundLoops=3` <*file*>

causes SPASS to throw away all clauses that have a weight greater 5. If this leads to an empty Us set without finding the empty clause, the bound is increased to the smallest size greater 5 that caused a clause to be deleted. This process is repeated at most 3 times, then any weight restrictions are discarded. Such an exploration of the search space can be particularly useful in the context of unit equational problems.

4. Inference and Reduction Rules

In this section we describe a variety of inference/reduction rules. For every rule, we start with a formal definition of the rule and then, if necessary, discuss aspects of its pragmatics, complexity, interaction with other rules or design concepts and its implementation and usage. Some rules are stated in a general, possibly non-effective form (e.g., see the conflict rule, Definition 4.19). In this case we also discuss effective instantiations. The rules don't form a particular calculus, instead several well-known calculi can be implemented by forming appropriate groups of rules. An example is the simple resolution calculus considered in Section 3.

Many reduction rules can be simulated by one or several inference rule applications followed by a (trivial) subsumption step. As long as the inference rule set is complete this observation is not too surprising, since we require all our rules to be sound. So one might think that the sophisticated reduction machinery introduced in this section is not really necessary but just a waste of resources when implemented. However, it is just the other way round. Reduction rules always lead to "simpler" clause stores by deleting some clause or by replacing a clause by a "simpler" one. This often ensures the termination of exhaustive application of (groups of) such rules and enables application of these rules to *all* clauses. Therefore, in the context of an implementation, reduction rules cannot be simulated by inference rule applications since those don't terminate when applied exhaustively. Inference rules are only applied to some selected $Given$ clause. Reduction rules should be viewed as restricted inference rules that eventually lead to simpler clause stores and help to explore the "easy" parts of the search space (problem). They replace search space exploration by (efficient) calculation. In fact, some of the reduction rules introduced in this section are motivated by decidability results for various first-order logic fragments.

4.1. Reduction Orderings

For many of the inference/reduction rules defined in the sequel, maximality restrictions on literals, terms play an important rôle. The two most popular orderings are the Knuth-Bendix ordering (KBO) [Knuth and Bendix 1970, Peterson 1983] and the recursive path ordering with status (RPOS) [Dershowitz 1982]. For a broad introduction to orderings, consider the article by Dershowitz [1987] and the more recent book by Baader and Nipkow [1998]. The definitions below differ in some details from other definitions found in the literature, but reflect implementation experience.

Nearly all orderings used in todays provers are variations of the KBO and the RPOS. In particular, weaker versions the orderings are often used. For example, purely weight based orderings or variants of the RPOS without recursive consideration of subterms. These weaker versions have the advantage of cheaper computation and when used to restrict inference rules (see Section 4.3) of a broader exploration of the search space. This can be useful for the search of short proofs. We describe KBO and RPOS exactly the way they are implemented in SPASS.

Let $>$ be a strict order on the set of signature symbols (functions, predicates), called a *precedence*. Let $weight$ be a mapping from the set of signature symbols into the non-negative integers. We call a weight function *admissible* for some precedence if for every unary function symbol f with $weight(f) = 0$, the function f is maximal in the precedence, i.e., $f \geq g$ for all other function symbols g. The function $weight$ is extended to a weight function for terms (atoms) as follows: (i) if t is a variable, then $weight(t) = k$, where k is the minimum weight of any constant and (ii) if $t = f(t_1, \ldots, t_n)$, then $weight(t) = weight(f) + \sum_i weight(t_i)$. Let occ be a function returning the number of occurrences $occ(s, t)$ of a term s in a term t, defined by $occ(s, t) = |\{p \in pos(t) \mid t|_p = s\}|$ and let $status$ be a mapping from the signature symbols to the set $\{left, right, mul\}$.

4.1. DEFINITION *(KBO).* If t, s are terms, then $t \succ_{kbo} s$ if $occ(x, t) \geq occ(x, s)$ for every variable $x \in (vars(t) \cup vars(s))$ and
(1) $weight(t) > weight(s)$ or
(2) $weight(t) = weight(s)$ and $t = f(t_1, \ldots, t_k)$ and $s = g(s_1, \ldots, s_l)$ and
  (2a) $f > g$ in the precedence or
  (2b) $f = g$ and
    (2b1) $status(f) = left$ and $(t_1, \ldots, t_k) \succ^{lex}_{kbo} (s_1, \ldots, s_l)$ or
    (2b2) $status(f) = right$ and $(t_k, t_{k-1}, \ldots, t_1) \succ^{lex}_{kbo} (s_l, s_{l-1}, \ldots, s_1)$

Note that in case (2b) the condition $f = g$ implies $k = l$. Multiset status for function symbols can also be defined but does not pay off in practice for the KBO. If the weight function is admissible for the precedence, then the KBO is a reduction ordering [Baader and Nipkow 1998]. If the precedence $>$ is total, then the KBO is total on ground terms (atoms). For some finite set of signature symbols[7] and two terms s, t with $s \succ_{kbo} t$, there are finitely many terms s' with $s \succ_{kbo} s' \succ_{kbo} t$.

The motivation to consider unary function symbols with weight zero comes in particular from group theory. The standard group axioms can be turned into a convergent system

[7] For an infinite set the condition does obviously not hold.

[Baader and Nipkow 1998] using the KBO with precedence $i > f > e$ and weights $weight(i) = 0$, $weight(f) = weight(e) = 1$ where i is the inverse function, f denotes group multiplication and e represents the neutral element. During the saturation (completion) process it is crucial to orient the derived equation $i(f(x,y)) \approx f(i(y),i(x))$ from left to right, for otherwise the saturation process won't terminate. The only way to achieve $i(f(x,y)) \succ f(i(y),i(x))$ is to assign weight 0 to the function symbol i.

Implementation of the KBO can be done straightforward from the definition. For the RPOS we also assume $>$ to be a strict order (precedence) on the set of signature symbols (functions, predicates).

4.2. DEFINITION *(RPOS).* If t, s are terms, then $t \succ_{rpos} s$ if
(1) $s \in vars(t)$ and $t \neq s$ or
(2) $t = f(t_1,\ldots,t_k)$ and $s = g(s_1,\ldots,s_l)$ and
  (2a) $t_i \succeq_{rpos} s$ for some $1 \leq i \leq k$ or
  (2b) $f > g$ and $t \succ_{rpos} s_j$ for all $1 \leq j \leq l$ or
  (2c) $f = g$ and
    (2c1) $status(f) = left$ and $(t_1,\ldots,t_k) \succ^{lex}_{rpos} (s_1,\ldots,s_l)$ and
      $t \succ_{rpos} s_j$ for all $1 \leq j \leq l$ or
    (2c2) $status(f) = right$ and $(t_k,t_{k-1},\ldots,t_1) \succ^{lex}_{rpos} (s_l,s_{l-1},\ldots,s_1)$ and
      $t \succ_{rpos} s_j$ for all $1 \leq j \leq l$ or
    (2c3) $status(f) = mul$ and $\{t_1,\ldots,t_k\} \succ^{mul}_{rpos} \{s_1,\ldots,s_l\}$

The RPOS is a reduction ordering as well and if the precedence $>$ is total RPOS is also total on ground terms (atoms), up to the congruence relation $=_{mul}$ generated from the symbols with multiset status. If f is a function symbol with $status(f) = mul$ then $f(t_1,\ldots,t_n) =_{mul} f(s_1,\ldots,s_n)$ if $\{t_1,\ldots,t_n\} =^{mul}_{mul} \{s_1,\ldots,s_n\}$, for example $f(a,f(a,b)) =_{mul} f(f(b,a),a)$. Even for some finite set of signature symbols and two terms s, t with $s \succ_{rpos} t$, there are in general infinitely many terms s' with $s \succ_{rpos} s' \succ_{rpos} t$.

The RPOS can for example be used to orient distributivity the "right way". If $f > g$, then the equation $f(x,g(y,z)) \approx g(f(x,y),f(x,z))$ is oriented by RPOS from left to right. Note that KBO cannot orient the equation from left to right, because the right hand side has one more occurrence of the variable x.

Given a specific theorem proving problem, the relevant signature is finite and fixed. In this case it can be useful to further refine an ordering by defining $t \succ s$ if $t\sigma \succ s\sigma$ for all ground substitutions σ where $(vars(t) \cup vars(s)) \subseteq dom(\sigma)$. Following this idea, RPOS can be instantiated to an ordering that totally orders all atoms by predicate symbols and only in second place considers possible argument terms, independently from variable occurrences! This can be achieved by making all (some) predicate symbols larger in the precedence than all function symbols. For example, with respect to the above suggested lifting and a signature with predicate symbols P, Q and function symbols f, a where $P > Q > f > a$ it holds that $P(x) \succ_{rpos} Q(f(x,y))$ because P is greater in the precedence than Q, f, a and hence any ground term that can be substituted for x or y. Predicates can be declared to be superior over function symbols, by a declaration

`set_DomPred(`< *predicate sequence* >`).`

in the SPASS settings section of an input file [Hähnle, Kerber and Weidenbach 1996]. Such an application of RPOS can, e.g., be useful to make literals built from newly introduced formula renaming predicates minimal. This prevents the generation of the standard CNF via ordered resolution [Nonnengart and Weidenbach 2001].

Straightforward recursive implementation of RPOS following the definition results in an algorithm with worst case exponential complexity. Using a dynamic programming idea, a polynomial algorithm can be devised [Snyder 1993]. However, in practice, it turns out that the straightforward implementation is superior to the dynamic programming approach, if the following filter is added. Whenever we test $t \succ_{rpos} s$ for two terms s, t, we first check $vars(s) \subseteq vars(t)$.

4.2. Sorts

The motivation for sorts comes from programming languages, where one likes to catch as many errors at compile time as possible. For example, if the addition function is only defined for number sorts (types) but used in a program with a list type, the compiler can complain about such a statement by exploiting the sort information. Of course, the sort checking must be tractable, i.e., it should at least be decidable and/or show acceptable performance for real world programs. A prerequisite for the sort information to be checked at compile time, is that the sort information is separated from the program and it is typically included in an extra declaration part.

Here we generalize this situation. The sort information is not separated from the first-order problem as, e.g., done in algebraic specification languages, but part of the problem itself. Therefore, we cannot check sort information at compile time, after or while reading the problem. Instead the sort information is used at run time, during proof search, to detect ill-sorted and therefore redundant clauses and to simplify the sort information contained in the clauses by specific algorithms. These algorithms exploit the sort information in a much more efficient way than their standard first-order reduction rule counterparts.

Sorts are activated in SPASS by the `-Sorts` option. If SPASS is called with option `-Sorts=1` all negative monadic literals with a variable argument are considered for the initial sort constraints, whereas `-Sorts=2` causes SPASS to consider all negative monadic literals for the initial sort constraints. The latter choice can affect completeness, because of the basicness restriction on the sort constraint.

4.3. DEFINITION *(Sort Constraint Resolution).* The inference

$$\mathcal{I}\frac{T_1(t), \ldots, T_n(t), \Psi \,\|\, \Gamma \rightarrow \Delta \qquad \Theta_i \,\|\, \Gamma_i \rightarrow \Delta_i, T_i(s_i) \quad (1 \leq i \leq n)}{(\Theta_1, \ldots, \Theta_n, \Psi \,\|\, \Gamma_1, \ldots, \Gamma_n, \Gamma \rightarrow \Delta_1, \ldots, \Delta_n, \Delta)\sigma}$$

where (i) σ is the simultaneous mgu of $t, s_1, \ldots, s_n$, (ii) t is a non-variable term and there is no further literal $S(t) \in \Psi$, (iii) all Θ_i are solved, (iv) all $T_i(s_i)\sigma$ are reductive for $(\Theta_i \,\|\, \Gamma_i \rightarrow \Delta_i, T_i(s_i))\sigma$ is a *sort constraint resolution* inference.

Sort constraint resolution is a hyper resolution (see Definition 4.14) like inference rule. It simulates the rule weakening of sorted unification [Weidenbach 1998] on the relativization

of sorted variables represented by the sort constraint. Sort constraint resolution is activated by the `-ISoR` option.

4.4. DEFINITION *(Empty Sort).* The inference

$$\mathcal{I}\frac{T_1(x),\ldots,T_n(x),\Psi \,\|\, \Gamma \rightarrow \Delta \qquad \Theta_i \,\|\, \Gamma_i \rightarrow \Delta_i, T_i(s_i) \quad (1 \leq i \leq n)}{(\Theta_1,\ldots,\Theta_n,\Psi \,\|\, \Gamma_1,\ldots,\Gamma_n,\Gamma \rightarrow \Delta_1,\ldots,\Delta_n,\Delta)\sigma}$$

where (i) σ is the simultaneous mgu of $s_1,\ldots,s_n$, (ii) $x \notin vars(\Gamma \cup \Delta \cup \Psi)$ and no non-variable term occurs in Ψ, (iii) all Θ_i are solved, (iv) all $T_i(s_i)\sigma$ are reductive for $(\Theta_i \,\|\, \Gamma_i \rightarrow \Delta_i, T_i(s_i))\sigma$ is an *empty sort* inference.

Empty sort is similar to sort resolution and, in fact, in some of our papers (e.g., Jacquemard et al. [1998]) we unified both rules into one inference rule. For the purpose of the decidability results presented in these papers this is appropriate. It makes sense to distinguish these rules, because the eventual success of empty sort, i.e., we are able to show that some sort is non-empty, does not rely on the particular sort constraint, but only on the set of monadic (sort) symbols that share their variable argument. We check emptiness of an intersection of sort symbols. Since there are only finitely many different such sorts with respect to some finite clause store, it may make sense to store constraints that resulted in successful non-emptiness proofs and to reuse them. One application domain are the proofs required in the context of static soft typing (Definition 4.6). Empty sort is activated by the `-IEmS` option.

4.5. DEFINITION *(Sort Simplification).* Let N be the current clause store and $N' \subseteq N$ be exactly the set of all declarations in N. The reduction

$$\mathcal{R}\frac{S(t),\Theta \,\|\, \Gamma \rightarrow \Delta}{\Theta \,\|\, \Gamma \rightarrow \Delta}$$

where $N' \models \forall x_1,\ldots,x_n\,[S_1(x_1),\ldots,S_n(x_n) \supset S(t)]$ and $\{S_1(x_1),\ldots,S_n(x_n)\} \subseteq \Theta$ is the maximal subset of Θ for which $\{x_1,\ldots,x_n\} \subseteq vars(t)$ is called *sort simplification*.

Given an arbitrary sort theory N', the relation

$$N' \models \forall x_1,\ldots,x_n\,[S_1(x_1),\ldots,S_n(x_n) \supset S(t)]$$

is always decidable in polynomial time. In terms of sorted unification the problem means deciding well-sortedness [Weidenbach 1998]. A bottom-up algorithm based on dynamic programming yields the polynomial complexity whereas a simple top down approach results in an exponential procedure. The latter procedure would correspond to solve the problem with ordered resolution and an SOS strategy. The former algorithm is implemented in SPASS. Sort simplification is activated by the `-RSSi` option.

Sort simplification is one important reason why it makes sense to treat particular occurrences of monadic predicates in a special way. Sort simplification cannot be simulated via other standard reduction techniques like matching replacement resolution (see Definition 4.20) and cannot be extended to non-monadic predicates. For example, for binary

relations, the undecidable problem whether two ground terms are contained in a transitive binary relation generated by some positive unit clauses [Schmidt-Schauß 1988] can be reduced to deciding applicability of an extended sort simplification rule for binary relations. So without further restrictions, sort simplification cannot be effectively used for other n-ary relations.

4.6. DEFINITION *(Static Soft Typing).* Let N be the current clause store over some fixed signature Σ and M be a sort theory such that $N \models S(t)$ implies $M \models S(t)$ for any ground monadic atom $S(t)$ over Σ where S occurs in some sort constraint in N. The reduction

$$\mathcal{R}\frac{\Theta \parallel \Gamma \to \Delta}{}$$

where $M \not\models \exists x_1, \ldots, x_n\, \Theta$ with $vars(\Theta) = \{x_1, \ldots, x_n\}$ is called *static soft typing*.

The above definition of static soft typing is not effective. The problem $M \not\models \exists x_1, \ldots, x_n\, \Theta$ is not decidable for arbitrary sort theories M and sort constraints Θ. It includes the general problem of sorted unification [Weidenbach 1998] that is well-known to be undecidable, in general. Furthermore, it is not obvious how the sort theory M can be constructed out of N such that it meets the requirements of Definition 4.6. A solution to all these problems is the following. First, all clauses that contain positive monadic atoms are safely approximated and restricted to the sort information they contain:

$$\mathcal{R}\frac{\Theta \parallel \Gamma \to \Delta, S_1(t_1), \ldots, S_n(t_n)}{\begin{array}{c}\Theta_1 \parallel \to S_1(t_1)\\ \vdots \\ \Theta_n \parallel \to S_n(t_n)\end{array}}$$

where no monadic atom occurs in Δ, Θ is solved and $\Theta_i = \{S(x) \mid S(x) \in \Theta \text{ and } x \in vars(S_i(t_i))\}$ for $1 \leq i \leq n$. By construction all Θ_i are solved and the rule does not modify declarations. If the initial clause set N does not contain positive equations, then the sort theory N' obtained by a fix-point computation of the above reduction on N approximates N in the desired way (see Definition 4.6). Second, the sort theory N' is approximated to a sort theory N'' such that satisfiability of sort constraints in N'' gets decidable.

$$\mathcal{R}\frac{\Theta \parallel \to S(f(t_1, \ldots, t_n))}{\begin{array}{c}\Theta_1, T(x) \parallel \to S(f(s_1, \ldots, s_n))\\ \Theta_2 \parallel \to T(t_i)\end{array}}$$

where t_i is not a variable and for all $1 \leq j \leq n$ we define $s_j = x$ if $t_j = t_i$ and $s_j = t_j$ otherwise. Furthermore, $\Theta_1 = \{S(y) \mid S(y) \in \Theta \text{ and } y \in vars(S(f(s_1, \ldots, s_n)))\}$ and Θ_2 is the restriction of Θ to atoms with argument $x \in vars(t_i)$.

By construction, the derived clauses have a solved sort constraint and N'' approximates N' as desired. The sort theory N'' is shallow and satisfiability of sort constraints with respect to shallow sort theories is decidable by the inference rules sort resolution, empty

sort and the reduction rules sort simplification, subsumption deletion (Definition 4.16) and condensation (Definition 4.17) [Jacquemard et al. 1998, Weidenbach 1999]. Hence, this instance of static soft typing is effective.

So if we start with a clause store N that does not contain positive equations, we construct once the approximated sort theory N''. If this theory is not trivial, i.e., there is at least one monadic predicate S with $N'' \not\models \forall x\, S(x)$, the sort theory N'' is stored and static soft typing is applied to any input or derived clause. Since N'' is only approximated once, typically at the beginning of the inference process, the rule is called static soft typing. If in the input clause store all sort constraints are solved and there are no positive equations, static soft typing preserves completeness [Weidenbach 1996, Ganzinger, Meyer and Weidenbach 1997].

If equations occur in a clause store a dynamic soft typing approach seems to be more suitable. Consider Ganzinger et al. [1997] and Meyer [1999] for details. These techniques are not implemented in SPASS Version 2.0 but are an option for later releases. Static soft typing is activated by the `-RRSST` option.

4.3. Inference Rules

The introduced inference rules can be composed to a variety of (well-known) calculi. The calculi range from the ordinary resolution calculus investigated in Section 3 to a superposition calculus with selection, splitting and sort constraints that are subject to the basicness restriction. To cover all these cases, the rules defined here are given in generic way such that each definition covers several variants of the rule. In particular, all rules are available with a selection restriction of negative literals that does not destroy completeness [Bachmair and Ganzinger 1994]. For any clause we can select some negative literals with the effect that all inference rule applications taking this clause as a parent clause must involve the selected literals. For example, if we select the literal $R(x, y)$ in the clause $\| R(x, y), f(g(x), y) \approx f(y, z) \rightarrow$ then no equality resolution inference (see below) is possible from this clause.

4.7. DEFINITION *(Equality/Reflexivity Resolution).* The inference

$$\mathcal{I}\frac{\Theta \,\|\, l \approx r, \Gamma \rightarrow \Delta}{(\Theta \,\|\, \Gamma \rightarrow \Delta)\sigma}$$

where (i) σ is the mgu of l and r, (ii) Θ is solved, (iii) $l \approx r$ is selected or $(l \approx r)\sigma$ is maximal in $(\Theta \,\|\, l \approx r, \Gamma \rightarrow \Delta)\sigma$ and no literal is selected in Γ is called an *equality resolution* inference. If condition (iii) is replaced by $l \approx r$ is selected or no literal is selected in Γ, the inference is called *reflexivity resolution*.

Equality resolution is activated by the `-IEqR` option. Reflexivity resolution is activated by the `-IERR` option.

4.8. DEFINITION *((Ordered) Paramodulation/Superposition Left).* The inferences

$$\mathcal{I}\frac{\Theta_1 \,\|\, \Gamma_1 \to \Delta_1, l \approx r \qquad \Theta_2 \,\|\, s[l']_p \approx t, \Gamma_2 \to \Delta_2}{(\Theta_1, \Theta_2 \,\|\, s[p/r] \approx t, \Gamma_1, \Gamma_2 \to \Delta_1, \Delta_2)\sigma}$$

and

$$\mathcal{I}\frac{\Theta_1 \,\|\, \Gamma_1 \to \Delta_1, l \approx r \qquad \Theta_2 \,\|\, A[l']_p, \Gamma_2 \to \Delta_2}{(\Theta_1, \Theta_2 \,\|\, A[p/r], \Gamma_1, \Gamma_2 \to \Delta_1, \Delta_2)\sigma}$$

where (i) σ is the mgu of l' and l, (ii) l' is not a variable, (iii) Θ_1 and Θ_2 are solved (iv) no literal in Γ_1 is selected, (v) $s \approx t$ (the atom A) is selected or no literal in Γ_2 is selected, is called a *paramodulation left* inference. If, in addition, $r\sigma \not\succ l\sigma$ the inference is an *ordered paramodulation left* inference. If, in addition, $l\sigma \approx r\sigma$ is reductive for $(\Theta_1 \,\|\, \Gamma_1 \to \Delta_1, l \approx r)\sigma$, (v) is replaced by $s\sigma \approx t\sigma$ (the atom $A\sigma$) is selected or it is maximal in $(\Theta_2 \,\|\, s \approx t, \Gamma_2 \to \Delta_2)\sigma$ (in $(\Theta_2 \,\|\, A, \Gamma_2 \to \Delta_2)\sigma$) and no literal in Γ_2 is selected and $t\sigma \not\succ s\sigma$ then the inference is called a *superposition left* inference.

Standard paramodulation is activated by the `-ISPm` option. Ordered paramodulation is activated by the `-IOPm` option. Superposition left is activated by the `-ISpL` option.

Note that no paramodulation/superposition inference is performed into the sort constraint. Hence, the sort constraint is subject to the basicness restriction. In case all sort constraints of an initial clause store were solved, the basicness restriction preserves completeness.

4.9. DEFINITION *((Ordered) Paramodulation/Superposition Right).* The inferences

$$\mathcal{I}\frac{\Theta_1 \,\|\, \Gamma_1 \to \Delta_1, l \approx r \qquad \Theta_2 \,\|\, \Gamma_2 \to \Delta_2, s[l']_p \approx t}{(\Theta_1, \Theta_2 \,\|\, \Gamma_1, \Gamma_2 \to \Delta_1, \Delta_2, s[p/r] \approx t)\sigma}$$

and

$$\mathcal{I}\frac{\Theta_1 \,\|\, \Gamma_1 \to \Delta_1, l \approx r \qquad \Theta_2 \,\|\, \Gamma_2 \to \Delta_2, A[l']_p}{(\Theta_1, \Theta_2 \,\|\, \Gamma_1, \Gamma_2 \to \Delta_1, \Delta_2, A[p/r])\sigma}$$

where (i) σ is the mgu of l' and l, (ii) l' is not a variable, (iii) Θ_1 and Θ_2 are solved (iv) no literal in Γ_1, Γ_2 is selected is a *paramodulation right* inference. If, in addition, $r\sigma \not\succ l\sigma$ the inference is an *ordered paramodulation right* inference. If, in addition, $l\sigma \approx r\sigma$ is reductive for $(\Theta_1 \,\|\, \Gamma_1 \to \Delta_1, l \approx r)\sigma$, $s\sigma \approx t\sigma$ $(A\sigma)$ is reductive for $(\Theta_2 \,\|\, \Gamma_2 \to \Delta_2, s \approx t)\sigma$ $(\Theta_2 \,\|\, \Gamma_2 \to \Delta_2, A)\sigma$ the inference is called a *superposition right* inference.

Superposition right is activated by the `-ISpR` option.

In SPASS the parallel extensions of the above defined paramodulation/superposition left/right inferences [Benanav 1990] are preferred. Whenever such an inference rule is applicable, we don't only replace the initially found occurrence of $l\sigma$ in the second clause by $r\sigma$, but all occurrences. On the ground level the parallel replacement corresponds to an application of the inference rules exactly the way they are defined above plus exhaustive application of non-unit rewriting with the left premise (Definition 4.21) on the conclusion.

Note that the ordering conditions of the above inference rules as well as the ordering conditions of the inference rules defined below, are checked with respect to the found unifier. This is called the a posteriori ordering check. For all inference rules that have ordering

restrictions, SPASS first orders the clauses as they are and only searches for inferences with respect to the found candidates (maximal literals, maximal sides of equations). This is called the a priori ordering check. Then, after having found a second candidate clause together with a unifier, the a posteriori check is evaluated. This second check is more expensive than the first, because it has to be dynamically computed with respect to any found unifier. However, since most of the time in a saturation prover is spent with reduction (see Section 3), the extra time for the a posteriori check does not matter, but needs some effort for an (efficient) implementation.

The following example shows that the a posteriori check can in fact prevent the generation of extra clauses. Consider the two clauses

$$\rightarrow f(x,y) \approx f(y,x)$$
$$\rightarrow P(f(a,b))$$

The equation $f(x,y) \approx f(y,x)$ cannot be oriented by any reduction ordering. So without an a posteriori ordering check, we can derive the clause $\rightarrow P(f(b,a))$ by a superposition right inference. Now consider the very same example where we use an RPOS with precedence $f > b > a$ and $status(f) = \mathit{left}$. This implies $f(b,a) \succ_{rpos} f(a,b)$ and therefore the a posteriori ordering check for the potential superposition right inference conclusion $\rightarrow P(f(b,a))$ fails. No inference is possible between the above two clauses.

Next we define three factoring rules, namely (ordered) factoring, equality factoring and merging paramodulation. The different rules are needed to obtain completeness results with respect to different inference rule sets. For the standard resolution/paramodulation calculus [Robinson 1965, Robinson and Wos 1969, Chang and Lee 1973, Peterson 1983] the factoring rule without ordering restrictions suffices for completeness. For the ordered resolution/superposition calculus, ordered factoring has to be combined with either equality factoring or merging paramodulation to obtain completeness [Bachmair and Ganzinger 1994].

4.10. DEFINITION *((Ordered) Factoring).* The inferences

$$\mathcal{I}\frac{\Theta \parallel \Gamma \rightarrow \Delta, E_1, E_2}{(\Theta \parallel \Gamma \rightarrow \Delta, E_1)\sigma}$$

and

$$\mathcal{I}\frac{\Theta \parallel \Gamma, E_1, E_2 \rightarrow \Delta}{(\Theta \parallel \Gamma, E_1 \rightarrow \Delta)\sigma}$$

where (i) σ is the mgu of E_1 and E_2, (ii) Θ is solved (iii) (E_1, E_2 occur positively, E_1 is maximal and no literal in Γ is selected) or (E_1, E_2 occur negatively, E_1 is maximal and no literal in Γ is selected or E_1 is selected) are called *ordered factoring right* and *ordered factoring left*, respectively. If condition (iii) is replaced by (E_1, E_2 occur positively and no literal in Γ is selected) or (E_1, E_2 occur negatively, E_1 is selected or no literal in Γ is selected) the inferences are called *factoring right* and *factoring left*, respectively.

Standard factoring is activated by the `-ISFc` option. Ordered factoring is activated by the `-IOFc` option.

There is an overlap between Ordered Factoring defined above and Equality Factoring defined below, because the rule ordered factoring also considers equations. We did so because for the ordered paramodulation calculus with respect to our definitions Equality Factoring is not needed for completeness. The rule Ordered Factoring suffices for completeness.

4.11. DEFINITION *(Equality Factoring).* The inference

$$\mathcal{I}\frac{\Theta \parallel \Gamma \to \Delta, l \approx r, l' \approx r'}{(\Theta \parallel \Gamma, r \approx r' \to \Delta, l' \approx r')\sigma}$$

where (i) σ is the mgu of l' and l, (ii) $r\sigma \not\succeq l\sigma$, (iii) Θ is solved, (iv) no literal in Γ is selected, (v) $l\sigma \approx r\sigma$ is a maximal occurrence in $(\Theta \parallel \Gamma \to \Delta, l \approx r, l' \approx r')\sigma$ is called an *equality factoring* inference.

Equality factoring is activated by the `-IEqF` option.

4.12. DEFINITION *(Merging Paramodulation).* The inference

$$\mathcal{I}\frac{\Theta_1 \parallel \Gamma_1 \to \Delta_1, l \approx r \qquad \Theta_2 \parallel \Gamma_2 \to \Delta_2, s \approx t[l']_p, s' \approx t'}{(\Theta_1, \Theta_2 \parallel \Gamma_1, \Gamma_2 \to \Delta_1, \Delta_2, s \approx t[p/r], s \approx t')\sigma}$$

where (i) σ is the composition of the mgu τ of l and l' and the mgu λ of $s\tau$ and $s'\tau$, (ii) the clause $(\Theta_1 \parallel \Gamma_1 \to \Delta_1, l \approx r)\sigma$ is reductive for $l\sigma \approx r\sigma$, (iii) Θ_1 and Θ_2 are solved, (iv) no literal in Γ_1, Γ_2 is selected, (v) the clause $(\Theta_2 \parallel \Gamma_2 \to \Delta_2, s \approx t, s' \approx t')\sigma$ is reductive for $s\sigma \approx t\sigma$, (vi) $s\tau \succ t\tau$, (vii) l' is not a variable is called a *merging paramodulation* inference.

Merging paramodulation is activated by the `-IMPm` option.

4.13. DEFINITION *((Ordered) Resolution).* The inference

$$\mathcal{I}\frac{\Theta_1 \parallel \Gamma_1 \to \Delta_1, E_1 \qquad \Theta_2 \parallel E_2, \Gamma_2 \to \Delta_2}{(\Theta_1, \Theta_2 \parallel \Gamma_1, \Gamma_2 \to \Delta_1, \Delta_2)\sigma}$$

where (i) σ is the mgu of E_1 and E_2, (ii) Θ_1 and Θ_2 are solved, (iii) no literal in Γ_1 is selected, (iv) $E_1\sigma$ is strictly maximal in $(\Theta_1 \parallel \Gamma_1 \to \Delta_1, E_1)\sigma$, (v) the atom $E_2\sigma$ is selected or it is maximal in $(\Theta_2 \parallel E_2, \Gamma_2 \to \Delta_2)\sigma$ and no literal in Γ_2 is selected is called *ordered resolution*. If conditions (iv), (v) are replaced by E_2 is selected or no literal is selected in Γ_2, the inference is called *resolution*.

Standard resolution is activated by the `-ISRe` option. Ordered resolution is activated by the `-IORe` option. If any of the options is set to `2`, equations are also considered for the inferences.

If, in Definition 4.13, one of the parent clauses of the inference is a unit, the inference is called *(ordered) unit resolution*. The standard resolution rule is an instance of this rule if we omit the conditions (ii)–(iv) and restrict our attention to non-equational atoms.

4.14. DEFINITION *((Ordered) Hyper Resolution).* The inference

$$\mathcal{I}\frac{\Theta \parallel E_1, \ldots, E_n \rightarrow \Delta \qquad \Theta_i \parallel \rightarrow \Delta_i, E_i' \quad (1 \leq i \leq n)}{(\Theta, \Theta_1, \ldots, \Theta_n \parallel \rightarrow \Delta, \Delta_1, \ldots, \Delta_n)\sigma}$$

(i) σ is the simultaneous mgu of $E_1, \ldots, E_n, E_1', \ldots, E_n'$, (ii) Θ as well as all Θ_i are solved, (iii) all $E_i'\sigma$ are strictly maximal in $(\Theta_i \parallel \Gamma_i \rightarrow \Delta_i, E_i')\sigma$ is called an *ordered hyper resolution* inference. If condition (iii) is dropped, the inference is called a *hyper resolution* inference.

Standard hyper resolution is activated by the `-ISHy` option. Ordered hyper resolution is activated by the `-IOHy` option.

In the application of the inference rule hyper resolution as well as the inference rules sort resolution (Definition 4.3) and empty sort (Definition 4.4) more than two parent clauses are involved, in general. So the search for candidate clauses gets more complicated. In particular, an appropriate ordering of the literals $E_1, \ldots, E_n$ for searching partner clauses can be indispensable for efficiency reasons. For example, if we search partners for the literals $P(x), Q(a, f(x))$ it may be the case that we find thousand potential partners for $P(x)$ (all clauses with a positive (maximal) literal $P(t)$) but only a few for $Q(a, f(x))$ (only clauses with a positive (maximal) literal $Q(a, f(t))$ or with variable occurrences at the positions of a, $f(t)$). So starting with $Q(a, f(x))$ for partner search is the more efficient way, since it will potentially provide instantiation of x when we subsequently search for partners of $P(x)$. So a good heuristic is to proceed at any time of the partner search with the literals that has a maximal number of symbols with respect to the already established partial unifier. Nevertheless, please note that the number of hyper resolvents grows in the worst case exponentially in n.

4.4. Reduction Rules

Our philosophy is that reduction rules are at the heart of successful automated theorem proving. The aim of reduction rules is to transform clauses (or even formulas see Chapter VI) in simpler ones. So whereas inference rules are at the search side of automated theorem proving, reduction rules are at the computation side.

4.15. DEFINITION *(Duplicate/Trivial Literal Elimination).* The reductions

$$\mathcal{R}\frac{\Theta \parallel \Gamma \rightarrow \Delta, E, E}{\Theta \parallel \Gamma \rightarrow \Delta, E}$$

and

$$\mathcal{R}\frac{\Theta \parallel \Gamma, E, E \rightarrow \Delta}{\Theta \parallel \Gamma, E \rightarrow \Delta}$$

and

$$\mathcal{R}\frac{\Theta, A, A \parallel \Gamma \rightarrow \Delta}{\Theta, A \parallel \Gamma \rightarrow \Delta}$$

are called *duplicate literal eliminations*. The reductions

$$\mathcal{R}\frac{\Theta \,\|\, \Gamma, t \approx t \to \Delta}{\Theta \,\|\, \Gamma \to \Delta}$$

and

$$\mathcal{R}\frac{\|\, t \approx s \to}{\Box}$$

where for the final variant we assume that t and s are unifiable, are called *trivial literal eliminations*.

Duplicate/trivial literal elmination are both activated by the `-RObv` option.

Please recall that although trivial literal elimination can be simulated by equality resolution or factoring, for these inference rules to apply a clause must first be selected as $Given$ clause. Reduction rules like duplicate or trivial literal elimination apply to all (newly) generated clauses.

4.16. DEFINITION *(Subsumption Deletion).* The reduction

$$\mathcal{R}\frac{\Theta_1 \,\|\, \Gamma_1 \to \Delta_1 \qquad \Theta_2 \,\|\, \Gamma_2 \to \Delta_2}{\Theta_1 \,\|\, \Gamma_1 \to \Delta_1}$$

where $\Theta_2 \,\|\, \Gamma_2 \to \Delta_2$ is subsumed by $\Theta_1 \,\|\, \Gamma_1 \to \Delta_1$ is called *subsumption deletion*.

Subsumption deletion is activated by the `-RFSub` and `-RBSub` option for forward and backward subsumtion, respectively (see Section refsecfirstsimpprover).

Testing subsumption between two clauses is an *NP*-complete problem [Garey and Johnson 1979]. Nevertheless, subsumption is indispensable for saturation based theorem proving as we already discussed in Section 3. Hence, there exist a variety of papers presenting algorithms that show a polynomial behavior on certain subclasses of clauses (e.g., [Gottlob and Leitsch 1985]) or that introduce specific data structures to speed up the subsumption test in practice (e.g., [Socher 1988, Tammet 1998]). Many of todays provers use a variant of the Stillman [1973] algorithm for the subsumption test. Basically, the algorithm tries to find for every literal in $\Theta_1 \,\|\, \Gamma_1 \to \Delta_1$ a different instance in $\Theta_2 \,\|\, \Gamma_2 \to \Delta_2$ such that all single instantiations are compatible, i.e., identical variables are mapped to identical terms. This simple version is not tractable in practice. Prefilters must be added to the algorithm that make it tractable in practice. In SPASS we introduced two filters [Nonnengart et al. 1998]. The first filter is based on the size of the clauses. A necessary condition for a subsumption deletion application over multisets is that $size(\Theta_1 \,\|\, \Gamma_1 \to \Delta_1) \leq size(\Theta_2 \,\|\, \Gamma_2 \to \Delta_2)$. Since the size of clauses is usually needed for selection heuristics (see the discussion on the $choose$ function in Section 3), the size of a clause is already stored in a clause data structure and therefore this test is almost for free. For every two clauses passing this test, the second prefilter checks whether for every literal in $\Theta_1 \,\|\, \Gamma_1 \to \Delta_1$ there exists some instance literal in $\Theta_2 \,\|\, \Gamma_2 \to \Delta_2$ at all. So we consider the literals in $\Theta_1 \,\|\, \Gamma_1 \to \Delta_1$ separately and don't check compatibility between the different substitutions. This check is again a necessary condition for the subsumption

test to succeed and can be done in polynomial time. Clauses passing these two tests are then subject to the Stillman algorithm. In practice, more than 95% of all subsumption tests can already be rejected by the two filters.

Note that there is a subtle difference between multiset subsumption (considered here) and set subsumption. The clause $\rightarrow Q(a,x), Q(y,b)$ subsumes the clause $\rightarrow Q(a,b)$ if we consider clauses to be sets, but does not if we consider clauses to be multisets. Therefore, in our version of the Stillman algorithm we require matched literals to be different.

When integrated into a prover, subsumption deletion is not an operation applied to two clauses but applied to two sets of clauses or a clause and a set of clauses (see Table 1 and Table 3). The former test can be reduced to the latter by considering the clauses in one set separately. So it remains to test whether some clause C subsumes some clause in a set N or is subsumed by some clause in N. We already argued that the set N (in particular the Us set) can become very large. Then it is in practice intractable to traverse all clauses in N and then to apply the subsumption test to each clause. An additional filter is needed: Indexing. Indexing is the data base technology of automated theorem proving. The crucial operations provided by an index of a clause store N are: compute all clauses that include an atom that is an instance/a generalization/unifiable with some query atom. Typically, the result of such a query consists of the clauses together with the found atom. So in order to test whether some clause C is subsumed by a clause in an indexed clause store N, one picks a literal from C that has a low probability of being subsumed, searches the index for generalizations of that literal and then tests C and the found clauses for subsumption. Since the query and the result literal are already found, using appropriate data structures of SPASS it is sufficient to test the clauses without these literals. We use a more general subsumption test with the possibility to hide at least one literal in each clause and we are able to keep the bindings of an indexing query result. This extended test is also needed for the reduction rules matching replacement resolution (Definition 4.20) and non-unit rewriting (Definition 4.21).

4.17. DEFINITION *(Condensation).* The reduction

$$\mathcal{R}\frac{\Theta_1 \,\|\, \Gamma_1 \rightarrow \Delta_1}{\Theta_2 \,\|\, \Gamma_2 \rightarrow \Delta_2}$$

where $\Theta_2 \,\|\, \Gamma_2 \rightarrow \Delta_2$ subsumes $\Theta_1 \,\|\, \Gamma_1 \rightarrow \Delta_1$ and $\Theta_2 \,\|\, \Gamma_2 \rightarrow \Delta_2$ is derived from $\Theta_1 \,\|\, \Gamma_1 \rightarrow \Delta_1$ by instantiation and (exhaustive) application of trivial literal elimination is called *condensation*.

Condensation is activated by the `-RCon` option.

In the literature condensation is often defined on the basis of factoring applications. From an implementation point of view the above definition is much sharper, because it only suggests matchers to generate duplicate literals that can be eventually removed, not unifiers as suggested by a definition based on factoring. All these candidate instantiation substitutions can be effectively computed by subsequently searching for matchers σ such that $E_1\sigma = E_2$ for $E_1, E_2 \in \Gamma_1$ (respectively for Θ_1, Δ_1) and then testing whether $(\Theta_1 \,\|\, \Gamma_1 \setminus \{E_2\} \rightarrow \Delta_1)\sigma$ subsumes $\Theta_1 \,\|\, \Gamma_1 \rightarrow \Delta_1$. This idea leads to a procedure that is more efficient than the factoring based algorithm suggested by Joyner Jr. [1976] and related to the techniques presented by Gottlob and Fermüller [1993].

4.18. DEFINITION *(Tautology Deletion).* The reduction

$$\mathcal{R}\frac{\Theta \,\|\, \Gamma \to \Delta}{}$$

where $\models \Theta \,\|\, \Gamma \to \Delta$ is called *tautology deletion*.

The above rule is sometimes also called semantic tautology deletion, since it is based on a semantic tautology test. This test corresponds to testing unsatisfiability of a set of ground literals. If we keep in mind that any literal can be coded as an (dis)equation, in order to test unsatisfiability of a set of ground literals it is sufficient to test congruence closure with respect to the positive equations. This can be done in polynomial time [Downey, Sethi and Tarjan 1980]. There are certain weaker syntactic conditions that can be checked in linear time:

$$\mathcal{R}\frac{\Theta \,\|\, \Gamma, E \to \Delta, E}{}$$

or

$$\mathcal{R}\frac{\Theta, A \,\|\, \Gamma \to \Delta, A}{}$$

or

$$\mathcal{R}\frac{\Theta \,\|\, \Gamma \to \Delta, t \approx t}{}$$

In SPASS the syntactic as well as the semantic check is implemented (see Appendix A). For the semantic test we adopted the algorithm presented by Downey et al. [1980] to our data structures. These conditions are implemented by nearly all todays theorem provers. The semantic check requires appropriate data structures for an efficient implementation. It is contained in the provers E, Saturate and SPASS.

Tautology deletion is activated by the `-RTaut` option. If the option is set to `1` only syntactic tautologies are deleted and if the option is set to `2` also the semantic test is performed.

4.19. DEFINITION *(Conflict).* The reduction

$$\mathcal{R}\frac{\Theta_1 \,\|\, \Gamma_1 \to \Delta_1 \quad \ldots \quad \Theta_n \,\|\, \Gamma_n \to \Delta_n}{\square}$$

where $\Theta_1 \,\|\, \Gamma_1 \to \Delta_1, \ldots, \Theta_n \,\|\, \Gamma_n \to \Delta_n \models \square$ is called *conflict*.

Even if n is fixed, the rule conflict is not effective, in general. It basically solves the general unsatisfiability problem of first-order logic. The rule sort simplification is an effective instance of this rule. Two further effective instantiations of this rule that are not related to specific theories are implemented in todays provers: *unit conflict* and the *terminator* [Antoniou and Ohlbach 1983]. [8] The former is the rule

$$\mathcal{R}\frac{\| \to E_1 \qquad \| E_2 \to}{\square}$$

[8] Hasta la vista baby!

such that E_1 and E_2 are unifiable. It seems that this rule is superfluous since it only detects a contradiction between two unit clauses. However, since we consider unification between E_1 and E_2, this rule cannot be simulated by, for example, matching replacement resolution (see Definition 4.20), but only by a resolution step. In the context of problems where the majority of generated clauses are units (e.g., unit equational problems or condensed detachment problems [McCune and Wos 1992]) the probability that both clauses are selected for inferences can become arbitrarily low. Then it can pay off to add this reduction rule that implements a (global) one step search for the empty clause.

The terminator is a generalization of unit conflict and a restriction of the general conflict rule to at most k non-unit clauses out of the n clauses, k fixed. For some given, finite set of clauses it is decidable whether we can derive the empty clause by resolution, if any derivation is restricted to contain at most k non-unit clauses. This is easy to see, since there are only finitely many different derivations using k non-unit clauses and resolving with a unit clause strictly reduces the length of the resolvent compared to the maximal length of one of its parent clauses. That's the terminator. In practice the terminator can be useful with values $k \leq 3$. Larger values rarely make sense, since the number of clauses that have to be considered for this rule grows exponentially in k times the length of the non-unit clauses. Note that if the terminator is applied to a Horn clause store without equality, it can be turned into a complete refutation procedure by subsequently increasing n.

As an exception from all other reduction rules, in practice the terminator is integrated in the search procedure like an inference rule, not like a reduction rule. It is too expensive to apply the terminator to all newly generated clauses and often it does not pay off. So the terminator is solely applied to the selected $Given$ clauses, if it is activated.

Unit conflict is activated by the `-RUnC` option. The terminator is activated by the `-RTer=`<*n*> option, where *n* specifies the number of considered non-unit clauses.

4.20. DEFINITION *(Matching Replacement Resolution).* The reductions

$$\mathcal{R}\frac{\Theta_1 \,\|\, \Gamma_1 \rightarrow \Delta_1, E_1 \quad \Theta_2 \,\|\, \Gamma_2, E_2 \rightarrow \Delta_2}{\begin{array}{c}\Theta_1 \,\|\, \Gamma_1 \rightarrow \Delta_1, E_1 \\ \Theta_2 \,\|\, \Gamma_2 \rightarrow \Delta_2\end{array}}$$

and

$$\mathcal{R}\frac{\Theta_1 \,\|\, \Gamma_1, E_1 \rightarrow \Delta_1 \quad \Theta_2 \,\|\, \Gamma_2 \rightarrow \Delta_2, E_2}{\begin{array}{c}\Theta_1 \,\|\, \Gamma_1, E_1 \rightarrow \Delta_1 \\ \Theta_2 \,\|\, \Gamma_2 \rightarrow \Delta_2\end{array}}$$

and

$$\mathcal{R}\frac{\Theta_1 \,\|\, \Gamma_1 \rightarrow \Delta_1, A_1 \quad \Theta_2, A_2 \,\|\, \Gamma_2 \rightarrow \Delta_2}{\begin{array}{c}\Theta_1 \,\|\, \Gamma_1 \rightarrow \Delta_1, A_1 \\ \Theta_2 \,\|\, \Gamma_2 \rightarrow \Delta_2\end{array}}$$

where (i) $E_1\sigma = E_2$ ($A_1\sigma = A_2$ for the third variant), (ii) $\Theta_1\sigma \subseteq \Theta_2$, $\Gamma_1\sigma \subseteq \Gamma_2$, $\Delta_1\sigma \subseteq \Delta_2$ are called *matching replacement resolutions.*

Matching replacement resolution is activated by the `-RFMRR` and `-RBMRR` option, for the forward and backward direction, respectively.

Matching replacement resolution is a restricted variant of replacement resolution, itself a restricted form of resolution where the conclusion must subsume one of its parent clauses. For matching replacement resolution we restrict the unifier of the complementary literals computed for replacement resolution to be a matcher. This speeds up the applicability test significantly.

The third variant of the rule that applies to the sort constraint cannot be simulated by sort simplification (Definition 4.5), because it also considers clauses that are not declarations. On the other hand, matching replacement resolution can also not simulate sort simplification. Consider the clauses

$$\begin{aligned} T(x), S(f(x)), \Theta_1 \,\|\, \Gamma_1 &\rightarrow \Delta_1 \\ R(x) \,\|\, &\rightarrow S(f(x)) \\ T(x) \,\|\, &\rightarrow R(x) \end{aligned}$$

The negative occurrence of $S(f(x))$ in the first clause cannot be eliminated by matching replacement resolution but by sort simplification.

4.21. DEFINITION *(Non-Unit Rewriting).* The reductions

$$\mathcal{R}\frac{\Theta_1 \,\|\, \Gamma_1 \rightarrow \Delta_1, s \approx t \quad \Theta_2 \,\|\, \Gamma_2, E[s']_p \rightarrow \Delta_2}{\begin{array}{c}\Theta_1 \,\|\, \Gamma_1 \rightarrow \Delta_1, s \approx t \\ \Theta_2 \,\|\, \Gamma_2, E[p/t\sigma] \rightarrow \Delta_2\end{array}}$$

and

$$\mathcal{R}\frac{\Theta_1 \,\|\, \Gamma_1 \rightarrow \Delta_1, s \approx t \quad \Theta_2 \,\|\, \Gamma_2 \rightarrow \Delta_2, E[s']_p}{\begin{array}{c}\Theta_1 \,\|\, \Gamma_1 \rightarrow \Delta_1, s \approx t \\ \Theta_2 \,\|\, \Gamma_2 \rightarrow \Delta_2, E[p/t\sigma]\end{array}}$$

where (i) $s\sigma = s'$, (ii) $s \succ t$, (iii) $\Theta_1\sigma \subseteq \Theta_2$, $\Gamma_1\sigma \subseteq \Gamma_2$, $\Delta_1\sigma \subseteq \Delta_2$ are called *non-unit rewriting*.

Non-unit rewriting and unit rewritung (see below) are activated by the `-RFRew` and `-RBRew` option, for the forward and backward direction, respectively.

The ordering restrictions for non-unit rewriting are a priori ordering restrictions, i.e., we do not compare the terms s and t with respect to the found matcher σ. The ordering test with respect to σ is sharper, but an efficient implementation of this check is non-trivial because it requires a tight connection between indexing, ordering computation and subsumption. Therefore, SPASS uses the a priori ordering check, i.e., SPASS verifies $s \succ t$. See also the discussion on page 24.

4.22. DEFINITION *(Unit Rewriting).* The reductions

$$\mathcal{R}\frac{\| \rightarrow s \approx t \quad \| E[s']_p \rightarrow}{\begin{array}{c}\| \rightarrow s \approx t \\ \| E[p/t\sigma] \rightarrow\end{array}}$$

and

$$\mathcal{R}\frac{\| \rightarrow s \approx t \quad \| \rightarrow E[s']_p}{\begin{array}{c}\| \rightarrow s \approx t \\ \| \rightarrow E[p/t\sigma]\end{array}}$$

where (i) $s\sigma = s'$, (ii) $s\sigma \succ t\sigma$, are called *unit rewriting*.

Unit rewriting is an instance of the second version of non-unit rewriting where all Θ_i, Γ_i, Δ_i are empty. We mention it here explicitely, because it is the style of rewriting used in purely equational completion, a theorem proving discipline of its own. Furthermore, the a posteriori ordering check is much easier to implement, because we need no subsumption check. Actually, it is implemented in SPASS.

In practice the rewriting reductions are among the most expensive reductions. Note that any subterm of any clause has to be considered and that subsequent rewriting steps to the same clause are common. Therefore, many provers don't use the full power of non-unit rewriting, but restrict the left clause to be a positive unit equation. They reduce non-unit clauses by positive unit equations.

Even the a posteriori check, condition (ii), can be further refined. Consider an equation where the left and right hand side don't share any variables. Then the a posteriori check will typically fail but may succeed by appropriate further instantiations. For example the equation $f(x, y) \approx g(z)$ cannot be oriented and hence the equation $f(a, b) \approx a$ cannot be rewritten by unit rewriting using the first equation. Now assume a RPOS with precedence $f > g > a > b$. Then the equation $f(a, b) \approx g(z)$ (the result of matching $f(x, y)$ with $f(a, b)$) can be turned into an oriented equation by instantiating z with a or b. This enables rewriting of $f(a, b)$ to $g(a)$ or $g(b)$. In general it is sufficient to consider the minimal constant and the crucial extra variables for further instantiation. Note also that the equation $f(x, y) \approx g(z)$ subsumes an equation like $f(x, y) \approx g(y)$ that is oriented and can therefore be used for rewriting in a straightforward way. This refinement is not implemented in SPASS Version 2.0 but remains an option for further releases.

Another way to solve the problem of unorientable equations because of extra variables is to split equations. Given some equation $s \approx t$ where $vars(s) \not\subseteq vars(t)$ and $vars(t) \not\subseteq vars(s)$, we introduce a new function symbol h where the arity of h is exactly $|vars(s) \cap vars(t)|$. If $\{x_1, \ldots, x_n\} = vars(s) \cap vars(t)$ then the equation $s \approx t$ is replaced by the equations $s \approx h(x_1, \ldots, x_n)$ and $t \approx h(x_1, \ldots, x_n)$. Given a KBO or RPOS and a precedence where the new symbol h is smaller than the top symbols of s and t, both introduced equations are oriented from left to right. In order to obtain a complete calculus that includes splitting of equations splitting must not be applied infinitely many times. Splitting equations is not implemented in SPASS Version 2.0.

4.23. DEFINITION *(Contextual Rewriting).* The reductions

$$\mathcal{R}\frac{\Theta_1 \,\|\, \Gamma_1 \rightarrow \Delta_1, s \approx t \quad \Theta_2 \,\|\, \Gamma_2, E[s']_p \rightarrow \Delta_2}{\begin{array}{c}\Theta_1 \,\|\, \Gamma_1 \rightarrow \Delta_1, s \approx t \\ \Theta_2 \,\|\, \Gamma_2, E[p/t\sigma] \rightarrow \Delta_2\end{array}}$$

and

$$\mathcal{R}\frac{\Theta_1 \,\|\, \Gamma_1 \to \Delta_1, s \approx t \quad \Theta_2 \,\|\, \Gamma_2 \to \Delta_2, E[s']_p}{\begin{array}{c}\Theta_1 \,\|\, \Gamma_1 \to \Delta_1, s \approx t \\ \Theta_2 \,\|\, \Gamma_2 \to \Delta_2, E[p/t\sigma]\end{array}}$$

where (i) $s\sigma = s'$, (ii) $s \succ t$, (iii) $s \approx t$ is strictly maximal in $\Theta_1 \,\|\, \Gamma_1 \to \Delta_1, s \approx t$, (iv) for any term t' in $\Theta_1 \,\|\, \Gamma_1 \to \Delta_1, s \approx t$, $s \succ t'$, (v) if $E[s']_p$ does not occur negatively, $s\sigma \approx t\sigma \prec E[s']_p$, (vi) $\models \mathit{fred}(\Theta_2 \to S(t'))$ for all sort atoms $S(t') \in \mathit{sortsimp}(\Theta_1\sigma)$, (vii) $\models \mathit{fred}(\Gamma_2 \to E)$ for all atoms $E \in \Gamma_1\sigma$, (viii) $\models \mathit{fred}(E \to \Delta_2)$ for all atoms $E \in \Delta_1\sigma$ are called *contextual rewriting*.

The expression $\mathit{sortsimp}(\Theta_1\sigma)$ denotes the sort constraint $\Theta_1\sigma$ after exhaustive application of the rule sort simplification. The function fred, see Table 5, (recursively) applies the reduction rules to the constructed subclauses before they are checked as tautologies. The performance of fred, i.e. which rules are actually tested/applied, determines the strength as well as the cost for testing and applying contectual rewriting. Our current idea is to use all reduction rules except contextual rewriting. We do not test/apply contextual rewriting recursively. Contextual rewriting is not implemented in SPASS Version 2.0 but will be included in the next release. We already have a prototype implementation of the rule.

The final reduction rule exploits particular equations of the form $x \approx t$ (called assignment equations), where x does not occur in t nor in the rest of the clause. Negative equations of this form can simply be removed from a clause. In order to remove positive assignment equations the domain structure shared by any model of the current clause store has to be examined. In particular, we exploit the case that the domain is non-trivial. Therefore, before this rule can be applied, certain properties of any clause store model must be checked. This can, e.g., be done by a sufficient criterion that can be tested syntactically.

4.24. DEFINITION *(Assignment Equation Deletion).* Let N be the current clause store. The reductions

$$\mathcal{R}\frac{\Theta \,\|\, x \approx t, \Gamma \to \Delta}{\Theta \,\|\, \Gamma \to \Delta}$$

and

$$\mathcal{R}\frac{\Theta \,\|\, \Gamma \to x \approx t, \Delta}{\Theta \,\|\, \Gamma \to \Delta}$$

where for both variations of the rule we assume (i) $x \notin \mathit{vars}(t)$, (ii) $x \notin \mathit{vars}(\Theta \,\|\, \Gamma \to \Delta)$ and for the second variant, where we remove a positive equation, we assume in addition that $|\mathcal{D}| > 1$ for any interpretation $\mathcal{M}$ with $\mathcal{M} \models N$, are called *assignment equation deletion*.

Assignment equation deletion is activated by the `-RAED` option. If `-RAED` is set to `1` only the first variant is applied, if set to `2` both variants are used.

For the elimination of the positive equation to be sound it is necessary to guarantee a non-trivial domain for any model of the current clause store. A syntactic condition is the existence of a clause $\| s \approx t \to$ where s and t are arbitrary. If such a clause is contained

in the clause store, the domain of any model is non-trivial and we can apply the second variant of the rule.

4.5. Splitting

The effect of a splitting rule application is not only to extend the current clause store but also to modify and extend the current clause store collection.

4.25. DEFINITION *(Splitting).* The inference

$$\mathcal{S}\frac{\Theta_1, \Theta_2 \parallel \Gamma_1, \Gamma_2 \rightarrow \Delta_1, \Delta_2}{\Theta_1 \parallel \Gamma_1 \rightarrow \Delta_1 \quad | \quad \Theta_2 \parallel \Gamma_2 \rightarrow \Delta_2}$$

where $vars(\Theta_1 \parallel \Gamma_1 \rightarrow \Delta_1) \cap vars(\Theta_2 \parallel \Gamma_2 \rightarrow \Delta_2) = \emptyset$ and $\Delta_1 \neq \emptyset$, $\Delta_2 \neq \emptyset$ is called *splitting*.

Splitting is activated by the `-Splits=`<*n*> option, where *n* specifies the overall number of splitting applications in a SPASS run. Unlimited application of the rule can be achieved by choosing -1 for *n*.

Without the condition that the two split clauses must not share variables, Splitting is very much like the β-rule of free variable tableau. Since the clauses don't share variables, the two cases are completely independent and the derived clauses can be used for simplification/reduction without any restriction. For example, both clauses subsume the parent clause.

In case the first split part is ground, i.e., $vars(\overline{S(t)} \parallel \overline{E} \rightarrow \overline{E'}) = \emptyset$, where $\overline{S(t)} = S_1(t_1), \ldots, S_n(t_n)$, $\overline{E} = E_1, \ldots, E_m$ and $\overline{E'} = E'_1, \ldots, E'_l$ and $1 \leq i \leq n$, $1 \leq j \leq m$, $1 \leq k \leq l$, it is very useful to add the negation of the first split clause to the second part

$$\mathcal{S}\frac{\overline{S(t)}, \Theta_2 \parallel \overline{E}, \Gamma_2 \rightarrow \overline{E'}, \Delta_2}{\overline{S(t)} \parallel \overline{E} \rightarrow \overline{E'} \;\Big|\; \begin{array}{l} \Theta_2 \parallel \Gamma_2 \rightarrow \Delta_2 \\ \quad \parallel \quad \rightarrow S_i(t_i) \\ \quad \parallel \quad \rightarrow E_j \\ \quad \parallel E'_k \rightarrow \end{array}}$$

All these additional unit clauses can help a lot in reducing the clause set of the second part (see matching replacement resolution, Definition 4.20 or non-unit rewriting, Definition 4.21). In a purely propositional setting, a calculus solely based on unit conflict (see Definition 4.19) and extended splitting can polynomially simulate truth tables, whereas a calculus based on unit conflict and the simple splitting rule cannot [D'Agostino 1992].

Of course, for any first part split clause we could add its negation to the second part. However, in general this leads to the introduction of new Skolem constants, and in practice this tends to extend the search space for the second part. Note that in this case the ground units resulting from the negated first part cannot be used for matching replacement resolutions, because the introduced Skolem constants are new. As an alternative one could

also record the ground instances of the variables in the first split clause used in the refutation of the first part and then add their negation as a disjunct to the second part. But it is questionable whether such an effort pays off in practice.

Splitting itself often tends to generate a huge search tree, so additional refinements are necessary. Therefore, we required that Δ_1 and Δ_2 are non-empty in Definition 4.25. So we only split non-Horn clauses into clauses having strictly less positive literals. The rationale behind this comes from the propositional level. For a set of propositional Horn clauses, satisfiability can be decided in linear time [Dowling and Gallier 1984], whereas satisfiability for arbitrary clauses is an *NP*-complete problem. The reduction rule matching replacement resolution (Definition 4.20) is also a decision procedure for propositional Horn clauses (although it results in a quadratic time implementation). Hence, non-Horn splitting and matching replacement resolution are a reasonable decision procedure for propositional clauses. In case a clause can be split into a propositional part (no variables) and a non-propositional one, it is very useful to split the clause that way and to add the negation of the propositional part to the second as indicated before.

An alternative to an explicit case analysis is to split clauses by the introduction of new propositional symbols. For example, the clause

$$S(x) \parallel f(x) \approx y \rightarrow Q(x,x), Q(a,z)$$

can be replaced by the clauses

$$\begin{array}{rr} S(x) \,||\, f(x) \approx y, A \rightarrow Q(x,x) \\ || \qquad\qquad B \rightarrow Q(a,z) \\ || \qquad\qquad \rightarrow A, B \end{array}$$

where A, B are new propositional symbols. The replacement preserves satisfiability of the current clause store and if it is only applied finitely many often during a proof attempt it also preserves completeness. If A, B are minimal in the ordering, no inference on A, B will be performed as long as other literals are contained in the respective clauses. So the different parts of the original clause don't interfere as long as they are completely resolved. This simulates splitting without the need to extend the notion of a clause store to a collection of clause stores and hence a less complicated implementation. The second advantage of this approach is that it does not introduce the inherent redundancy of an explicit splitting approach. The main disadvantage of this splitting style is that none of the generated clauses can be directly used for reductions, because the propositional variables A, B must be new. This splitting style is available in the provers Saturate and Vampire, explicit splitting (Definition 4.25) is available in SPASS.

5. Global Design Decisions

5.1. Main-Loop

The main-loop without splitting, Table 4, is a generalization of the main-loop introduced in Section 3, Table 1.

Compared to the simple, resolution-based prover, all inferences are computed in the extra function inf and (inter)reduction takes place in the function $ired$. In the automatic

1	PROVER(N)
2	$Wo := \emptyset$;
3	$Us := ired(N, N)$;
4	While $(Us \neq \emptyset$ and $\Box \notin Us)$ {
5	$(Given, Us) := choose(Us)$;
6	$Wo := Wo \cup \{Given\}$;
7	$New := inf(Given, Wo)$;
8	$(New, Wo, Us) := ired(New, Wo, Us)$;
9	}
10	If $(Us = \emptyset)$ then print "Completion Found";
11	If $(\Box \in Us)$ then print "Proof Found";

Table 4: The Overall Loop without Splitting

mode of SPASS, the inference rules applied in inf are chosen after an analysis of the input problem, such that the resultion calculus is sound and complete. For example, if the input problem contains no equality, the superposition/paramodulation rules are not activated or if the input problem is Horn, factoring rules are not needed. In Appendix A.1 we sum up all SPASS options needed to (de)activate the inference/reduction rules introduced in Section 4. Please note that a manual setting setting of inference/reduction rules can result in an unsound or incomplete calculus. SPASS does not verify your manual settings.

The combination of the reduction rules gets more subtle compared to the combination of subsumption/tautology deletion presented in Section 3. The function $ired$ serves this purpose, Table 6. Please recall that the terminator is integrated like an inference rule and hence does not show up. First, line 4, any newly derived clause is forward reduced with respect to the sets Wo and Us. The function $fred$ is presented in detail in Table 5. The ordering of the tested forward reductions is determined by potential dependencies between the rules and by their respective implementation costs.

We don't consider the lazy reduction approach introduced in Section 3, Table 3. It is a bit tricky but not too difficult to develop it out of the presented full reduction algorithms. Inside the algorithms a redundant clause is not always directly deleted, but represented by the constant $\top$.

In practice, tautology deletion (Table 5, line 2), elimination of trivial literals (line 4), condensation (line 5) and assignment equation deletion (line 6) are cheap operations, because only the derived clause has to be considered for testing their applicability. This is not completely true for the assignment equation deletion (see Definition 4.24), but the suggested syntactic domain size criterion can be tested once at the beginning of the search process, so no extra effort is necessary. Clauses that pass these tests, are checked for forward sub-

```
 1 fred(Given, Wo, Us)
 2   Given  := taut(Given);
 3   If (Given = ⊤) then return(⊤);
 4   Given  := obv(Given);
 5   Given  := cond(Given);
 6   Given  := aed(Given, Wo, Us);
 7   If (fsub(Given, Wo, Us)) then return(⊤);
 8   (Hit, Given)  := frew(Given, Wo, Us);
 9   If (Hit) then {
10     Given  := taut(Given)
11     If (Given = ⊤) then return(⊤);
12     Given  := obv(Given);
13     Given  := cond(Given);
14     If (fsub(Given, Wo, Us)) then return(⊤);
15   }
16   Given  := ssi(Given, Wo, Us);
17   Given  := fmrr(Given, Wo, Us);
18   Given  := unc(Given, Wo, Us);
19   Given  := sst(Given, Wo, Us);
20   return(Given);
```

Table 5: Forward Reduction

```
 1 ired(New, Wo, Us)
 2   While (New ≠ ∅)   {
 3     (Given, New) := choose(New);
 4     Given        := fred(Given, Wo, Us);
 5     If (Given ≠ ⊤) then {
 6       (New, Wo, Us)   := bsub(Given, New, Wo, Us);
 7       (New, Wo, Us)   := bmrr(Given, New, Wo, Us);
 8       (New, Wo, Us)   := brew(Given, New, Wo, Us);
 9       Us              := Us ∪ {Given};
10       }
11     }
12   return(∅, Wo, Us);
```

Table 6: Interreduction

sumption with respect to Wo and Us (line 7) and for forward rewriting (line 8) where *Hit* is set to true, if a rewriting step actually took place. If a rewriting step is performed, the rules tautology deletion, elimination of duplicate/trivial literals, condensation and forward subsumption are checked a second time (lines 10–14). Below is a simple example that demonstrates dependencies between the different reduction rules.

$$
\begin{array}{rr}
1: & \rightarrow f(x) \approx x \\
2: & \rightarrow a \approx b \\
3: & P(f(x)) \rightarrow P(x) \\
4: & P(f(x)), P(b) \rightarrow \\
5: & P(a), P(c) \rightarrow \\
6: & P(g(f(x))), P(g(x)) \rightarrow
\end{array}
$$

The clauses 3–6 are completely interreduced with respect to the reductions presented in Section 4. Rewriting with clause 1 into clause 3 generates a syntactic tautology, rewriting with clause 1 into clause 4 enables a further condensation step on clause 4 resulting in $4'$:$P(b) \rightarrow$ and rewriting clause 5 with clause 2 produces $5'$:$P(b), P(c) \rightarrow$ (assuming $a \succ b$) that is forward subsumed by clause $4'$. After rewriting with clause 1, duplicate literal elimination can be applied to clause 6.

Finally the reductions sort simplification (line 16), forward clause reduction (line 17), unit conflict (line 18) and static soft typing are tested. These reduction rules don't enable further applications of other rules, because they either strictly reduce the number of literals or reduce the clause to $\top$ (static soft typing). In order to test the applicability of these rules the overall clause stores Wo and Us must be considered.

All clauses that pass forward reduction (Table 6, line 4) are used for back reduction (lines 6–8) and are finally added to the Us set (line 9). Backward subsumption (line 6) deletes all clauses from Wo and Us that are subsumed by $Given$. Backward matching replacement resolution (line 7) tests all clauses in Wo and Us for matching replacement resolution with $Given$. Reduced clauses are always deleted from their respective source set and added to New. Back rewriting (line 8) behaves the same, but tests rewriting. The clauses in New are not directly tested for all these reduction rules. They are tested after having entered the Us set. This is motivated by efficiency issues which we will discuss below.

The function $choose$ (Table 6, line 3) selects a clause with the smallest number of symbols. Small clauses have a higher probability to reduce other clauses. For example, a subsuming clause must have fewer symbols (consider the discussion after Definition 4.16) than the clause it subsumes. For many other reductions like rewriting, selecting small clauses is still a good heuristic because the size ordering of terms is often included in the reduction ordering.

Note that a clause can be selected several times as $Given$ clause in the while-loop of the interreduction algorithm, if it is successfully reduced several times. Selecting small $Given$ clauses tries to minimize the number of such situations.

The main-loop presented in Table 7 extends the already discussed main-loop of Table 4 with splitting. Whether SPASS applies splitting or not depends on the input problem[9] and can be controlled by the splitting option (see Appendix A.2). In case splitting is not applied, executing the main-loop in Table 7 or Table 4 results in exactly the same behavior.

If splitting is possible (Table 7, line 10) it is preferred over all other inferences. The rationale behind this decision is that a splitting application results in a strictly smaller clause store collection. The exploration of the binary tree generated by the splitting rule is performed by a standard depth-first, backtracking search (lines 7, 11, 12). Compared to the SPASS Versions 1.0.x, Version 2.0 has a more sophisticated splitting clause selection. It selects the clause with the highest unit reduction potential after the split.

All functions implementing inference/reduction operations have to be refined and must also consider the *split level* of a clause. Initially, all clauses have split level zero and the current split level is zero. Then any clause generated by a splitting inference gets the current split level plus one as its split level and the current split level is incremented. Clauses generated by all other inference/reduction rules get the maximal split level of their parent clauses as their new split level. Backtracking resets the current split level to the split level of the activated branch. But what happens if a clause C is now subsumed by a clause D with a greater split level? We must not delete C, but only remove it from the current clause store, store it at D's split level on the split stack and reinsert it if backtracking considers that level. Clauses that are rewritten or reduced by clauses with a higher split level must be copied and also kept appropriately on the split stack.

Basically that is all to integrate splitting into a saturation based prover. Nevertheless, some refinements are possible. First, since all clauses have a split level, also the empty clause has a split level. This level indicates where backtracking should start to consider open branches and all branches at a higher split level can be discarded. For example, if we

[9] SPASS only splits non-Horn clauses, see Definition 4.25.

```
1 PROVER(N)
2   Wo      := ∅;
3   Us      := ired(N, N);
4   Stack   := emptystack();
5   While (Us ≠ ∅ and (□ ∉ Us or not stackempty(Stack)))   {
6     If (□ ∈ Us) then
7       (Stack, Wo, Us)  := backtrack(Stack, Wo, Us);
8     else {
9       (Given, Us)     := choose(Us);
10      If (splittable(Given)) then {
11        New  := firstsplitcase(Given);
12        Stack := push(Stack, secondsplitcase(Given));
13      }
14      else {
15        Wo   := Wo ∪ {Given};
16        New  := inf(Given, Wo);
17      }
18      (New, Wo, Us)    := ired(New, Wo, Us);
19    }
20  }
21  If (Us = ∅) then print "Completion Found";
22  If (□ ∈ Us) then print "Proof Found";
```

Table 7: The Overall Loop with Splitting

derive the empty clause at split level zero, then we can immediately stop and don't have to consider any further possibly open branches. Second, if we don't only store the split level with each clause but also a bit array with length of the split level, the following improvement is possible. The bit array is updated together with the split level and indicates every level that contributed to the clause. If we now derive an empty clause at some split level and detect that it does not depend on some earlier levels above the previous backtracking level that have open branches left, we can erase these levels, their split clauses and all clauses depending on these. We call this operation *branch condensation* and it is indispensable to make splitting feasible in practice. In the AI literature branch condensation it often referred to as dependency directed backtracking.

In Section 3 we also introduced a main-loop with lazy reduction, Table 3. Although we did not present it here, lazy reduction is also possible with the extended inference rule set and it is not too difficult to think of lazy extensions of the main-loops according to Table 3. Therefore, we omitted an extra presentation here.

5.2. Proof Documentation/Checking

By default, SPASS does not output a proof in case it derives the (final) empty clause nor does SPASS provide a final saturated set of clauses, in case all possible inferences have been performed without finding an empty clause. This can be changed by activating the proof documentation option (see Appendix A.1).

Proof documentation is possible by implicitly or explicitly storing all clauses during the overall search process that might contribute to a proof. As a consequence, a run with proof documentation has a higher memory consumption and thus causes the prover to slow down. This effect is further supported by splitting applications, where all clauses from all significant branches must be kept as well. Therefore, in favor of execution speed, by default SPASS does not output proofs (saturated clause stores). Nevertheless, SPASS can handle proofs of several hundred thousand steps in reasonable time.

Automated proof checking is a very important topic in any theorem proving project. The inference/reduction rules are non-trivial to (efficiently) implement, so there is a high potential for bugs. This is also shown by several a posteriori disqualifications at the CASC theorem proving competitions happened so far [Sutcliffe and Suttner 1999]. Proofs of automated theorem provers cannot be checked by hand in practice. So there is a need for automated proof checking. Our solution is a separately implemented proof checker. The checker takes a proof and starts with an analysis of the splitting rule applications. The checker generates the binary tree resulting from subsequent splitting inference applications and tests whether all branches contain an empty clause, whether the split level assignments are done correctly and whether the splitting inference rule is applied in a sound way. Then the checker generates for every inference/reduction rule application the corresponding first-order theorem proving problem and provides it for a separate prover. The single step proof problems can typically be easily solved if they are correct. This way, it is possible to validate proofs up to several hundred thousand steps in reasonable time and that completely automatically. As such a proof checker solely relies on logical implication, it supports most of today's saturation-based inference systems and is robust against

modifications to inference/reduction rules.

5.3. Data Structures and Algorithms

In Section 3 we discussed a simple prover based on resolution. In Section 5.1 we extended this prover to cope with the inference/reduction rules of SPASS. For our simple resolution-based prover we already argued that

- the Us set grows very fast,
- reductions are indispensable to reduce the number of clauses in the Us set
- most of the time is spent with reductions.

The situation is getting even more dramatic if we consider the inference rules for equality introduced in Section 4 and the suggested main-loops (Table 4, Table 7). For example, if the selected $Given$ clause C in the main-loop contains a positive equation, then any non-variable subterm of any literal of a clause in Wo that unifies with the left or right hand side of the equation generates a new clause by paramodulation. If the considered left hand side is a variable then any subterm of any different clause unifies with the variable and produces a new clause via paramodulation. An example for such a clause is one that forces a finite, two element domain: $\rightarrow x \approx a, x \approx b$. Ordering restrictions improve the situation (not for the finite domain clause), but we nevertheless have to find reasonable ways to store large Us sets and to efficiently find reduction/inference partners.

There are several solutions to these problems. We now focus on one solution and discuss alternatives at the end of this section. The first design decision in SPASS is to store all atoms in Wo, Us in a shared way, respectively. That means every occurrence of any subterm of an atom in Us (Wo) exists exactly once and is shared by all superterms containing this subterm. The idea is to save space and to keep indexing structures small. As all terms are shared, any subterm is only submitted once to the indexing structure that provides retrieval for inferences/reductions. This works fine for ground terms, but in general clauses contain variables that are considered different if they occur in different clauses. Therefore, almost nothing can be shared between two different non-ground clauses. The solution to this problem is our second design decision that is to *normalize* variables of clauses. For any clause, if the clause is considered from left to right as a sequence of its literals, the variables are named with respect to their occurrence according to a fixed variable sequence. After normalization there is a high probability that clauses with variables share non-ground subterms. This is confirmed by experiments.

A consequence of this decision is that algorithms for unification/matching/generalization have to keep track of this fact. For example, we have variants of the unification algorithm that use two substitutions (which are called *contexts* inside the unification algorithm) in order to store bindings between variables for two terms stemming from two different clauses in an appropriate way.[10]

Putting newly generated clauses into a sharing/indexing data structure is extra effort. Since newly generated clauses have a high probability of being subsumed or reduced by already existing clauses, the New set (Table 4, 7, 6) is kept unshared. Furthermore, reductions

[10] See the discussion in Section 3.

are (by definition) destructive, so they can only be efficiently applied to unshared clauses. Shared clauses have to be extracted/copied from the sharing structure before modification, because destructive manipulation of shared terms may also effect other clauses where the considered reduction is not permitted. So any forward reduction to the $Given$ clause inside the interreduction algorithm (Table 6) is done destructively, but before a back reduction operation can actually be performed (lines 6–8) the clause has to be extracted/copied from the Wo, Us sharing structure first. As it is unshared afterwards, it can be moved to the New set. At line 9 of the $ired$ function (Table 6), the $Given$ clause is inserted into the sharing/indexing structures of the Us set.

An alternative solution not implemented in SPASS is to completely abstract from variable positions, e.g., by introducing one dummy variable for all occurring variables. We build one or several atom/term trees that represent all atoms/terms without considering variables to be different. These trees are then linked to the real atoms/terms and efficient algorithms can be devised to search for candidate atoms/terms out of such skeleton trees. Atoms/terms found this way are still candidates, because of the variable abstraction. They have then to be verified by the appropriate matching/unification test. The so called discrimination trees support such an approach.

If we resign from complete interreduction and focus on lazy reduction (see Table 3), no reduction rule needs to be tested with respect to the Us set. So inserting the Us clauses into an indexing structure for access is not needed. The Us clauses are only needed to provide a pool from which the next $Given$ clause is selected. To this end, we only need the necessary information for the $choose$ function and the clause itself. If the $choose$ function relies on size, the necessary information is simply a number. Since all Us clauses are children of two Wo clauses, instead of storing the clause, we store the numbers of the parents and the way it was generated represented by, say, one extra number. So, every clause in the Us set can be represented by four numbers, in practice in constant space. This results in a huge reduction of memory consumption and hence in an increase of execution speed. The Waldmeister system (see Appendix C) can treat the Us set this way. Note that the extra time needed to generate the $Given$ clause can be neglected. In case a parent of a selected $Given$ clause is no longer in the Wo set, it must have become redundant, hence the $Given$ clause is redundant as well and needs not to be considered.

A further possibility to restrict the number of clauses in the Us set is to simply throw away clauses. This may cause incompleteness of the theorem prover. Such techniques are available in Fiesta, Otter, SPASS and Vampire, see Appendix C and the discussion in Section 3.

Acknowledgments

Knowledge about the design of automated theorem provers is mostly distributed by discussions among the authors of such systems. I want to thank Bill McCune the author of Otter that is the father system of all today's "modern" automated saturation based theorem provers. We learned a lot about the implementation of theorem provers by inspecting Otter. My colleagues Arnim Buch, Thomas Hillenbrand, Bernd Löchner, authors of Waldmeister, Jörg Denzinger, author of Discount, Hans de Nivelle, author of Bliksem, Tanel Tammet,

author of Gandalf, Stephan Schulz, author of E, Harald Ganzinger, author of Saturate, Andrei Voronkov, co-author of Vampire, Robert Nieuwenhuis, author of Fiesta (and Saturate) contributed a lot to this chapter.

As mentioned in the introduction the development of a competative theorem prover is a challenging software project, exemplified in the following for the SPASS theorem prover: Although there existed some preliminary versions of SPASS before 1994, the first version called SPASS was started in that year and was finished in 1995 by Bernd Gaede in the context of his diploma thesis. This version already relied on a library of data structures we called EARL.[11] The library already contained indexing support and was developed by Peter Graf and Christoph Meyer and myself. Clause normal form translation was added to SPASS by Georg Rock as a diploma project. Further development of SPASS took place in paid student projects that typically lasted for several months each. Christian Cohrs introduced splitting to SPASS, Enno Keen was responsible for inference rules and parsing support, Thorsten Engel wrote our proof checker, Dalibor Topic significantly improved our memory management module and contributed to the implementation of reduction rules and Christian Theobalt wrote a whole bunch of documentation support scripts and mastered the challenge to port SPASS to the Windows world. As a prerequisite he developed a neat graphical user interface. Bijan Afshordel contributed to reductions on the formula level and programmed the atom definition module, Uwe Brahm was indispensable for putting SPASS on the Web and Christof Brinker added the most recent development, the detection and deletion of non-syntactic tautologies. Thanks to all of them.

Finally, I'm indebted to Thomas Hillenbrand, Enno Keen, Andreas Nonnengart and Andrei Voronkov for many comments on this chapter that lead to significant improvements.

Bibliography

ANTONIOU G. AND OHLBACH H. J. [1983], Terminator, *in* A. Bundy, ed., 'Proceedings of 8th International Joint Conference on Artificial Intelligence, IJCAI-83', pp. 916–919.

BAADER F. AND NIPKOW T. [1998], *Term Rewriting and All That*, Cambridge University Press.

BACHMAIR L. AND GANZINGER H. [1994], 'Rewrite-based equational theorem proving with selection and simplification', *Journal of Logic and Computation* **4**(3), 217–247. Revised version of Max-Planck-Institut für Informatik technical report, MPI-I-91-208, 1991.

BACHMAIR L. AND GANZINGER H. [2001], Resolution theorem proving, *in* A. Robinson and A. Voronkov, eds, 'Handbook of Automated Reasoning', Elsevier.

BACHMAIR L., GANZINGER H. AND WALDMANN U. [1993], Superposition with simplification as a decision procedure for the monadic class with equality, *in* G. Gottlob, A. Leitsch and D. Mundici, eds, 'Computational Logic and Proof Theory, Third Kurt Gödel Colloquium', Vol. 713 of *LNCS*, Springer, pp. 83–96.

BENANAV D. [1990], Simultaneous paramodulation, *in* M. E. Stickel, ed., 'Proceedings of the 10th International Conference on Automated Deduction', Vol. 449 of *LNAI*, Springer, pp. 442–455.

CHANG C.-L. AND LEE R. C.-T. [1973], *Symbolic Logic and Mechanical Theorem Proving*, Computer Science and Applied Mathematics, Academic Press.

CHARATONIK W., MCALLESTER D., NIWINSKI D., PODELSKI A. AND WALUKIEWICZ I. [1998], The horn mu-calculus, *in* 'Proceedings 13th IEEE Symposium on Logic in Computer Science, LICS'98', IEEE Computer Society Press, pp. 58–69.

[11] Efficient Automated Reasoning Library

D'AGOSTINO M. [1992], 'Are tableaux an improvement on truth-tables?', *Journal of Logic, Language and Information* **1**, 235–252.

DAVIS M. AND PUTNAM H. [1960], 'A computing procedure for quantification theory', *Journal of the ACM* **7**, 201–215.

DERSHOWITZ N. [1982], 'Orderings for term-rewriting systems', *Theoretical Computer Science* **17**, 279–301.

DERSHOWITZ N. [1987], 'Termination of rewriting', *Journal of Symbolic Computation* **3**(1), 69–115.

DOWLING W. F. AND GALLIER J. H. [1984], 'Linear-time algorithms for testing the satisfiability of propositional horn formulae', *Journal of Logic Programming* **1**(3), 267–284.

DOWNEY P. J., SETHI R. AND TARJAN R. E. [1980], 'Variations on the common subexpression problem', *Journal of the ACM* **27**(4), 758–771.

EISINGER N. [1991], *Completeness, Confluence, and Related Properties of Clause Graph Resolution*, Research Notes in Artificial Intelligence, Pitman Ltd., London.

FISCHER B., SCHUMANN J. AND SNELTING G. [1998], Deduction-based software component retrieval, *in* W. Bibel and P. H. Schmitt, eds, 'Automated Deduction - A Basis for Applications', Vol. 3 of *Applied Logic*, Kluwer, chapter 11, pp. 265–292.

FRÜHWIRTH T., SHAPIRO E., VARDI M. Y. AND YARDENI E. [1991], Logic programs as types for logic programs, *in* A. R. Meyer, ed., 'Proceedings of the 6th Annual IEEE Symposium on Logic in Computer Science, LICS'91', IEEE Computer Society Press, pp. 300–309.

GANZINGER H., MEYER C. AND WEIDENBACH C. [1997], Soft typing for ordered resolution, *in* 'Proceedings of the 14th International Conference on Automated Deduction, CADE-14', Vol. 1249 of *LNAI*, Springer, Townsville, Australia, pp. 321–335.

GAREY M. R. AND JOHNSON D. S. [1979], *Computers and intractability : A guide to the theory of NP-completeness*, Mathematical Sciences Series, Freeman, New York.

GOTTLOB G. AND FERMÜLLER C. G. [1993], 'Removing redundancy from a clause', *Artificial Intelligence* **61**, 263–289.

GOTTLOB G. AND LEITSCH A. [1985], 'On the efficiency of subsumption algorithms', *Journal of the ACM* **32**(2), 280–295.

GRAF P. [1996], *Term Indexing*, Vol. 1053 of *LNAI*, Springer.

HÄHNLE R., KERBER M. AND WEIDENBACH C. [1996], Common syntax of the dfg-schwerpunktprogramm "deduktion", Interner Bericht 10/96, Universität Karlsruhe, Fakultät für Informatik, Germany. Current version available from `http://spass.mpi-sb.mpg.de/`.

HEINTZE, N. AND CLARKE, E., EDS [1999], *Workshop on Formal Methods and Security Protocols*, Self Publishing, Trento, Italy.

HILLENBRAND T., JAEGER A. AND LÖCHNER B. [1999], Waldmeister – improvements in performance and ease of use, *in* H. Ganzinger, ed., '16th International Conference on Automated Deduction, CADE-16', LNAI, Springer, pp. 232–236.

HUSTADT U. AND SCHMIDT R. A. [1997], On evaluating decision procedures for modal logics, *in* 'Proceedings of 15th International Joint Conference on Artificial Intelligence, IJCAI-97', pp. 202–207.

JACQUEMARD F., MEYER C. AND WEIDENBACH C. [1998], Unification in extensions of shallow equational theories, *in* T. Nipkow, ed., 'Rewriting Techniques and Applications, 9th International Conference, RTA-98', Vol. 1379 of *LNCS*, Springer, pp. 76–90.

JOYNER JR. W. H. [1976], 'Resolution strategies as decision procedures', *Journal of the ACM* **23**(3), 398–417.

KAUTZ H. AND SELMAN B. [1996], Pushing the envelope: Planning, propositional logic and stochastic search, *in* 'Proceedings of the 13th National Conference on AI, AAAI'96', Vol. 2, AAAI Press / MIT Press, pp. 1194–1201.

KNUTH D. E. AND BENDIX P. B. [1970], Simple word problems in universal algebras, *in* I. Leech, ed., 'Computational Problems in Abstract Algebra', Pergamon Press, pp. 263–297.

MCCUNE W. AND WOS L. [1992], Experiments in automated deduction with condensed detachment, *in* '11th International Conference on Automated Deduction, CADE-11', Vol. 607 of *LNCS*, Springer, pp. 209–223.

MCCUNE W. AND WOS L. [1997], 'Otter', *Journal of Automated Reasoning* **18**(2), 211–220.

MEYER C. [1999], Soft Typing for Clausal Inference Systems, Dissertation, Technische Fakultät der Universität des Saarlandes, Saarbrücken, Germany.

NIEUWENHUIS R. [1996], Basic paramodulation and decidable theories (extended abstract), *in* 'Proceedings 11th IEEE Symposium on Logic in Computer Science, LICS'96', IEEE Computer Society Press, pp. 473–482.

NIEUWENHUIS R. AND RUBIO A. [2001], Paramodulation-based theorem proving, *in* A. Robinson and A. Voronkov, eds, 'Handbook of Automated Reasoning', Elsevier.

NIVELA P. AND NIEUWENHUIS R. [1993], Saturation of first-order (constrained) clauses with the *Saturate* system, *in* C. Kirchner, ed., 'Rewriting Techniques and Applications, 5th International Conference, RTA-93', Vol. 690 of *Lecture Notes in Computer Science, LNCS*, Springer, Montreal, Canada, pp. 436–440.

NONNENGART A., ROCK G. AND WEIDENBACH C. [1998], On generating small clause normal forms, *in* C. Kirchner and H. Kirchner, eds, '15th International Conference on Automated Deduction, CADE-15', Vol. 1421 of *LNAI*, Springer, pp. 397–411.

NONNENGART A. AND WEIDENBACH C. [2001], Computing small clause normal forms, *in* A. Robinson and A. Voronkov, eds, 'Handbook of Automated Reasoning', Vol. 1, Elsevier, chapter 6, pp. 335–367.

PETERSON G. E. [1983], 'A technique for establishing completeness results in theorem proving with equality', *SIAM Journal of Computation* **12**(1), 82–100.

RIAZANOV A. AND VORONKOV A. [1999], Vampire, *in* H. Ganzinger, ed., '16th International Conference on Automated Deduction, CADE-16', Vol. 1632 of *LNAI*, Springer, pp. 292–296.

ROBINSON G. AND WOS L. [1969], Paramodulation and theorem-proving in first-order theories with equality, *in* B. Meltzer and D. Michie, eds, 'Machine Intelligence 4', pp. 135–150.

ROBINSON J. A. [1965], 'A machine-oriented logic based on the resolution principle', *Journal of the ACM* **12**(1), 23–41.

SCHMIDT-SCHAUSS M. [1988], 'Implication of clauses is undecidable', *Theoretical Computer Science* **59**, 287–296.

SCHULZ S. [1999], System abstract: E 0.3, *in* H. Ganzinger, ed., '16th International Conference on Automated Deduction, CADE-16', Vol. 1632 of *LNAI*, Springer, pp. 297–301.

SNYDER W. [1993], 'On the complexity of recursive path orderings', *Information Processing Letters* **46**, 257–262.

SOCHER R. [1988], A subsumption algorithm based on characteristic matrices, *in* E. Lusk and R. Overbeek, eds, '9th International Conference on Automated Deduction, CADE-9', Vol. 310 of *LNCS*, Springer, pp. 573–581.

STILLMAN R. B. [1973], 'The concept of weak substitution in theorem-proving', *Journal of the ACM* **20**(4), 648–667.

SUTCLIFFE G. AND SUTTNER C. B. [1998], 'The tptp problem library – cnf release v1.2.1', *Journal of Automated Reasoning* **21**(2), 177–203.

SUTCLIFFE G. AND SUTTNER C. B. [1999], 'The cade-15 atp system competition', *Journal of Automated Reasoning* **23**(1), 1–23.

TAMMET T. [1998], Towards efficient subsumption, *in* C. Kirchner and H. Kirchner, eds, '15th International Conference on Automated Deduction, CADE-15', Vol. 1421 of *LNAI*, Springer, pp. 427–441.

WEIDENBACH C. [1996], Computational Aspects of a First-Order Logic with Sorts, Dissertation, Technische Fakultät der Universität des Saarlandes, Saarbrücken, Germany.

WEIDENBACH C. [1998], Sorted unification and tree automata, *in* W. Bibel and P. H. Schmitt, eds, 'Automated Deduction - A Basis for Applications', Vol. 1 of *Applied Logic*, Kluwer, chapter 9, pp. 291–320.

WEIDENBACH C. [1999], Towards an automatic analysis of security protocols in first-order logic, *in* H. Ganzinger, ed., '16th International Conference on Automated Deduction, CADE-16', Vol. 1632 of *LNAI*, Springer, pp. 378–382.

WEIDENBACH C., AFSHORDEL B., BRAHM U., COHRS C., ENGEL T., KEEN E., THEOBALT C. AND TOPIC D. [1999], System description: Spass version 1.0.0, *in* H. Ganzinger, ed., '16th International Conference on Automated Deduction, CADE-16', Vol. 1632 of *LNAI*, Springer, pp. 314–318.

WOS L., ROBINSON G. AND CARSON D. [1965], 'Efficiency and completeness of the set of support strategy in theorem proving', *Journal of the ACM* **12**(4), 536–541.

Appendix

A. SPASS Version 2.0 Options

From a (unix) shell, SPASS is called

```
SPASS [options] [<input-file>]
```

where `options` influence the behavior of the prover. Here, we only introduce SPASS options that relate to the content of this chapter. Further options, e.g., options controlling the output, are documented in the SPASS man-pages. The options discussed here apply to SPASS Versions 2.0. Options can be set to integer values. For boolean options `0` means falsity and `1` means truth. For example, the option `-IMPm=1` enables the inference rule merging paramodulation which can be abbreviated by `-IMPm` whereas `-IMPm=0` disables the inference rule.

A.1. Control

`Auto`	Automatic Mode, after a problem analysis, all options are set automatically.
`FullRed`	Full Reduction, Section 3. If full reduction is enabled, the overall SPASS loop corresponds to the loop presented in Table 1, if the option is disabled, it corresponds to the lazy reduction loop presented in Table 3.
`BoundMode`	Bound Mode selects the mode for resource controlled generation of the search space, Section 3. If set to `1` clauses are weight restricted, if set to `2` clauses are depth restricted.
`BoundStart`	Bound Start determines the start value for resource restriction, Section 3.
`BoundLoops`	Bound Loops determines the number of resource restricted main-loop iterations.
`DocProof`	activates proof documentation.

A.2. Inference Rules

`ISoR`	Sort Constraint Resolution, Definition 4.3.
`IEmS`	Empty Sort, Definition 4.4.
`IEqR`	Equality Resolution, Definition 4.7.
`IERR`	Reflexivity Resolution, Definition 4.7.
`ISpL`	Superposition Left, Definition 4.8.
`IOPm`	Ordered Paramodulation, Definition 4.8 and Definition 4.9.
`ISPm`	(Standard) Paramodulation, Definition 4.8 and Definition 4.9.

`ISpR`	Superposition Right, Definition 4.9.
`IOFc`	Ordered Factoring, Definition 4.10.
`ISFc`	(Standard) Factoring, Definition 4.10.
`IEqF`	Equality Factoring, Definition 4.11.
`IMPm`	Merging Paramodulation, Definition 4.12.
`IORe`	Ordered Resolution, Definition 4.13.
`ISRe`	(Standard) Resolution, Definition 4.13.
`IOHy`	Ordered Hyper Resolution, Definition 4.14.
`ISHy`	(Standard) Hyper Resolution, Definition 4.14.
`Splits`	Splitting, Definition 4.25. The option determines the number of splitting applications where any negative number means that splitting is not restricted.

A.3. Reduction Rules

`RSSi`	Sort Simplification, Definition 4.5.
`RSST`	Static Soft Typing, Definition 4.6.
`RObv`	Trivial Literal Elimination, Definition 4.15.
`RFSub`	Forward Subsumption Deletion, Definition 4.16, Table 5.
`RBSub`	Backward Subsumption Deletion, Definition 4.16, Table 6.
`RCon`	Condensation, Definition 4.17.
`RTaut`	Tautology Deletion, Definition 4.18. If the option is set to `1` only syntactic tautologies are eliminated. If it is set to `2`, semantic tautologies are deleted as well.
`RUnC`	Unit Conflict, Definition 4.19.
`RTer`	Terminator, Definition 4.19, where the value of the option determines the number of non-unit clause occurrences in the searched refutation.
`RFMMR`	Forward Matching Replacement Resolution, Definition 4.20, Table 5.
`RBMMR`	Backward Matching Replacement Resolution, Definition 4.20, Table 6.
`RFRew`	Forward Rewriting, Definition 4.21 and Definition 4.22, Table 5.
`RBRew`	Backward Rewriting, Definition 4.21, Table 6.
`RAED`	Assignment Equation Deletion, Definition 4.24. If set to `2`, it is assumed that any model has a non-trivial domain and the corresponding eliminations are performed.

B. Pointers into the SPASS Source Code

The below tabular relates algorithms presented in this chapter to the actual source code. It is meant to provide a starting point to explore further details or to adapt the code to personal desires. For every topic, we point to the SPASS source file and the name of the corresponding function.

Main-Loop, Table 7	$\longrightarrow$	`top.c`
	$\longrightarrow$	`top_ProofSearch`
$fred$, Table 5	$\longrightarrow$	`rules-red.c`
	$\longrightarrow$	`red_CompleteReductionOnDerivedClause`
$ired$, Table 6	$\longrightarrow$	`rules-red.c`
	$\longrightarrow$	`red_CompleteReductionOnDerivedClauses`
inf, Table 7	$\longrightarrow$	`rules-inf.c`
	$\longrightarrow$	`inf_DerivableClauses`

C. Links to Saturation Based Provers

Bliksem	by Hans de Nivelle `http://www.mpi-sb.mpg.de/~nivelle/`
Discount	by Jörg Denzinger `http://agent.informatik.uni-kl.de/denzinge/denzinger.html`
E	by Stephan Schulz [Schulz 1999] `http://wwwjessen.informatik.tu-muenchen.de/personen/schulz.html`
Fiesta	by Robert Nieuwenhuis, Pilar Nivela and Guillem Godoy `http://www.lsi.upc.es/~roberto/`
Gandalf	by Tanel Tammet `http://www.cs.chalmers.se/~tammet/gandalf/`
Otter	by William McCune [McCune and Wos 1997] `http://www-unix.mcs.anl.gov/AR/otter/`
Saturate	by Harald Ganzinger, Robert Nieuwenhuis and Pilar Nivela [Nivela and Nieuwenhuis 1993] `http://www.mpi-sb.mpg.de/SATURATE/`

SPASS	by Christoph Weidenbach, Bijan Afshordel, Enno Keen, Christian Theobalt, Dalinor Topić [Weidenbach et al. 1999] `http://spass.mpi-sb.mpg.de/`
Vampire	by Alexandre Riazanov and Andrei Voronkov [Riazanov and Voronkov 1999] `http://www.cs.man.ac.uk/fmethods/vampire/`
Waldmeister	by Arnim Buch, Thomas Hillenbrand, Roland Vogt, Bernd Löchner and Andreas Jaeger [Hillenbrand, Jaeger and Löchner 1999] `http://agent.informatik.uni-kl.de/waldmeister/`

Index