# On the Saturation of YAGO

Martin Suda and Christoph Weidenbach and Patrick Wischnewski

MPI–I–2010–RG1–001 February 2010

**Authors' Addresses**

Christoph Weidenbach
Max-Planck-Institut für Informatik
Campus E1 4
66123 Saarbrücken
Germany

Patrick Wischnewski
Max-Planck-Institut für Informatik
Campus E1 4
66123 Saarbrücken
Germany

Martin Suda
Max-Planck-Institut für Informatik
Campus E1 4
66123 Saarbrücken
Germany

**Publication Notes**

This report is an extended version of an article intended for publication elsewhere.

**Abstract**

YAGO is an automatically generated ontology out of Wikipedia and WordNet. It is eventually represented in a proprietary flat text file format and a core comprises 10 million facts and formulas. We present a translation of YAGO into the Bernays-Schönfinkel Horn class with equality. A new variant of the superposition calculus is sound, complete and terminating for this class. Together with extended term indexing data structures the new calculus is implemented in SPASS-YAGO. YAGO can be finitely saturated by SPASS-YAGO in about 1 hour. We have found 49 inconsistencies in the original generated ontology which we have fixed. SPASS-YAGO is able to prove non-trivial conjectures with respect to the resulting saturated and consistent clause set of about 1.4 GB in less than one second.

**Keywords**

superposition, large finite domain reasoning, reasoning in large ontologies, term indexing

# Contents

# 1 Introduction

YAGO (Yet Another Great Ontology) has been developed by our colleagues from the database/information retrieval group at the Max Planck Institute for Informatics [17]. It attracted a lot of attention in the information retrieval community because it was the first automatically retrieved ontology with both an accuracy of about 97% and a high coverage as it includes a unification of Wikipedia and WordNet. It contains about 20 million "facts" of the YAGO language. A detailed introduction to YAGO containing a comparison to other well-known ontologies can be found in [18].

After a close inspection of the YAGO language it turned out that the Bernays-Schoenfinkel Horn class with equality, abbreviated *BSHE* from now on, is sufficiently expressive to cover a core of YAGO. In 2008 the idea was born to write a translation procedure from YAGO into BSHE and then use Spass in order to find *all* inconsistencies in YAGO and to answer queries. The translation procedure is described in Section 3. We then started running Spass on the resulting formulas in a kind of "test and refine" loop, eventually leading to the Spass-YAGO variant of Spass, a new superposition calculus for BSHE, an extension to context tree indexing, and this paper.

The first step was actually to make Spass ready for handling really big formula and clause sets. Some of this work went already into Spass 3.5 [26], the basis for Spass-YAGO, but further refinements were needed in order to actually start the experiments on YAGO. The engineering steps taken are explained in Section 6.

After the first experiments on smaller fragments of YAGO it immediately became clear that the standard superposition calculus does not work sufficiently well on BSHE. We started searching for a calculus that is sound, complete and terminating on BSHE and at the same time generates "small" saturations. The YAGO language assumes a unique name assumption, i.e., all constants are different. This can be translated into first-order logic by enumerating disequations $a \not\approx b$ for all different constants $a, b$. For several million constants this translation is not tractable. Bonancina and Schulz [15] there-

fore suggested additional inference rules instead of adding the disequations. We followed this approach and further refined one of their rules according to the BSHE fragment and the rest of our calculus. The BSH fragment can be decided by positive hyper resolution. Hyper resolution is a good choice anyway, because it prevents the prolific generation of intermediate resolvents of the form $\neg A_1 \vee \ldots \neg A_n \vee B$ that would be generated and kept by (ordered) binary resolution if there are no resolution partners for some $\neg A_i$. Experiments showed that this works nicely for most types of clauses resulting from the translation. For example, in YAGO a relation $Q$ can be defined to be functional, translated into the clause $\neg Q(x, y) \vee \neg Q(x, z) \vee y \approx z$. If hyper resolution succeeds on generating a ground clause $(y \approx z)\sigma$ out of this clause, it is either a tautology or the unique name assumption rule mentioned above will refute the clause. The search space generated by hyper resolution out of subsort definitions and transitive relations contained in YAGO turned out to be too prolific. Therefore, we further composed our calculus by adding chaining for transitive relations [3] and sort reasoning [25]. The latter is available in SPASS anyway, whereas for chaining we added a novel implementation. All details on the BSHE fragment generated out of YAGO and the eventual calculus including proofs for completeness, soundness, and termination plus implementation aspects are discussed in Section 4.

Thirdly, it turned out that the well-known indexing solutions for first-order theorem proving [11] are too inefficient for the size and structure of the YAGO BSHE fragment. The problem is that for example unifiability queries with a query atom $Q(x, a)$ need an index to both discriminate on the signature symbols $Q$ and $a$ without explicitly looking at all potential partner atoms in the index. In Section 5 we present an extension to context tree indexing [5] called *Filtered Context Trees* that discriminate for the above example on $Q$ and $a$ in logarithmic time in the number of symbols, i.e. in logarithmic time the filtered context tree index gives access to a structure that contains all potential partners containing these symbols. Context trees are a generalization of substitution trees used in SPASS. In SPASS-YAGO the context tree extension is finally implemented as an extension to substitution tree indexing.

Eventually, SPASS-YAGO saturates the BSHE translation of YAGO in 1 hour, generating 26379349 clauses. The generated saturated clause set consists of 9943056 clauses. We found 49 inconsistencies which we resolved by hand. With respect to saturated clause set we can prove queries in less than one second (Section 7). The paper ends with a summary of the obtained results and directions for future work (Section 8). Detailed proofs and algorithms are available in a technical report [20]. SPASS-YAGO and all input files are available from the SPASS homepage `http://www.spass-prover.org/` in

section prototypes and experiments.

# 2 Preliminaries

We follow the notation from [25]. A first-order language is constructed over a signature $\Sigma$. We assume $\Sigma$ to be a finite set of function symbols. In addition to the signature $\Sigma$ we assume that there is an infinite set $\mathcal{V}$ of variables. The set of terms $\mathcal{T}(\Sigma, \mathcal{X})$ over a signature $\Sigma$ and a set of variables $\mathcal{X}$ with $\mathcal{X} \subset \mathcal{V}$ is recursively defined: $\mathcal{X} \subseteq \mathcal{T}(\Sigma, \mathcal{X})$ and for every function symbol $f \in \Sigma$ with arity zero (*a constant*) $f \in \mathcal{T}(\Sigma, \mathcal{X})$ and if $f$ has arity $n$ and $t_1, \ldots t_n \in \mathcal{T}(\Sigma, \mathcal{X})$ then also $f(t_1, \ldots t_n) \in \mathcal{T}(\Sigma, \mathcal{X})$. The variables $\mathcal{V} \setminus \mathcal{X}$ are used as meta variables in context tree indexing. Let $\mathrm{vars}(t)$ for a term $t \in \mathcal{T}(\Sigma, \mathcal{X})$ be the set of all variables occurring in $t$. If $t = f(t_1, \ldots, t_n)$ then $\mathrm{top}(t) = f$.

A substitution $\sigma : \mathcal{V} \to \mathcal{T}(\Sigma, \mathcal{X})$ is a mapping from the set of variables into the set of terms such that $x\sigma \neq x$ for only finitely many $x \in \mathcal{V}$. The domain of a substitution $\sigma$ is defined as $\mathrm{dom}(\sigma) = \{x \mid x\sigma \neq x\}$ and the codomain is defined as $\mathrm{cod}(\sigma) = \{x\sigma \mid x\sigma \neq x\}$. Substitutions are lifted to terms as usual. Given two terms $s$ and $t$, a substitution $\sigma$ is called a *unifier* if $s\sigma = t\sigma$ and *most general unifier* (mgu) if, in addition, for any other unifier $\tau$ of $s$ and $t$ there exists a substitution $\lambda$ with $\sigma\lambda = \tau$. A substitution $\sigma$ is called a *matcher* from $s$ to $t$ if $s\sigma = t$. The term $s$ is then called a *generalization* of $t$ and $t$ an *instance* of $s$. A substitution $\sigma$ is a *unifier for substitutions* $\tau$ and $\rho$ if $\sigma$ is a unifier of $x\tau$ and $x\rho$ for all $x \in \mathrm{dom}(\tau)$. The definitions for matcher, generalization and instance can be lifted to substitutions analogously. The composition $\sigma \circ \tau$ of the two substitutions $\sigma$ and $\tau$ is defined as $(x\sigma)\tau$.

# 3 Translation of YAGO into BSHE

From a logical perspective, YAGO [17, 18] consists of about 20 million ground atoms of second-order logic. However, most of the second-order content is actually "syntactic sugar" that can be eventually translated into first-order logic.For example, subsort relations are represented as facts over the involved sort predicates.

The YAGO ontology comprises facts of the form

$$arg1 \quad rel \quad arg2$$

where $rel$ is a relation and $arg1$, $arg2$ are either individuals or are relations. For example, the following fact states that Albert Einstein is born in Ulm

$$\text{AlbertEinstein} \quad \text{bornIn} \quad \text{Ulm}$$

where bornIn is a relation, AlbertEinstein and Ulm are individuals. For the translation of YAGO into BSHE, we transform each fact of this form, where the arguments of a relation are entities, into a ground atom. The relation becomes a binary predicate symbol and an individual becomes a constant. We translate the above example into

$$bornIn(AlbertEinstein, Ulm)$$

The relation type of YAGO assigns a type to an individual or to an relation. For example, the following says that Angela Merkel is a human

$$\text{AngelaMerkel} \quad \text{type} \quad \text{human}$$

The fact stating that the relation bornIn is a function, is

$$\text{type} \quad \text{bornIn} \quad \text{yagoFunction}$$

In the first case we translate the fact into a ground instance of the sort predicate *human* as follows

$$human(AngelaMerkel)$$

The second case seems to be second-order but it is actually "syntactic sugar" for the following first-order constraint

$$\neg bornIn(x, y) \vee \neg bornIn(x, z) \vee x = z$$

Likewise, the fact stating that a relation is of type yagoTransitiveRelation is translated into the repsective first-order constraint. For example, the fact

locatedIn type yagoTransitiveRelation

is translated into the constraint

$$\neg locatedIn(x, y) \vee \neg locatedIn(y, z) \vee locatedIn(x, z)$$

The last kind of facts that we consider for our translation are facts of the relation subClassOf. The following example states that each human is a mammal

human subClassOf mammal

From a logical point this also seems to be second-order because this fact states over the sort predicates *human* and *mammal*. However, we can translate this into the following subsort relation

$$\neg human(x) \vee mammal(x)$$

The above kind of facts make up about half of YAGO, i.e., about 10 million facts translated into ground atoms and clauses of the above form. The translation results in first-order ground facts and non-unit clauses one half each. For this report we left out YAGO facts about the source of information as well as confidence values attached to the facts. For example, in YAGO for each relation occurring in a YAGO fact there is also a fact relating it to the link of the website it was extracted from as well as further facts relating to links of other websites containing the same relation.

# 4 A new Calculus for BSHE

We translated YAGO into the Bernays-Schönfinkel Horn class with equality where all the clauses are range restricted. This means that any clause has the form $C \vee A$ or just $C$ with the following conditions satisfied

- Horn clauses: $C$ contains only negative literals and $A$ is a positive literal,
- range restricted: $Var(A) \subseteq Var(C_n)$, where $C_n$ is the subclause of $C$ consisting of all the non-equality atoms of $C$,
- Bernays-Schönfinkel: the signature $\Sigma$ contains only constant symbols,
- equality ($\approx$) is present among the predicate symbols.

By using the unique name assumption, which is in our case imposed on all the constant symbols from $\Sigma$, the given set of clauses can be further simplified before starting the actual reasoning process. Each clause of the form $C \vee a \not\approx b$ is a tautology and can therefore be removed. If it is of the form $C \vee a \not\approx a$ the literal $a \not\approx a$ can be deleted. Moreover, clauses of the form $C \vee x \not\approx t$, for variable $x$ and term $t$ (either a variable or a constant) can be simplified to $C[x \leftarrow t]$. Thus we may assume that the clause set does not contain disequation literals. When we look at the positive occurrences of the equality predicate, we can do yet another simplification: a clause of the form $C \vee a \approx b$ can be simplified to $C$, because $a \approx b$ is false in any interpretation satisfying the unique name assumption. As noted in the introduction, we used the refinement of the calculus presented in [15] to deal with the unique name assumption.

Another key ingredient in the process of saturation of YAGO is the chaining calculus, a refinement of superposition designed to deal efficiently with transitive relations [3]. It is well known that the axiom stating that a relation $Q$ is transitive,

$$Q(x, y) \wedge Q(y, z) \rightarrow Q(x, z),$$

may be a source of non-termination in resolution proving. This is because the transitivity axiom clause may be resolved with (a variant of) itself to yield a new clause $Q(x,y) \wedge Q(y,z) \wedge Q(z,w) \rightarrow Q(x,w)$. Evidently, such process can be arbitrarily iterated. Even if we use selection of negative literals or hyperresolution to block the self-inference, (hyper)resolution will eventually explicitly compute the whole transitive closure of the relation $Q$.

The idea of chaining is to remove the prolific transitivity axiom from the given clause set, and instead to introduce a couple of specialized inference rules that encode the logical consequences of transitivity in a controlled way. The crucial restriction lies in requiring that the two literals $Q(u,v)$ and $Q(v,w)$ chain together only if $v \succeq u$ and $v \succeq w$, where $\succ$ is a standard superposition term ordering. In order to show that such a restricted version of the rule is still complete techniques from term rewriting are employed.

An important step in introducing the chaining calculus to a theorem prover is the implementation of a new literal ordering. In the standard superposition setting literal ordering is typically defined as the two-fold multiset extension of the term ordering on the so called *occurrences* of equations/atoms (see e.g. [25] for details). This, for instance, entails that $\neg A \succ A$ for any atom $A$, a property essential for the completeness of the calculus. Unfortunately, however, stronger properties are required for the chaining to work, namely to ensure that the chaining inferences are decreasing, i.e. that the conclusion of an inference is always smaller than the maximal premise. These properties are integrated under the notion of *admissibility* of the literal ordering.

**Definition 1.** *An ordering $\succ$ on ground terms and literals is called* admissible [3] *if*

- *it is well-founded and total on ground terms and literals,*
- *it is* compatible with reduction on maximal subterms*, i.e. $L \succ L'$ whenever $L$ and $L'$ contain the same transitive predicate symbol $Q$, and the maximal subterm of $L'$ is strictly smaller than the maximal subterm of $L$,*
- *it is* compatible with goal reduction*, i.e.*
  - *$\neg A \succ A$ for all ground atoms $A$,*
  - *$\neg A \succ B$ whenever $A$ is an atom $Q(s,t)$ and $B$ is an atom $Q(s',t')$, such that $Q$ is a transitive predicate and $max(s,t) \succeq max(s',t')$,*
  - *$\neg A \succ \neg B$ whenever $A$ is an atom $Q(s,s)$ and $B$ atom $Q(s,t)$ or $Q(t,s)$, where $Q$ is a transitive predicate and $s \succ t$.*

*An ordering on ground clauses is called* admissible *if it is the multiset extension of an admissible ordering on literals.*

In order to actually implement an admissible ordering on ground literals, we can work as follows. We associate to each literal $L$ a tuple $(max_L, p_L, min_L)$ and compare these lexicographically, using the superposition term ordering $\succ$ in the first and last component, and the ordering $1 > 0$ in the middle component. The individual members of the tuple are defined as follows: If $L$ is of the form $Q(s,t)$ for a transitive predicate $Q$ we set $max_L$ to the maximum of $s$ and $t$, and $min_L$ to the minimum of the two terms (with respect to $\succ$). If $L$ is of the form $A$ or $\neg A$ for some atom $A$ the top symbol of which is not a transitive predicate, we set $max_L = A$ and $min_L = \top$, where $\top$ is special symbol minimal in the term ordering $\succ$. We set $p_L = 1$, if $L$ is negative, and 0 otherwise.

We use $\succ$ to denote both the standard term ordering, which is as usual assumed to be total on ground terms, and the just described admissible ordering on literals and clauses. Context should always make clear what instance of $\succ$ is meant.

Lifting the lexicographic ordering of the tuples to the non-ground level is a non-trivial task. For instance, the maximum of $s$ and $t$ may not be unique, because the term ordering $\succ$ cannot be total on non-ground terms. Nevertheless, it is possible to proceed by simultaneously considering both cases. Then it can happen that we produce a distinctive result, as opposed to just reporting incomparability of the two literals in question, which is always a sound solution, because it only means that more inference will potentially have to be done. For example, comparing the two non-ground literals $L_1 = \neg R(s_1, t_1)$ and $L_2 = R(s_2, t_2)$ where the term pairs $s_1, t_1$, $s_2, t_2$, and $t_1, s_2$ are incomparable respectively, but $s_1 \succ s_2$ and $t_1 = t_2$, we can report that $L_1 \succ L_2$ although we don't know whether $max_{L_1}$ is $s_1$ or $t_1$. For the instances where $max_{L_1}$ is $t_1$ the $p_L$-member of the tuple takes over. Obviously, we try to identify as many such cases as possible, to be able to restrict applicability of the inferences.

## 4.1 The proof system

Here we present the inference rules of our calculus. They are refinements of calculi presented in [3] and [15] composed and specialized for BSHE. For the chaining rules, we assume that $Q$ is a transitive predicate.

*Ordered Chaining*

$$\frac{Q(l,s) \quad Q(t,r)}{Q(l,r)\sigma}$$

where $\sigma$ is the most general unifier of $s$ and $t$, $l\sigma \not\succeq s\sigma$, and $r\sigma \not\succeq t\sigma$.

*Negative Chaining*

$$\frac{Q(l,s) \quad D \vee \neg Q(t,r)}{D\sigma \vee \neg Q(s,r)\sigma}$$

where $\sigma$ is the most general unifier of $l$ and $t$, $s\sigma \not\succeq l\sigma$, and $r\sigma \not\succeq t\sigma$, and

$$\frac{Q(l,s) \quad D \vee \neg Q(t,r)}{D\sigma \vee \neg Q(t,l)\sigma}$$

where $\sigma$ is the most general unifier of $s$ and $r$, $l\sigma \not\succeq s\sigma$, and $t\sigma \not\succ r\sigma$.

*Hyperresolution*

$$\frac{A_1 \quad \ldots \quad A_n \quad \neg B_1 \vee \ldots \vee \neg B_n \vee P}{P\sigma},$$

where $n \geq 1$, $A_1, \ldots, A_n$ are unit clauses, $P$ is a positive literal or false, and $\sigma$ is the simultaneous most general unifier of $A_i$ and $B_i$ respectively, for all $i \in \{1, \ldots n\}$.

*OECut [15]*

$$\frac{a \approx b}{\bot},$$

where $a$ and $b$ are two different constants.

In negative chaining, the case $t\sigma = r\sigma$ needs to be dealt with by only one of the two negative chaining rules. We do not impose maximality restrictions on the negative literal as this would cause incompleteness in the combination with hyperresolution. Positive hyperresolution alone is known to decide Horn function-free clauses, but with respect to YAGO the search space becomes too prolific. Therefore, we developed the above calculus where transitivity is replaced by the specific chaining rules.

## 4.2 Completeness, soundness, and termination

In this section we show that our calculus is complete, sound, and terminating for the Bernays-Schönfinkel Horn class with equality with range restricted clauses.

The completeness proof is based on the ideas from [3] adapted to our special case. It incorporates the notion of redundancy, so the standard elimination rules like subsumption and tautology deletion can be added to the calculus. The model construction itself proceeds along standard lines. One takes the set of all ground instances of the given saturated clause set, and

uses the clause ordering which is total and well-founded on the ground level to inductively build partial interpretations. In order to satisfy all the clauses in the final interpretation, some of the clauses are designated as productive, which means they contribute with a positive atom to the interpretation. A specialty of our case is that we additionally need to consider a closure of the contributed atoms in order to obtain the right interpretation. Moreover, we only allow positive unit clauses to potentially become productive (this can be justified by viewing all the negative literals as implicitly selected). We now build up the theory needed to establish the completeness theorem formally.

We assume a fixed theory TRANS of axioms stating transitivity for predicates $Q_1, \ldots, Q_l$, and a theory UNA $= \{a \not\approx b | a \neq b, a \in \Sigma, b \in \Sigma\}$ for the unique name assumption. We define the following notions:

**Definition 2.** *A* chain *is a finite sequence of atoms*

$$Q(l_0, l_1), Q(l_1, l_2), \ldots, Q(l_{n-1}, l_n)$$

*where $n \geq 1$ and all terms $l_0, \ldots, l_n$ are ground and $Q$ is a transitive predicate. The* type *of such a chain is the atom $Q(l_0, l_n)$. A chain is called a* proof *in a Herbrand interpretation $I$ if all atoms $Q(l_{i-1}, l_i)$ are true in $I$. We say $Q(l_0, l_n)$ is provable in $I$ if there exists a proof of type $Q(l_0, l_n)$ in $I$.*

Note that this notion of proof enjoys the *subproof property* (subsequence of a proof is again a proof) and the *subproof replacement property* (whenever we replace a subproof with another subproof of the same type, we again obtain a proof).

**Definition 3.** *The* transitive closure *of $I$ (with respect to* TRANS*) is defined as the set $I$ plus all ground atoms $Q(l, r)$ that are provable in $I$.*

**Observation 1** (Characterization of transitive closure)**.** *A Herbrand interpretation $I$ is a model of a set of transitivity axioms* TRANS *if and only if it is identical to its transitive closure (w.r.t* TRANS*).*

*Proof.* For one direction, use induction on the length of proofs to show that whenever $I$ is a model of TRANS, then it is identical to its transitive closure. The other direction is straightforward. □

We now aim at defining rewrite proofs. We first fix a total well-founded ordering $\succ$ on ground terms, which allows us to classify *proof steps* $Q(l, r)$ according to the order relation between the two terms. We write:

- $l \Rightarrow_Q r$ if $l \succ r$,

- $l \Leftarrow_Q r$ if $r \succ l$,
- $l \Leftrightarrow_Q r$ if $l = r$.

The annotation $Q$ will be omited if it is clear from the context or inessential. We can now distinguish special kinds of proofs:

**Definition 4.** Valley *is a chain of the form*

$$l_0 \Rightarrow l_1 \ldots \Rightarrow l_k \Leftarrow l_{k+1} \Leftarrow \ldots \Leftarrow l_n$$

*or*

$$l_0 \Rightarrow l_1 \ldots \Rightarrow l_k \Leftrightarrow l_{k+1} \Leftarrow \ldots \Leftarrow l_n$$

*Valleys are also called* rewrite proofs. *A two step chain $l \Leftarrow t \Rightarrow r$ is called a* peak. *A chain $l \Leftrightarrow l \Rightarrow r$ or $l \Leftarrow r \Leftrightarrow r$ is called a* plateau. *A chain $l = l_0 \Leftrightarrow l_1 \Leftrightarrow \ldots \Leftrightarrow l_k = r$ is called a* plain *if $k \geq 2$.*

**Observation 2** (Characterization of a valley)**.** *A valley is a chain that contains no peak, plateau or plain.*

**Definition 5.** *We write $l \Downarrow_Q^I r$ to indicate that there exists a rewrite proof of (type) $Q(l, r)$ in $I$. We say that peak, plateau or plain* commutes *in $I$ if there exists a rewrite proof of the same type in $I$. A* rewrite closure *of $I$ is defined as $I \cup \{Q(l, r) : l \Downarrow_Q^I r\}$.*

Note that rewrite closure is obviously contained in the transitive closure.

**Definition 6** (Complexity of rewrite steps)**.** *We define*

- *the complexity of $l \Rightarrow_Q r$ as the multiset $\{l\}$,*
- *the complexity of $l \Leftarrow_Q r$ as the multiset $\{r\}$,*
- *the complexity of $l \Leftrightarrow_Q r$ as the multiset $\{l, r\}$.*

*The complexity of a chain is the multiset of the complexities of all its individual steps.*

We compare two chains by comparing their respective complexities in the two-fold multiset extension of the ordering $\succ$ and denote the resulting ordering by $\succ_\pi$. Such an ordering on proofs can be called *proof ordering* as it satisfies the following properties:

- A proper subproof of a proof is smaller than the original proof.

- Replacement of any subproof by a smaller proof will result in a smaller proof.

**Definition 7.** *A proof of $Q(l, r)$ in $I$ is said to be* minimal *(w.r.t. $\succ_\pi$) if there exists no smaller proof of the same type in $I$.*

**Observation 3** (Characterization of minimal proofs)**.** *Let $\succ$ be a well-founded ordering on ground terms, $\succ_\pi$ be the corresponding proof ordering, and $I$ be a Herbrand interpretation. If no peak, plateau, or plain in $I$ is a minimal proof, then all minimal proofs in $I$ are rewrite proofs. Furthermore, if a peak, plateau or plain commutes in $I$, then it is nonminimal.*

*Proof.* Direct inspection shows that any rewrite proof is simpler (according to $\succ_\pi$) than any peak, plateau, or plain of the same type. If a proof contains a peak, plateau, or plain as a subproof, then that subproof is nonminimal and hence can be replaced by a simpler proof. The result is a simpler proof of the same type, which implies that the original proof is nonminimal. Thus, all minimal proofs must be rewrite proofs. □

**Lemma 1** (Commutation)**.** *Let $\succ$ be a well-founded ordering on ground terms. The rewrite closure of $I$ w.r.t. a set of transitivity laws* TRANS *is a model of* TRANS *if and only if all peaks in $I$ commute.*

*Proof.* It can easily be seen that if the rewrite closure of $I$ is a model of TRANS, then all peaks in $I$ commute. For the other direction, it is sufficient by characterization of transitive closure to show that the rewrite closure of $I$ is the same as the transitive closure. Suppose all peaks in $I$ commute. First, note that if a proof contains at least two steps, then any one identity step can be deleted, the result being a simpler (and shorter) proof of the same type. This implies that no plateau or plain is minimal. By assumption, peaks commute and hence are also nonminimal. We may use Characterization of minimal proofs to infer that all minimal proofs are rewrite proofs. In short, if an atom $Q(l, r)$ is provable in $I$, then it also has a rewrite proof. Consequently, the rewrite closure of $I$ is the same as the transitive closure. □

We say that a ground inference is *decreasing* with respect to a clause ordering $\succ$ if its conclusion is smaller than the maximal premise.

**Lemma 2** (Decreasing inferences)**.** *If $\succ$ is an admissible clause ordering (i.e. the multiset extension of an admissible ordering on literals), then any ground inference is decreasing w.r.t. $\succ$.*

*Proof.* Let us consider the individual rules:

- Ordered Chaining: This follows from the compatibility with reduction of maximal subterms.

- Negative Chaining 1: Consider a ground negative chaining inference

$$\frac{Q(l,s) \quad D \vee \neg Q(l,r)}{D \vee \neg Q(s,r)},$$

  where $l \succ s$, $l \succ r$. Since $\succ$ is compatible with reduction of maximal subterms, we may infer that $\neg Q(l,r) \succ \neg Q(s,r)$. The conclusion is therefore smaller than the second premise.

  Negative Chaining 2 is very similar, but also needs the property "compatibility with goal reduction" point three, for the case where the negative transitive literal is of the form $\neg Q(l,l)$.

- Hyperresolution: $P$ is necessarily smaller than the nucleus $\neg B_1, \ldots, \neg B_n, P$ as it is its sub-multiset.

- OECut: trivial.

□

We say that a ground clause $C$ is *redundant* with respect to $N$ if there exists a set $\{C_1, \ldots, C_k\}$ of ground instances of $N$ such that $C$ is true in every model of $\{C_1, \ldots, C_k\}$ and $C \succ C_j$, for all $j$ with $1 \leq j \leq k$. A nonground clause is called *redundant* if all its ground instances are.

A ground inference $\pi$ is *redundant* with respect to $N$ if either one of its premises is redundant, or else there exists a set $\{C_1, \ldots, C_k\}$ of ground instances of $N$ such that the conclusion of $\pi$ is true in every model of $\{C_1, \ldots, C_k\}$ and $C \succ C_j$, for all $j$ with $1 \leq j \leq k$, where $C$ is the maximal premise of $\pi$. A nonground inference is called *redundant* if all its ground instances are redundant.

We say that a set of clauses $N$ is *saturated up to redundancy* with respect to some inference system, if all inferences from $N$ are redundant.

Given a set of ground clauses $N$ we define a corresponding Herbrand interpretation $I$ (a "candidate model" for $N$) by induction on $\succ$.

**Definition 8** (Candidate models). • *For every clause $C$ in $N$ let $R_C$ be the set $\bigcup_{C \succ D} E_D$ and $I_C$ the rewrite closure of $R_C$.*

- *If $C$ is a unit clause $P$, where $P$ is a positive literal and $C$ is false in $I_C$ then $E_C = \{P\}$. In this case we also say that $C$ is* productive *(and* produces *$P$). In all other cases, $E_C = \emptyset$.*

- *Finally, let $R$ be the set $\bigcup_C E_C$ and $I$ the rewrite closure of $I$.*
- *We also use $R^C$ to denote the set $R_C \cup E_C$ and $I^C$ to denote the rewrite closure of $R^C$.*

**Lemma 3** (Productive clause)**.** *If $C$ is a productive clause in $N$, then it is true in $I^C$.*

**Lemma 4** (Monotonicity)**.** *Let $\succ$ be an admissible ordering. If a ground clause $C$ (which need not to be in $N$) is true in some interpretation $I_D$ or $I^D$, where $D \succeq C$, then it is also true in $I$ and in all interpretations $I_{D'}$ and $I^{D'}$, where $D' \succ D$ (and $D$ and $D'$ are clauses in $N$).*

*Proof.* Let $C$, $D$ and $D'$ be ground clauses, such that $D' \succ D \succeq C$ and $D$ and $D'$ are elements of $N$. From the above definitions, it can be seen that $I_D \subseteq I^D \subseteq I_{D'} \subseteq I^{D'} \subseteq I$. Thus if a positive literal $A$ in $C$ is true in $I_D$ or $I^D$, then $A$ (and hence $C$) is also true in $I_{D'}$, $I^{D'}$ and $I$.

If, on the other hand, a negative literal $\neg A$ in $C$ is true in $I_D$ or $I^D$, then $A$ is false in $I_D$ or $I^D$. We claim that $A$ is also false in $I$. Since the clause ordering is admissible (i.e. the multiset extension of an admissible ordering on literals), we know that if $B$ is an atom produced by a clause greater than or equal to $D$ then $B \succ \neg A$. Since $\succ$ is compatible with goal reduction, we have $\neg A \succ A$ and also $\neg A \succ A'$, for any atom $A' = Q(l, r)$ for which $Q(l, r)$ may occur in a rewrite proof of type $A$. In other words, no atom $B$ produced by any clause greater than $D$ can possibly be used in a rewrite proof of $A$. This implies that $A$ is false, and $\neg A$ true, in $I$. We conclude that $C$ is true in $I_{D'}$ and $I^{D'}$, as well as in $I$.

□

The lemma is typically used in its contrapositive form to infer that $C$ is false in the interpretations $I_C$ and $I^C$ whenever it is false in $I_{D'}$ or $I^{D'}$, for some $D' \succ C$.

**Lemma 5** (Model construction)**.** *Let $\succ$ be an admissible ordering and $N$ be a set of ground Horn clauses that is saturated up to redundancy and does not contain the empty clause. If $I$ is the interpretation constructed from $N$ then for every clause $C$ in $N$ we have:*

1. *If $C$ is productive, then it is non-redundant.*
2. *Both $I^C$ and $I_C$ are transitivity interpretations satisfying the unique name assumption.*
3. *The clause $C$ is true in $I^C$.*

*Proof.* The proof is by induction on $\succ$. Let $C$ be a ground clause in $N$, such that assertions (1)-(3) are satisfied for all clauses in $N$ that are smaller than $C$.

1. We prove the contraposivite statement. Suppose $C$ is redundant in $N$; that is, there exist smaller ground instances $C_1, \ldots, C_n$ of $N$ such that $C$ is true in every model of $\{C_1, \ldots, C_n\}$. We may use parts (2) and (3) of the induction hypotheses and Monotonicity lemma to infer that $I_C$ is a model of $\{C_1, \ldots, C_n\}$. Therefore $C$ is true in $I_C$, which implies it is non-productive.

2. The equation $a \approx b$ can never be produced for two different constants $a$ and $b$, because otherwise there would be a one step OECut inference turning the clause $a \approx b$ into the empty clause, which we assume is not in $N$ (and which can never be redundant).[1]

   By Commutation Lemma, it suffices now to prove that all peaks in $R_C$ and $R^C$, respectively, commute. Each peak in $R_C$ is a peak in $R^{C'}$, for some $C'$ with $C \succ C'$, and commutes in $R^{C'}$ by the induction hypotheses. Thus it also commutes in $R_C$.

   If $E_C$ is nonempty, then there may be peaks in $R^C$, which are not provable in $R_C$. In that case, $C$ is productive. Let $l \Leftarrow_Q t \Rightarrow_Q r$ be a peak in $R^C$. Then there exists a unit clause $Q(l,t)$ that produces $Q(l,t)$, and another clause $Q(t,r)$ that produces $Q(t,r)$. The two clauses are not identical. Both clauses are nonredundant by part (1) of the induction hypotheses, and the larger of the two is $C$. From these two clauses we obtain $C' = Q(l,r)$ by ordered chaining. Since $N$ is saturated up to redundancy, there exists clauses $C_1, \ldots, C_n$ smaller than $C$, such that $C'$ is true in every model of $\{C_1, \ldots, C_n\}$. We may use the Monotonicity lemma and parts (2) and (3) of the induction hypothesis to infer that $I_C$ is one such model. Thus, the clause $C'$ is true in $I_C$ and therefore, the atom $Q(l,r)$ must be true in $I_C$, that is, $l \Downarrow_Q^{R_C} r$, which indicates that the peak commutes in $R_C$.

3. We already know that all ground instances of $N$ that are smaller than $C$ are true in $I^C$ and that $I^C$ is a transitivity interpretation satisfying the unique name assumption. Therefore, if $C$ is redundant, it is true in $I^C$. If $C$ is productive, it is true in $I^C$ by definition. Suppose $C$ is neither redundant nor productive. This means that $C$ is of the form

[1] As a side remark we note that we also never produce the equations of the form $a \approx a$, simply because they are redundant. This has the nice consequence that any Herbrand interpretation automatically has to be an interpretation of the equality predicate.

$\neg B_1 \vee \ldots \vee \neg B_n \vee P$, where $n \geq 1$ and $P$ is a positive unit clause or the empty clause. Assume $B_i$ are true in $I_C$, $P$ is false in $I_C$, otherwise we are done.

If all the $B_i$ are directly produced by unit clauses, i.e. $B_i \in R_C$, then we get the clause $C' = P$ by hyperresolution. Since $N$ is saturated up to redundancy, (and $B_i$'s are not redundant being productive – part (1) of induction hypotheses), there exist clauses $C_1, \ldots, C_k$ smaller than $C$, such that $C'$ is true in every model of $\{C_1, \ldots, C_k\}$. We may use the induction hypotheses to infer that $I_C$ is one such model. Necessarily $C'$ is not the empty clause, $P$ is true in $I_C$, and hence $C$ is true in $I^C$, which is a contradiction.

If some of $B_i \notin R_C$, it has to be an atom $Q(l, r)$ with a rewrite proof of at least two steps in $R_C$. Then there exists a productive clause $Q(l, l')$ (where $l \succ l'$ and $l' \Downarrow_Q^{R_C} r$) or $Q(r', r)$ (where $r \succ r'$ and $l \Downarrow_Q^{R_C} r'$). By negative chaining we get $\neg B_1, \ldots, \neg B_{i-1}, \neg Q(l', r), \neg B_{i+1}, \ldots, \neg B_n, P$ or $\neg B_1, \ldots, \neg B_{i-1}, \neg Q(l, r'), \neg B_{i+1}, \ldots, \neg B_n, P$. In either case we may again use saturation up to redundancy to infer that inference conclusion is true in $I^C$, but that means that either $\neg Q(l', r)$ or respectively $\neg Q(l, r')$ is true in $I^C$, and hence $C$ is true in $I^C$, again a contradiction.

□

**Theorem 1** (Completeness)**.** *If a set of Horn clauses $N$ is saturated up to redundancy then the set $N \cup \mathrm{TRANS} \cup \mathrm{UNA}$ is unsatisfiable if and only if $N$ contains the empty clause.*

*Proof.* If $N$ does not contain the empty clause, we claim that the Herbrand interpretation $I$ constructed from the set of all ground instances of $N$ is a model of $N \cup \mathrm{TRANS} \cup \mathrm{UNA}$. Via the usual lifting argument[2] the set of all ground instances is saturated as well. By the model construction lemma, every ground instance $C$ of a clause in $N$ is true in $I$, and in addition $I$ is a transitivity interpretation and satisfies the unique name assumption. □

**Theorem 2** (Soundness)**.** *The presented calculus is sound. Conclusion of any inference is logically entailed by the premises of the inference and the theory* (TRANS ∪ UNA).

*Proof.* The claim is obvious for hyperresolution, and also for the OECut rule, where we use the unique name assumption. Finally, all the chaining rules

[2]Note that we only consider ground version of the OECut rule. Nevertheless, it does not need to be lifted in our case. It is because our clauses are range restricted, and therefore we can never generate a non-ground positive (unit) clause.

can be simulated as two resolution steps between the participating premises and the appropriate transitivity axiom clause. □

**Theorem 3** (Termination)**.** *The calculus terminates on the set of Horn clauses from Bernays-Schönfinkel class.*

*Proof.* No inference rule produces a longer clause than any of its premises. There are only finitely many clauses of given length (up to variable renaming) as all the function symbols are constants. □

# 5 Term Indexing

The invention of term indexing data structures has been pivotal for the success of automated theorem proving. Likewise, it is vital to develop efficient indexing mechanisms for the reasoning on huge sets of clauses such as the clause set resulting from the translation of YAGO into the BSHE class. The atoms occurring in these clauses are of the form: $Q(a, b)$, $Q(a, x)$, $Q(x, b)$, $Q(x, y)$, $S(a)$ and $S(x)$, where $Q$ is a binary predicate symbol, $a$, $b$ are constants and $S$ is a monadic predicate (sort symbol) from the signature. In order to perform retrieval operations on an index containing such atoms, we have to discriminate efficiently on all occurring term positions. Therefore, we develop a filtering mechanism for context tree indexing [5] which efficiently filters out subtrees of the indexing that do not lead to a success with respect to the current retrieval operation. The resulting new indexing is called *Filtered Context Tree* indexing. The filtered context tree indexing enables SPASS to efficiently reason about the clauses resulting from the translation of the core of YAGO. Without the filtering SPASS was even unable to load these clauses into the index.

In the first section, Section 5.1, we give a definition and the required notions for context trees. After that we give a complete overview of the algorithms for all the operations of the context tree indexing. These are the algorithms for the retrieval operations (instance, unifier, generalization) as well as the insertion and deletion operation of terms. Based on this notions and algorithms we introduce filtered context tree indexing as an extension to context tree indexing in Section 5.2. Also, we present details about the integration of the filtering into SPASS and show further optimizations.

## 5.1 Context Tree Indexing

Context tree indexing [5] is a generalization of substitution tree indexing [7]. In order to be self-contained the following section shows the definitions of

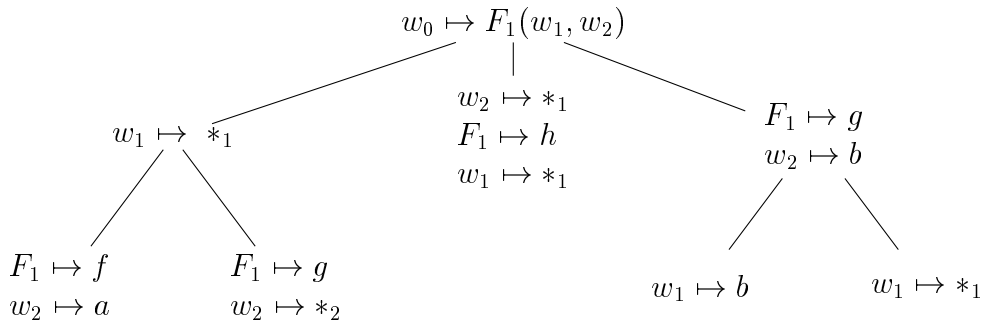


Figure 5.1: Context tree

context tree indexing following notions from [7] as well as the algorithms performing all the operations for term indexing structures. This section also completes the introductory article of context tree indexing [5] which only presents the algorithms for the retrieval of generalizations.

### 5.1.1 Context Trees

Let $t_1, \ldots, t_n$ be terms and $P$ be a predicate symbol with arity $n$ then $P(t_1, \ldots, t_n)$ is an atom. An atom or its negation is called a literal. Compared to substitution trees, context trees can additionally share common subterms even if they occur below different function symbols via the introduction of extra variables for function symbols. These variables are called function variables. For example, the terms $f(s,t)$ and $g(s,t)$ can be represented as $F_1(s,t)$ with children $F_1 = f$ and $F_1 = g$. The function variable $F_1$ represents a single function symbol. In the context of deep terms, this potentially increases the degree of sharing in an index structure.

Before inserting a term into the index, variables of the term are normalized. The normalization is a renaming of the variables of the term which increases the sharing. Assume a infinite set of variables $\mathcal{V}^*$ which are totally ordered with respect to a ordering $<^*$. Let $*_1$ be the smallest element in $\mathcal{V}^*$. A substitution $\sigma$ is a normalization for a term $t$ if $\sigma$ is a renaming for the variables of $t$ and $\mathrm{cod}(\sigma) = \{*_1, \ldots, *_n\}$ and for each $*_i, *_{i+1} \in \mathrm{cod}(\sigma)$ there is no $*_{i'} \in \mathcal{V}^*$ with $*_i <^* *_{i'} <^* *_{i+1}$.

Figure 5.1 depicts a context tree containing the terms $f(*_1, a)$, $g(*_1, *_2)$, $h(*_1, *_1)$, $g(b,b)$, and $g(*_1, b)$.

**Definition 9** (Function variables)**.** *We assume a set of function variables $\mathcal{U} \subset \mathcal{V}$ which is disjoint from the set of variables $\mathcal{X}$. The set of terms $\mathcal{T}(\Sigma \cup \mathcal{U}, \mathcal{X})$ are terms build over the signature $\Sigma$, the function variables $\mathcal{U}$ and the variables $\mathcal{X}$. The notion of a substitution can be adapted accordingly.*

**Definition 10** (Index variables). *Assume a set of index variables $\mathcal{W} \subset \mathcal{V}$ which is pairwise disjoint from $\mathcal{X}$ and $\mathcal{U}$. Index variables are denoted by $w_i$. We also assume a set of index function variables which are denoted by $F_i$.*

**Definition 11** (Context Tree). *A context tree is a tree $T = (V, E, \text{subst}, v_r)$ where $V$ is a set of vertexes, $E \subset V \times V$ is the edge relation, the function* subst *assigns to each vertex a substitution, $v_r \in V$ is the root node of $T$ and the following properties hold:*

1. *each node is either a leaf or an inner node with at least two children.*
2. *for every path $v_1 \dots v_n$ from the root ($v_1 = v_r$) to any node it holds:*

$$\text{dom}(\text{subst}(v_i)) \cap \bigcup_{1 \leq j < i} \text{dom}(\text{subst}(v_j)) = \emptyset$$

3. *for every path $v_1 \dots v_n$ from the root ($v_1 = v_r$) to a leaf $v_n$*

$$\text{vars}(\text{cod}(\text{subst}(v_1) \circ \dots \circ \text{subst}(v_n))) \subset \mathcal{X}$$

Each node in a context tree which is not a leaf node, must have at least two subtrees due to the first condition. The second condition ensures that each variable is bound at most once along a path. The third condition assures that all terms represented by a path from the root to a leaf are from $\mathcal{T}(\Sigma, \mathcal{X})$.

A term that is stored in a context tree is represented by a path from the root to a leaf. The respective term can be obtained by the composition of the substitutions along this path. Therefore, we extend the definition of the function vars returning the variables of a term, to the function returning the variables occurring unbound on a path of a context tree.

**Definition 12** (Variables of a path). *Let $v_1, \dots, v_n$ be a path from the root of a context tree to a node $v_n$ then the set of variables of this path is*

$$\text{vars}(v_1, \dots, v_n) = \bigcup_{i \in \{1 \dots n\}} \text{vars}(\text{cod}(\text{subst}(v_i))) \setminus \bigcup_{i \in \{1 \dots n\}} \text{dom}(\text{subst}(v_i))$$

Note, for a path $v_r = v_1, \dots, v_n$ of a context tree from the root $v_r$ to a leaf $v_n$ we have that $\text{vars}(x_1, \dots, x_n) \subset \mathcal{X}$ because of Condition 3 of Definition 11.

### 5.1.2 Algorithms for Context Trees

This section shows the algorithms for context trees implementing the standard operations for term indexing structures. The standard operations of

term indexing data structure can be separated into two categories. The first are the retrieval algorithms. These operations query a context tree index for unifiable terms, instantiations and generalizations of a given query term. In the second category are the algorithms for updating a context tree indexing structure. These are the algorithms for insertion of terms into the index and deletion of terms from the index.

### Retrieval algorithm

The query algorithms for unifiable terms, instantiations and generalization are based on a common lookup procedure which traverses the tree and applies to the substitution of each visited node the procedure Test. The procedure Test is either the test for unifiability, the test for instantiation or the test for generalization.

The query given to the lookup function is a query substitution containing the query term rather than the query term itself. This means, if $t$ is the query term, then the respective query substitution is $\{w_0 \mapsto t\}$ where $\{w_0\} = \mathrm{dom}(\mathrm{subst}(v_r))$.

**Algorithm 1:** Lookup

**Input**: context tree $T = (V, E, \mathrm{subst}, v_r)$, $v_n \in V$, substitution $\rho$, function Test

```
1 HITS = ∅;
2 foreach (v, v') ∈ E do
3 |   if Test(subst(v'),ρ) = (true, σ) then
4 |   |   if isLeaf(v') then HITS = HITS ∪ {v'};
5 |   |   HITS = HITS ∪ Lookup(T, v', ρ ∘ σ, Test);
6 |   end
7 end
8 return HITS;
```

**Lookup** The lookup procedure Lookup (Algorithm 1) expects a context tree $T$, a node $v_n$, a query substitution $\rho$ and the test function Test. The node $v_n$ is initially set to the root node of $T$ and it is the current processed node of $T$ during the recursive application of Lookup. The substitution $\rho$ is an accumulator argument. It is the composition (line 5) of the initial query substitution and all substitutions $\sigma$ computed in line 3 during the recursive application of Lookup. The function Test is one of the functions UnifyTest (Algorithm 6), GenTest (Algorithm 8) or InstTest (Algorithm 10) which tests

two substitutions for unifiability, generalization or instantiation, respectively. Each path in a context tree from the root node to a leaf node corresponds to a term stored in the index. The respective path is represented by its leaf node and each leaf node maintains a reference to the term it represents. Therefore, Lookup returns a set of leaf nodes rather than a set of terms.

The following theorem shows the correctness of the procedure Lookup for the retrieval of terms that are unifiable with the given query. The correctness of the remaining operations, generalization and instantiation, follows analogously. The correctness proof of the retrieval operation for substitution trees was originally given in [7] where we also refer to for the correctness proof of the test for unification, generalization and instantiation. These original proofs have to be adjusted slightly in order to be valid also for context trees.

**Theorem 4** (Correctness of Lookup)**.** *Let $t$ be a term, Test the test function for unification, $\rho = \{w_0 \mapsto t\}$ be the query substitution and $v_n \in$ Lookup$(T, v_r, \rho, \text{UnifyTest})$. Then $v_n$ is a leaf node and there is a path $v_r, v_1, \ldots, v_n$ and a substitution $\sigma$ with $\text{dom}(\sigma) \subset \mathcal{X}$ and*

$$w_0 \,\text{subst}(v_1) \ldots \text{subst}(v_n)\sigma = w_0\rho\sigma.$$

*Proof.* Let $v_n \in \text{Lookup}(T, v_r, \rho, \text{UnifyTest})$ and $\rho$ the query substitution. The function UnifyTest, applied in line 3, tests for two given substitutions $\tau$ and $\rho$ if there is a substitution $\sigma$ with $\forall x \in \text{dom}(\tau).x\tau\rho\sigma = x\rho\sigma$. Consequently, because of the recursive structure of Lookup there is a path $v_r, v_1, \ldots, v_n$ such that for $i \in \{1, \ldots, n\}$ and $\rho_i = \rho_{i-1} \circ \sigma_{i-1}$ with $\rho_0 = \rho$ and $\sigma_0 = \emptyset$ the following holds:

$$\exists\sigma_i.\forall x \in \text{dom}(\text{subst}(v_i)).\; x\,\text{subst}(v_i)\rho_i\sigma_i = x\rho_i\sigma_i \tag{5.1}$$

Additionally, the node $v_n$ is a leaf node because of line 4. For the correctness proof we show the following property by induction

$$\exists\sigma_m.\forall x \in \mathcal{V}.\,\text{subst}(v_1), \ldots, \text{subst}(v_m)\rho_m\sigma_m = x\rho_m\sigma_m \tag{5.2}$$

For $m = 1$ this follows immediately from (5.1). Now assume (5.2) holds for $m$. From (5.1) and Definition 11 - 2 it follows

$$\exists\sigma_{m+1}.\forall x \in \mathcal{V}.\,\text{subst}(v_1) \ldots \text{subst}(v_m)\,\text{subst}(v_{m+1})\rho_m\sigma_m\sigma_{m+1} = x\rho_m\sigma_m\sigma_{m+1} \tag{5.3}$$

The property follows for $m+1$ with $\rho_{m+1} = \rho_m \circ \sigma_m$. As a consequence

$$\exists \sigma_n . \forall x \in \mathcal{V}. \operatorname{subst}(v_1), \ldots, \operatorname{subst}(v_n) \rho_n \sigma_n = x \rho_n \sigma_n \tag{5.4}$$

We have $\rho_n = \rho \circ \sigma_1 \cdots \circ \sigma_{n-1}$ and from Definition 11 - 3 and $v_n$ is a leaf it follows that $\forall x \in \mathcal{V}.\operatorname{vars}(x \operatorname{subst}(v_1) \ldots \operatorname{subst}(v_n)) \subset \mathcal{X}$. As a result $\exists \sigma$ with $w_0 \operatorname{subst}(v_1) \ldots \operatorname{subst}(v_n)\sigma = w_0 \rho_1 \sigma$ and $\operatorname{dom}(\sigma) \subset \mathcal{X}$.

□

**Unification** The unification test of two substitutions $\tau$ and $\rho$ tests if there is a substitution $\sigma$ such that for all $x \in \operatorname{dom}(\tau)$ it holds $x\tau\rho\sigma = x\rho\sigma$. Note, that $\rho$ occurs on both sides of the equation. The substitution $\rho$ works as an accumulator argument of Lookup (Algorithm 1) and it may bind variables of $x\tau$. These bindings also have to be respected in the test function. The respective test procedure UnifyTest is depicted in Algorithm 6. The procedure UnifyTest uses the procedure TermUnify (Algorithm 5) which checks for two given terms $s$ and $t$ whether they are unifiable, i.e. if there exists a substitution $\sigma$ with $s\sigma = t\sigma$. The correctness proof of UnifyTest for substitutions trees is given in [7]. This proof can easily be extended to context trees.

**Generalization** The test function for generalization GenTest (Algorithm 8) checks for two given substitutions $\tau$ and $\rho$ if there exists a substitution $\sigma$ such that for all $x \in \operatorname{dom}(\tau) : x\tau\rho\sigma = x\rho$. Note that $\rho$ occurs on both sides because $\rho$ is the accumulator argument of Lookup (Algorithm 1) and may bind variables of $x\tau$. The implementation of this procedure is based on TermGen (Algorithm 7) that tests for two given terms $s$ and $t$ if $s$ is a generalization of $t$, i.e. if a substitution $\sigma$ exist with $s\sigma = t$. The correctness proof of GenTest for substitutions trees is given in [7]. This proof can easily be extended to context trees.

**Instance** The test function for instantiation InstTest (Algorithm 10) checks for two given substitutions $\tau$ and $\rho$ if there exists a substitution $\sigma$ such that for all $x \in \operatorname{dom}(\tau) : x\tau\rho\sigma = x\rho\sigma$ and $\operatorname{dom}(\sigma) \subset \operatorname{vars}(x\rho) \cup \mathcal{W}$. Note, that $\sigma$ occurs here on both sides of the equation. During the recursive browsing of the context tree it may become necessary for the retrieval that the substitution $\sigma$ binds index variables in $x\tau\rho$ as well as in $x\rho$. This is because of the fact, that a term in the context tree is represented by the composition of the substitutions along a path from the root to a leaf. Condition 3 in Definition 11 ensures that the algorithm has found an instance of the query once it has reached a leaf node. In the case of substitution trees we refer to [7] for the correctness proof. This proof can easily be extended to context trees.

The implementation of the procedure InstTest is based on the procedure TermInst (Algorithm 9) that tests for two given terms $s$ and $t$ if $s$ is an instance of $t$, i.e. if a substitution $\sigma$ exist with $s\sigma = t\sigma$ and $\mathrm{dom}(\sigma) \in \mathrm{vars}(t) \cup \mathcal{W}$.

### Update Algorithms

The procedures for inserting a term into a context tree and deleting a term from a context tree require a check for variations. The terms $s$ and $t$ are variants if and only if they are equal up to variable renaming. Note that all terms in a context tree are normalized. If $t'$ is the normalization of the term $t$ then the retrieval operation for variations of the term $t$ is the retrieval for unifiable terms of the query substitution $\rho = \{w_0 \mapsto t'\}$ such that for each unifier $\sigma$ we have $\mathrm{dom}(\sigma) \cap \mathcal{X} = \emptyset$.

With the variant test we can implement a procedure LookupVariant that searches a given context tree for variations analogously to Lookup (Algorithm 1). Initially $\rho = \{w_0 \mapsto t\}$ where $t$ is the normalized query term. The procedure LookupVariant returns a leaf node if $t$ is contained in the context tree. Otherwise, it returns the node $v_{best}$ which is the first node along a path from the root node to the node $v_{best}$ that is not a variant of the current substitution $\rho$.

For the insertion of the term $t$ into the index, the subnodes of $v_{best}$ are replaced by two new nodes. One node represents the former subtrees of $v_{best}$ and the other is a new leaf node which represents $t$. The substitutions of the modified node $v_{best}$ and the two new subnodes are computed such that the modified context tree fulfills Definition 11.

Considering the deletion of a term $t$ from a context tree, a term $t$ is contained in the context tree if and only if the procedure LookupVariant returns a leaf node. Then this leaf node is removed from the index. Analogous to the insertion of a term into a context tree, nodes are removed from the index during the deletion of a term. The deletion operation also ensures that the index fulfills Definition 11 after the deletion of a term.

The following section presents the procedure LookupVariant and the algorithms which implement the insertion and deletion operation using LookupVariant.

**Variation** The test procedure VariantTest (Algorithm 12) checks for a substitution $\tau$ and a substitution $\rho$ if for all $x \in \mathrm{dom}(\tau)$ $x\tau\rho\sigma = x\rho\sigma$ and $\mathrm{dom}(\sigma) \subset \mathcal{W}$. The implementation uses the procedure TermVariant (Algorithm 11) which tests for two given terms $s$ and $t$ if they are variations, i.e. $s\sigma = t\sigma$ and $\mathrm{dom}(\sigma) \subset \mathcal{W}$. Because of the fact that a term in a context tree

**Algorithm 2:** LookupVariant

**Input**: Context tree $T = (V, E, \text{subst}, v_r)$, $v_n \in V$, substitution $\rho$

```
1  HIT = ∅;
2  BEST = NULL;
3  foreach v' with (v, v') ∈ E do
4  |  if VariantTest(subst(v'), ρ) = (true, σ) then
5  |  |  if isLeaf(v') ∧ v_best = NULL then return (v', NULL, ρ ∘ σ);
6  |  |  (HIT, v_best, ρ') = LookupVariant(T, v', ρ ∘ σ, VariantTest);
7  |  |  if HIT then
8  |  |  |  return (HIT, NULL, ρ')
9  |  |
10 |  else if ∀x ∈ dom(subst(v')) top(x subst(v')) = top(xρ) and
   |  v_best = NULL then
11 |  |  v_best = v';
12 |  end
13 end
14 return (v, v_best , ρ);
```

is represented by a path from the root to a leaf, index variables are the only variables that are allowed to be bound during the retrieval for variations.

The procedure LookupVariant (Algorithm 2) is invoked with a context tree $T$, a node $v_n$, and the query substitution $\rho$. Like in the case of Lookup (Algorithm 1), the node $v_n$ is initially set to the root node of $T$ and it is the current examined node of $T$ during the recursive application of LookupVariant. The substitution $\rho$ is an accumulator argument, initially set to the substitution containing the term $t$ to be inserted. It is the composition (line 6) of the initial query substitution and all substitutions $\sigma$ computed in line 4 during the recursive application of LookupVariant. The procedure LookupVariant traverses the context tree $T$ as long as the variant test (line 4) is successful. The algorithm of VariantTest is given in Algorithm 12. If the algorithm has found a leaf node (line 5) the recursion stops and it returns this leaf node. If VariantTest fails then LookupVariant checks if the terms in the codomain of the substitution of the current node and the substitution $\rho$ have the same top symbols (line 10). If they have the same top symbols then LookupVariant remembers this node in $v_{best}$. If no variant is found then the algorithm returns $v_{best}$. This node indicates a suitable position in the context tree $T$ where a new leaf node can be created which represents $t$.

**Algorithm 3:** EntryCreate

**Input**: Context tree $T = (V, E, \text{subst}, v_r)$, term $t$

$\rho = \{w_0 \mapsto t\}$;
**if** $\neg$ IsLeaf$(v_r)$ **then**
  $(v, v_{best}, \rho') = \text{LookupVariant}(T, v_r, \rho)$;
**end**
**if** IsLeaf$(v) \wedge v_{best} = NULL$ **then** InsertReference$(v, t)$ ;
**else if** $v_{best} \neq NULL$ **then**
  $(\sigma_1, \sigma_2, \mu) = \text{mscg}(\text{subst}(v_{best}), \rho')$;
  $V = V \cup \{v_1, v_2\}$;
  **foreach** $(v_{best}, v') \in E$ **do** $E = (E \setminus \{(v_{best}, v')\}) \cup \{(v_1, v')\}$;
  $E = E \cup \{(v_{best}, v_1), (v_{best}, v_2)\}$;
  InsertReference(v$_2$, t);
  $\text{subst}(v_{best}) = \mu$;
  $\text{subst}(v_1) = \sigma_1$;
  $\text{subst}(v_2) = \sigma_2$;
**else**
  $V = V \cup \{v'\}$;
  $E = E \cup \{(v, v')\}$;
  InsertReference$(v', t)$;
**end**

**Most specific common generalization** When inserting a term $t$ into an index which contains no variant of this term, the procedure LookupVariant returns the node $v_{best}$ which is the first node along a path from the root node to the node $v_{best}$ that is not a variant of the current substitution $\rho$. For the insertion of the term $t$ into the index, the subnodes of $v_{best}$ are replaced by two new nodes. One node represents the former subtrees of $v_{best}$ and the other is a new leaf node which represents $t$. The computation of the *most specific common generalization* yields the substitutions of the modified $v_{best}$ and the two new subnodes such that they fulfill Definition 11. If $\tau$ and $\rho$ are two substitutions and there exist substitutions $\sigma_1$ and $\sigma_2$ and $\mu$ such that $\mu \circ \sigma_1 = \tau$ and $\mu \circ \sigma_2 = \rho$, then $\mu$ is called *common generalization*. Additionally, if there is a substitution $\delta$ for each other common generalization $\nu \neq \mu$ such that $\mu = \nu \circ \delta$, then $\mu$ is called *most specific common generalization* which is given by the function

$$\text{mscg}(\tau, \rho) := (\sigma_1, \sigma_2, \mu)$$

**Insert** The procedure EntryCreate inserts a term $t$ into a context tree $T$. Remember, we assume $t$ to be normalized. First the term $t$ is transformed into a query substitution $\rho = \{w_0 \mapsto t\}$. Then EntryCreate calls LookupVariant with $T$ the root node $v_r$ and the query substitution $\rho$. Three cases can occur. The first is that LookupVariant has found a leaf (line 5) which represents $t$. Then a reference to $t$ is inserted into the leaf node which is done by InsertReference. If there is no respective leaf node representing $t$ then LookupVariant returns a node $v_{best}$, if there is such a node. The node $v_{best}$ indicates a suitable insert position. In order to insert $t$ into the index, EntryCreate first computes the mscg(subst($v_{best}$), $\rho$) = $(\mu, \sigma_1, \sigma_2)$. After that, the procedure creates two new nodes $v_1$, $v_2$. All subnodes of $v_{best}$ become subnodes of $v_1$ and are deleted from the subnodes of $v_{best}$. Then $v_1$ and $v_2$ become the new subnodes of $v_{best}$ ($(v_{best}, v_1) \in E$ and $(v_{best}, v_2) \in E$). The substitutions of $v_{best}$, $v_1$ and $v_2$ are set to the substitutions computed by mscg(subst($v_{best}$), $\rho$) as follows: subst($v_1$) = $\mu$, subst($v_2$) = $\sigma_2$ and subst($v_{best}$) = $\sigma_1$. After that, the path $v_r, \ldots, v_{best}, v_1$ represents the same terms as the former path $v_r, \ldots, v_{best}$. The path $v_r, \ldots, v_{best}, v_2$ represents the inserted term. Additionally, a reference to $t$ is inserted into the leaf node $v_2$. The third case arises if none of the above occurs. This means, neither $t$ has been inserted into the index before nor is there a suitable insert position $v_{best}$. Then a new leaf node is inserted below $v$ representing $t$.

**Algorithm 4:** EntryDelete

```
   Input: Context tree T = (V, E, subst, v_r), substitution ρ
1  if IsLeaf(v_r) then
2  |  RemoveReference(T, v, ρ)
3  else
4  |  (v', v_best) = LookupVariant(T, v', ρ);
5  |  if v' ≠ ∅ then RemoveReference(T, v', ρ);
6  end
```

**Delete** The procedure EntryDelete (Algorithm 4) removes the term $t$ from the context tree $T$. Assume $t$ is normalized than the query substitution is $\rho = \{w_0 \mapsto t\}$. If $v_r$ is not a leaf node, EntryCreate applies LookupVariant in order to obtain the leaf node representing $t$. If there is such a leaf node $v'$ in $T$ then EntryDelete performs RemoveReference which removes the reference to $t$ from $v'$.

We have modified the deletion operation of the original context trees in such a way that EntryDelete does not remove nodes from the context tree when deleting a term. Instead it removes the reference of the term from

the respective leaf node. It turned out that deleting nodes from the index and ensuring that Definition 11 holds, is too expensive in our context. This requires that we also modify the invariant of context trees such that a term $t$ is contained in a context tree if and only if the leaf node representing $t$ contains also a reference to $t$. For the original algorithm we refer to [7].

**Algorithm 5:** TermUnify

**Input**: term $s$, term $t$, substitution $\sigma$

```
1  if s = x then
2  |  if sσ = t then
3  |  |  return (true, σ)
4  |  else if s ∉ dom(σ) then
5  |  |  σ = σ ∘ {s ↦ t};
6  |  |  return (true, σ);
7  |  else
8  |  |  return (false, ∅);
9  |  end
10 else if t = x then
11 |  if s = tσ then
12 |  |  return (true, σ)
13 |  else if t ∉ dom(σ) then
14 |  |  σ = σ ∘ {t ↦ s};
15 |  |  return (true, σ);
16 |  else
17 |  |  return (false, ∅);
18 |  end
19 else if s = F(s_1, ..., s_n) and t = f(t_1, ..., t_n) then
20 |  foreach i ∈ {1, ...n} do
21 |  |  (r, σ) = TermUnify(s_i, t_i, σ);
22 |  |  if r = false then return (false, ∅);
23 |  end
24 |  if F ∈ dom(σ) ∧ Fσ ≠ f then return (false, ∅);
25 |  if Fσ = f then return (true, σ) else return (true, σ ∘ {F ↦ f});
26 end
27 return (false, ∅);
```

**Algorithm 6:** UnifyTest

**Input**: substitution $\tau$, substitution $\rho$

```
1  foreach x ∈ dom(τ) do
2  |  (r, σ) = TermUnify(xτ,xρ,ρ);
3  |  if r = false then  return (false,σ)
4  end
5  return (true, σ);
```

**Algorithm 7:** TermGen

**Input**: term $s$, term $t$, substitution $\sigma$

**if** $s = x$ **then return** *(true,* $\{x \mapsto t\}$*)*;
**if** $s = F(s_1, \ldots, s_n)$ *and* $t = f(t_1, \ldots, t_n)$ **then**
  **foreach** $i \in \{1, \ldots n\}$ **do**
    $(r, \sigma) = \text{TermGen}(s_i, t_i, \sigma)$;
    **if** $r = false$ **then return** $(false, \emptyset)$;
  **end**
  **if** $F \in \text{dom}(\sigma) \ \wedge \ F\sigma \neq f$ **then return** $(false, \emptyset)$;
  **if** $F\sigma = f$ **then return** $(true, \sigma)$ **else return** $(true, \sigma \circ \{F \mapsto f\})$;
**end**
**return** $(false, \emptyset)$;

**Algorithm 8:** GenTest

**Input**: substitution $\tau$, substitution $\rho$

**foreach** $x \in \text{dom}(\tau) \cup \text{dom}(\rho)$ **do**
  $(r, \sigma) = \text{TermGen}(x\tau, x\rho, \rho)$;
  **if** $r = false$ **then return** *(false,*$\sigma$*)*
**end**
**return** $(true, \sigma)$;

**Algorithm 9:** TermInst

**Input**: term $s$, term $t$, substitution $\sigma$

```
1  if s ∈ 𝒲 then return (true, {s ↦ t});
2  if t = x then return (true, {x ↦ t});
3  if s = F(s_1, ..., s_n) and t = f(t_1, ..., t_n) then
4  |  foreach i ∈ {1, ... n} do
5  |  |  (r, σ) = TermInst(s_i, t_i, σ);
6  |  |  if r = false then return (false, ∅);
7  |  end
8  |  if F ∈ dom(σ) ∧ Fσ ≠ f then return (false, ∅);
9  |  if Fσ = f then return (true, σ) else return (true, σ ∘ {F ↦ f});
10 end
11 return (false, ∅);
```

**Algorithm 10:** InstTest

**Input**: substitution $\tau$, substitution $\rho$

```
1  foreach x ∈ dom(τ) do
2  |  (r, σ) = TermInst(xτρ,xρ,ρ);
3  |  if r = false then  return (false,σ)
4  end
5  return (true, σ);
```

**Algorithm 11:** TermVariant

**Input**: term $s$, term $t$, substitution $\sigma$

**if** $s = x \wedge s = t$ **then return** $(true, \sigma)$;
**if** $s \in \mathcal{W}$ **then**
  **if** $s\rho = t$ **then**
    **return** $(true, \sigma)$
  **else if** $s \notin \mathrm{dom}(\sigma)$ **then**
    $\sigma = \sigma \cup \{s \mapsto t\}$;
    **return** $(true, \sigma)$;
  **else**
    **return** $(false, \emptyset)$;
  **end**
**end**
**if** $s = F(s_1, \ldots, s_n)$ *and* $t = f(t_1, \ldots, t_n)$ **then**
  **foreach** $i \in \{1, \ldots n\}$ **do**
    $(r, \sigma) = \mathrm{TermVariant}(s_i, t_i, \sigma)$;
    **if** $r = false$ **then return** $(false, \emptyset)$;
  **end**
  **if** $F \in \mathrm{dom}(\sigma) \ \wedge \ F\sigma \neq f$ **then return** $(false, \emptyset)$;
  **if** $F\sigma = f$ **then return** $(true, \sigma)$ **else return** $(true, \sigma \circ \{F \mapsto f\})$;
**end**
**return** $(false, \emptyset)$;

**Algorithm 12:** VariantTest

**Input**: substitution $\tau$, substitution $\rho$

**foreach** $x \in \mathrm{dom}(\tau)$ **do**
  $(r, \sigma) = \mathrm{TermVariant}(x\tau, x\rho, \rho)$;
  **if** $r = false$ **then return** $(false, \sigma)$
**end**
**return** $(true, \sigma)$;

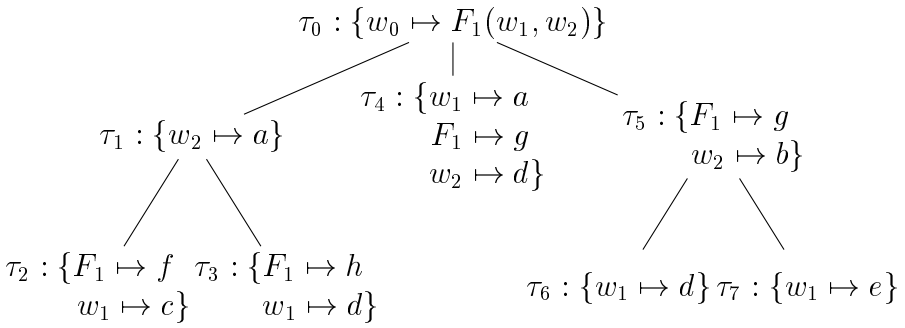


Figure 5.2: Context Tree

## 5.2 Filtered Context Tree Indexing

When performing a retrieval operation, the procedure Lookup (Algorithm 1) pursues paths that do not contribute to the current query. In the case of SPASS-YAGO this approach is not feasible because one subnode may have millions of subnodes and the term indexing is processed several thousand times in a reasoning loop. Therefore, we develop in the following a mechanism that efficiently filters out subtrees of a context tree indexing whose paths do not contribute to the current query. Without this new filtering technique, loading the clause set resulting from the translation of the core of YAGO into the index of SPASS was already not possible in reasonable time.

The following example demonstrates a retrieval operation on a context tree. The context tree of the example is a typical excerpt from the index containing the terms resulting from translating YAGO into the BSHE class.

**Example 1.** *Consider the context tree of Figure 5.2 and the retrieval of terms unifiable with the term $g(e, x)$. The query substitution $\rho$ for $g(e, x)$ is $\rho = \{w_0 \mapsto g(e, x)\}$. The algorithm starts with the query substitution $\rho$ at the node whose substitution is $\tau_0$. The substitution $\tau_0$ is unifiable with $\rho$ using the substitution $\sigma = \{w_1 \mapsto e, w_2 \mapsto x, F_1 \mapsto g\}$. Descending the indexing further requires to check all subnodes. In this case, these are the nodes containing $\tau_1$, $\tau_4$ and $\tau_5$. Unifiable under the current substitution $\rho \circ \sigma$ are the substitutions $\tau_1$ and $\tau_5$. At first, the algorithm proceeds by inspecting the subtree starting at the node with $\tau_1$. The substitution $\tau_1$ is unifiable with $\rho \circ \sigma$ using $\sigma' = \{x \mapsto a\}$. Continuing with the subnodes, the algorithms recognizes that neither $\tau_2$ nor $\tau_3$ are unifiable with $\rho \circ \sigma \circ \sigma'$. Then the algorithm backtracks, proceeds with $\tau_5$ and eventually finds a leaf where all substitutions along the path $\tau_0$, $\tau_5$, $\tau_7$ are unifiable under the respective substitution $\rho$ and returns the desired term which is $w_0\tau_0\tau_5\tau_7$.*

In this example, after processing the node containing the substitution $\tau_0$,

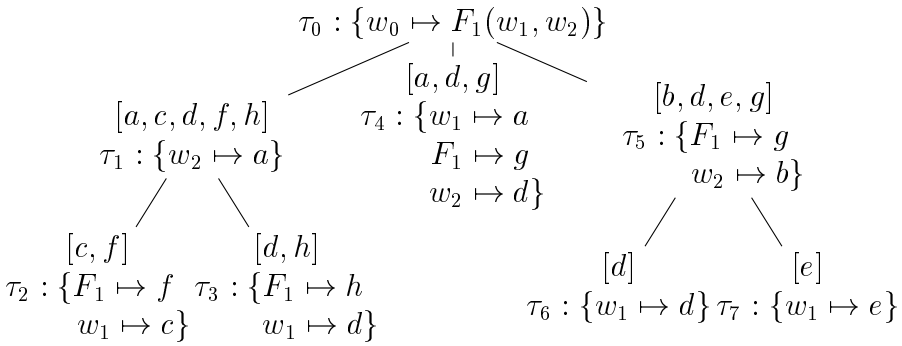


Figure 5.3: Filtered Context Tree

the retrieval procedure proceeds by examining all subnodes. These subnodes are the nodes containing the substitutions $\tau_1$, $\tau_4$ and $\tau_5$. Looking at the query the symbol $g$ has to occur in a substitution of some node along a successful path. However, if we inspect the subtree starting at the node with the substitution $\tau_1$ we recognize that the symbol $g$ does not occur in any substitution of this subtree. Consequently, this subtree does not have a successful path and can be excluded from further processing. It can be filtered by only looking at the occurring function symbols.

In the following, we introduce this new filtering technique in detail and show the respective retrieval operations. In Section 5.2.1, we introduce filtered context trees and in Section 5.2.2 we present the algorithms for the retrieval operations of filtered context trees. Additionally, we proof the correctness and completeness of these algorithms. Details about the implementation of filtered context trees in SPASS can be found in Section 5.2.3 and further potential improvements in Section 5.2.4.

### 5.2.1 Filtered Context Trees

In this section, we first define the characteristic function for a substitution $\sigma$ as the set of top symbols occurring in some term of $\text{cod}(\sigma)$. We call the result of applying the characteristic function to a substitution $\sigma$ the characteristic of $\sigma$. Once we have defined the characteristic function for a substitution we can define *Filtered context trees* as an extension of context trees. Filtered context trees contain additionally a mapping function $M$. The function $M$ of a filtered context tree $FT$ maps to each node $v$ and each symbol $s \in \Sigma$ the set of subnodes of $v$ such that $v_1 \in M(v, s)$ if and only if $v_1$ is a subnode of $v$ and there is a path $v_1, \ldots, v_n$ in $FT$ such that there is a node $v_i$ on this path with $s$ is in the characteristic of $\text{subst}(v_i)$.

Additionally, we change the lookup procedure Lookup (Algorithm 1) such

that it applies the function $M$ on the current node $v_n$ and on each symbol in the characteristic of $\rho$. The results from $M$ are the subnodes of $v_n$ that have a subtree which is compatible to $\rho$ with respect to the characteristic function. This means that the symbols in the characteristic of $\rho$ also occur in the characteristic of a substitution of a node in the subtree starting at $v_n$. The subtrees compatible with $\rho$, are potentially successful with respect of the current retrieval operation. As a consequence, all other nodes can be excluded from further processing.

**Example 2.** *Reconsider Example 1 with the unification retrieval operation for the query substitution* $\rho = \{w_0 \mapsto g(e, x)\}$. *Figure 5.3 depicts the filtered context tree obtained by extending Figure 5.2 such that at each node v those symbols are indicated that occur as top symbols in a term of the codomain of a substitution along a path starting at v. This represents the function M of the filtered context tree. The retrieval algorithm applied to Figure 5.3 examines the node containing the substitution* $\tau_0$. *The substitution* $\tau_0$ *is unifiable with* $\rho$ *using the unifier* $\sigma = \{w_1 \mapsto e, w_2 \mapsto x, F_1 \mapsto g\}$. *As we have seen, only those subtrees can contribute to the current retrieval operation that contain g in a term of the codomain of the substitution of any of its nodes. In our example these are the subtrees starting at the nodes containing the substitution* $\tau_4$ *and* $\tau_5$. *Consequently, the node containing the substitution* $\tau_1$ *does not need to be considered during the retrieval.*

A mapping mechanism has also been used for discrimination tree indexing. In discrimination tree indexing the mapping assigns to a given label the respective successor node of the discrimination tree. For example, this has been added to the indexing of the theorem prover E [14].

As mentioned before, we define the characteristic function for a substitution as the set of top symbols occurring in its codomain. If there are only variables in the codomain of a substitution $\sigma$, we define the characteristic of $\sigma$ as the set $\{\bot\}$ where $\bot$ is a symbol with $\bot \notin \Sigma$.

Because of condition 2 of Definition 11 each index variable occurs at most once on a path of a context tree. For this reason, we restrict the substitution $\rho$ when computing the characteristic function as depicted in the following example.

**Example 3.** *Consider Example 2 with the query substitution* $\rho = \{w_0 \mapsto g(e, x)\}$. *Assume the retrieval procedure descends to the node with* $\tau_5$. *The new* $\rho$ *becomes* $\rho = \rho \circ \sigma = \{w_0 \mapsto g(e, x), w_1 \mapsto e, w_2 \mapsto x, F_1 \mapsto g\}$. *The variables* $w_2$ *and* $F_1$ *have already been bound in* $\tau_5$. *Consequently, the only variables that can be bound in the substitution of a node occurring below* $\tau_5$ *in the context tree is* $w_1$. *Therefore, we only need to compute the characteristic*

*function of* $\{w_1 \mapsto x\}$. *The result of the characteristic function is* $\{\bot\}$ *because* $x$ *is a variable.*

As a result we define the characteristic function with respect to a set of variables $\mathcal{O}$. In the improved lookup procedure FilteredLookup (Algorithm 13) the set $\mathcal{O}$ is instantiated with $\text{vars}(v_r, \ldots, v_n)$ for a path $v_r, \ldots, v_n$ of a filtered context tree $FT$. These are exactly these index variables that are bound below $v_n$ in $FT$.

**Definition 13** (Characteristic function)**.** *Let* $\sigma$ *be a substitution and* $\mathcal{O}$ *be a set of variables. The set of top symbols of* $\sigma$ *and* $\mathcal{O}$ *is defined as*

$$ts(\sigma, \mathcal{O}) = \{f \mid \exists x \in \text{dom}(\sigma) \cap \mathcal{O} \text{ with } x\sigma = f(\ldots)\}$$

*The* characteristic function $\text{chr}(\sigma, \mathcal{O})$ *of a substitution* $\sigma$ *with respect to the set of variables* $\mathcal{O}$ *is defined as follows:*

$$\text{chr}(\sigma, \mathcal{O}) = \begin{cases} ts(\sigma, \mathcal{O}) & \text{if } ts(\sigma, \mathcal{O}) \neq \emptyset \\ \{\bot\} & \text{if } ts(\sigma, \mathcal{O}) = \emptyset \wedge \exists x \in \text{dom}(\sigma) \text{ with} \\ & \quad x\sigma \in \mathcal{X} \vee x\sigma \in \mathcal{T}(\mathcal{U}, \mathcal{X}) \vee x \in \mathcal{X} \\ \emptyset & \text{otherwise} \end{cases}$$

Note that this definition also includes the cases where $x\sigma$ is a constant or $x\sigma$ is a function symbol mapped from a function variable.

**Example 4.** *Reconsider the query substitution* $\rho = \{w_0 \mapsto g(e, x)\}$ *of Example 1. The characteristic function of* $\rho$ *is* $\text{chr}(\rho, \{w_0\}) = \{g\}$. *Note that* $g$ *is the only symbol of the characteristic function of* $\rho$ *because this is the top symbol of the term* $g(e, x)$. *A term that is unifiable with* $g(e, x)$ *is of the form* $g(y, x)$, *where* $y$ *is either a variable or the constant* $e$. *Consequently, the symbol* $g$ *is the only symbol characterizing* $\rho$.

Once we have defined the characteristic function for a substitution, we can extend the definition of context trees with a function $M$ that assigns to a given node $v$ and a symbol $s$ a set of successor nodes. For each node $v'$ in the set of successor nodes it holds that there is a node on a path, starting at $v'$, which contains the symbol $s$ in the characteristic of its substitution. This lifts the characteristic function of a substitution of one node to the characteristic of a subtree of a context tree.

**Definition 14** (Filtered Context Tree)**.** *A* filtered context tree $FT = (V, E, \text{subst}, v_r, M)$ *is a context tree* $(V, E, \text{subst}, v_r)$ *together with*

**Algorithm 13:** FilteredLookup

**Input**: $FT = (V, E, \text{subst}, v_r, M)$, $v_n \in V$, substitution $\rho$, function Test

1 $HITS = \emptyset$;
/* $v_r = v_1, ..., v_n$ path from the root $v_r$ to $v_n$ */
2 $C = \text{chr}(\rho, \text{vars}(v_r, ..., v_n))$;
3 **if** $C = \{\bot\}$ **then** $N = \{v' | (v_n, v') \in E\}$;
4 **else if** $C \neq \emptyset$ **then** $N = \bigcup_{s \in C \cup \{\bot\})} M(v_n, s)$;
5 **else** $N = \emptyset$;
6 **foreach** $v' \in N$ **do**
7 | **if** $\text{Test}(\text{subst}(v'), \rho) = (true, \sigma)$ **then**
8 | | **if** isLeaf$(v')$ **then return** $\{v'\}$;
9 | | $HITS = HITS \cup \text{FilteredLookup}(FT, v', \rho \circ \sigma, \text{Test})$;
10 | **end**
11 **end**
12 **return** *HITS*

*a function* $M : V \times (\Sigma \cup \{\bot\}) \to 2^V$ *from nodes and function symbols to a subset of* $V$ *such that* $v_{k+1} \in M(v_k, s)$ *if and only if there is a path* $v_1, \ldots, v_k, v_{k+1}, \ldots, v_n$ *where* $v_1$ *is the root node* $v_r$ *with*

$$s \in \bigcup_{i \in \{k+1, \ldots, n\}} \text{chr}(\text{subst}(v_i), \text{vars}(v_1, \ldots, v_k))$$

### 5.2.2 Algorithms for Filtered Context Trees

The procedure FilteredLookup (Algorithm 13) depicts the function performing the lookup operation on a given filtered context tree $FT$, a starting node $v_n$, a query substitution $\rho$ and a function Test. The node $v_n$ is the current processed node of $FT$ during the recursive application of FilteredLookup. Initially, the node $v_n$ is the root node $v_r$. The function Test is either UnifyTest(Algorithm 6), GenTest (Algorithm 8) or InstTest (Algorithm 10). These are the standard algorithms for the test functions shown in Section 5.1 which are independent from the underlying indexing. This is because they expect only two substitutions as their argument. As a result, the standard algorithms can also be used for filtered context trees.

In line 2 FilteredLookup (Algorithm 13) computes the characteristic function of $\rho$ with respect to the set of variables that have not yet occurred in the domain of a substitution of a node on the path $v_r, \ldots, v_n$. If the characteristic function returns $\{\bot\}$ then the loop in line 6 inspects all subnodes

of the given node $v_n$. Otherwise, the algorithm looks for the symbols in $M$ and considers only those nodes which are returned by $M$ (line 4). Beginning with line 6, FilteredLookup is exactly the same algorithm as Lookup (Algorithm 1). Computing the characteristic of the substitution $\rho$ in line 2 is in time $O(|\operatorname{dom}(\rho)|)$, where $|\operatorname{dom}(\rho)|$ is the number of elements of the domain of $\rho$. As a result, obtaining the set $N$ from $M$ in line 4 is in time $O(|\operatorname{dom}(\rho)| * \log|\Sigma|)$ where $|\Sigma|$ is the number of symbols in the signature. Hence, the overhead for the filtering is in $O(|\operatorname{dom}(\rho)| * \log|\Sigma|)$.

The algorithms for insertion EntryCreate (Algorithm 3) and deletion EntryDelete (Algorithm 4) use the procedure LookupVariant (Algorithm 2). The procedure LookupVariant has to be modified analogously to FilteredLookup (Algorithm 13) due to the fact that LookupVariant is a variation of Lookup (Algorithm 1).

Additionally, the procedure EntryCreate (Algorithm 3) has to maintain the map $M$ when inserting a term into the indexing. If the procedure inserts a new inner node in line 6 - 14 then the function $M$ has to be updated in order to meet the properties required in Definition 14. All nodes $v_i$ along the path $v_r, \ldots, v_1$ have to be updated as follows

$$\forall s \in \operatorname{chr}(\sigma_1, \operatorname{vars}(v_r, \ldots, v_i)).M(v_i, s) = M(v_i, s) \cup \{v_{i+1}\}$$

The nodes along the path $v_r, \ldots, v_2$ have to be updated analogously.

The function $M$ is realized via a mapping and can, therefore, be accessed in $O(\log|\Sigma|)$ where $|\Sigma|$ is the number of signature symbols. As a result, updating the nodes along a path with length $n$ is in

$$O(n * (|\operatorname{chr}(\sigma_1, \mathcal{W})| + |\operatorname{chr}(\sigma_2, \mathcal{W})|) * \log|\Sigma|)$$

In the context of YAGO the characteristic functions of $\sigma_1$ and $\sigma_2$ have size at most two and the index has depth at most three. So, maintaining $M$ is very cheap.

Remember, that we have modified the original procedure for deleting terms from an context tree. Nodes are not deleted from a context tree during the deletion of a term $t$ from the context tree because this is not feasible in the context of YAGO. Instead the term $t$ is deleted from the context tree by removing the reference to $t$ from the leaf node representing $t$. Consequently, we have changed the invariant such that a term $t$ is contained in an context tree if and only if there is a leaf node in the context tree that represents $t$ and has a reference to $t$. As a consequence, the function $M$ of a filtered context tree is not updated when deleting a term and the complexity for EntryDelete of filtered context trees is the same as for LookupVariant.

**Theorem 5** (Correctness)**.** *FilteredLookup is correct.*

*Proof.* Since, the algorithm only restricts the number of nodes in the context tree which are considered for testing, the correctness follows from the correctness of Lookup (Theorem 4). □

In the following, we proof the completeness of FilteredLookup (Algorithm 13) for the retrieval of substitutions that are unifiable with the query substitution. The proof for the retrieval of substitutions with respect to instantiation and generalization is analogous. Since LookupVariant is a slight modification of Lookup, the completeness proof for LookupVariant for filtered context trees is also analogous.

**Lemma 6.** *Let* $FT = (V, E, \text{subst}, v_r, M)$ *be a filtered context tree,* $\rho$ *be a substitution,* $v' \in V$ *a node,* $(v', v) \in E$, $\tau = \text{subst}(v)$ *and* $\mathcal{O} = \text{vars}(v_r, \ldots, v')$. *If* $\exists\sigma\forall x.x\rho\sigma = x\tau\rho\sigma$ *then* $\exists s \in \text{chr}(\rho, \mathcal{O})$ *with* $v \in M(v', s)$ *or* $\text{chr}(\tau, \mathcal{O}) = \{\bot\}$ *or* $\text{chr}(\rho, \mathcal{O}) = \{\bot\}$.

*Proof.* Assume $\exists\sigma\forall x.x\rho\sigma = x\tau\rho\sigma$. If $\exists x \in \text{dom}(\rho) \cap \mathcal{O} \cap \mathcal{W}$ with $x\rho = f(\ldots)$ (this also includes $F\rho = f$ with $F \in \mathcal{U}$) then the following two cases have to be considered:

- If $\forall w_i' \in \text{dom}(\tau)$ it holds that $w_i'\tau \in \mathcal{X}$ then $\text{chr}(\tau, \mathcal{O}) = \bot$ by Definition 13
- Else, $\exists w_i' \in \text{dom}(\tau)$ such that $w_i'\tau \notin \mathcal{X}$, then by Definition 14 and Definition 12 $w_i \in \text{dom}(\tau)$ or there is a node $v''$ that is in a subtree of $v$ and $w_i \in \text{dom}(\text{subst}(v''))$. If $w_i \in \text{dom}(\tau)$ then there exist a substitution $\sigma$ with $w_i\rho\sigma = w_i\tau\rho\sigma$ by assumption. Consequently, $f \in \text{chr}(\rho, \mathcal{O})$, $f \in \text{chr}(\tau, \mathcal{O})$ and by Definition 14 $v \in M(v', f)$.

  If $w_i \in \text{dom}(\text{subst}(v''))$ then $v \in M(v', f)$ follows inductively.

If there is no $w_i \in \text{dom}(\rho) \cap \mathcal{O} \cap \mathcal{W}$ with $w_i\rho = f(\ldots)$ then $\forall x \in \text{dom}(\rho) \cap \mathcal{O} \cap \mathcal{W}$ one of the following holds:

- $x \in \mathcal{X}$
- $x\rho \in \mathcal{X}$
- $x\rho \in \mathcal{T}(\mathcal{U}, \mathcal{X})$

For all of these cases $\text{chr}(\rho, \mathcal{O}) = \{\bot\}$. □

**Theorem 6** (Completeness)**.** *Let* $\rho$ *be a substitution and* $T = (V, E, \text{subst}, v_r)$ *a context tree. If Lookup (Algorithm 1) applied to* $T$ *and* $\rho$ *returns a non-empty set of leaf nodes* $L$ *then FilteredLookup (Algorithm 13) returns the same set* $L$ *when applied to* $\rho$ *and the filtered context tree* $FT = (V, E, \text{subst}, v_r, M)$.

*Proof.* Assume $v, v' \in V$, $E(v', v)$, $\tau = \text{subst}(v)$, $\mathcal{O} = \text{vars}(v_r, \ldots, v')$ and $\exists\sigma.\forall x.x\rho\sigma = x\tau\rho\sigma$. We have to show that $v$ is in $N$ in line 6 of Algorithm 13. From Lemma 6 we have to consider three cases:

- If $\text{chr}(\rho, \mathcal{O}) = \{\bot\}$ then $v \in N$ because of line 3.
- If $\text{chr}(\tau, \mathcal{O}) = \{\bot\}$ then $v \in N$ because of line 4.
- If $\exists s \in \text{chr}(\rho, \mathcal{O})$ with $v \in M(v', s)$ then $v \in N$ because of line 4.

Then, the theorem follows by induction on the path length. □

### 5.2.3 Implementation in Spass-YAGO

Since context trees are a generalization of substitution trees and SPASS has an implementation of substitution tree indexing, our implementation of SPASS-YAGO contains the substitution tree indexing of SPASS together with the above described filtered techniques.

In SPASS, symbols are internally represented as integers. Consequently, they can be compared with respect to their integer value. So, we implemented the lookup function $M$ using $\mathsf{CSB}^+$-trees [12] , a cache conscious variant of $B$-trees.

The implementation of the set of variables of a path $\text{vars}(v_r = v_1, \ldots, v_n)$ is realized via a marking mechanism. Each time a substitution $\tau$ of a node is compatible with the current query $\rho$ all index variables of $\text{dom}(\tau)$ are marked.

Since, one node of a filtered context tree could be reached via several symbols from its parent node, we mark each visited node in order to avoid multiple inspections of the same node.

For each of the retrieval operations (unification, instantiation and generalization) we have implemented a separate version of the procedure Filtered-Lookup (Algorithm 13) because this allows a more efficient implementation for each individual retrieval operation. More subnodes of a given filtered context tree may be filtered. For example, assume the retrieval for instances of the substitution $\{w_i \mapsto g(x)\}$. In this case, nodes that solely contain substitutions of the form $\{w_i \mapsto x\}$ do not contribute and can be excluded from further processing. A similar argument holds for generalizations.

### 5.2.4 Further Improvements

There are further opportunities to improve our current implementation of SPASS-YAGO. For example, the occur check for the unification operation can be omitted.

In the context of YAGO, the notion of function variables provides a mechanism to query for term symbols. For example, we can query the index for terms that contain the symbol $a$ as its second argument. The respective query term is $F(x, a)$. Applying this query to the context tree given in Figure 5.2 returns the terms $f(c, a)$ and $h(d, a)$. So, an implementation of filtered context trees in SPASS-YAGO provides these features.

We can also use context trees to index each term stored in the context tree by each of its symbols. For example, consider the term $f(c, a)$ which is stored in the context tree of Figure 5.2. Following the path from the root to the leaf we find the substitutions $\tau_1$ and $\tau_2$ with $f(c, a) = w_0\tau_0\tau_1\tau_2$. The order of the application of the substitutions $\tau_1$ and $\tau_2$ to $w_0\tau_0$ does not matter. As a result, $w_0\tau_0\tau_1\tau_2 = f(c, a) = w_0\tau_0\tau_2\tau_1$. If we store both paths in the context tree we can choose the path that is more efficient for the current retrieval operation. For example, consider the query term $F(x, a)$. Here the only symbol occurring is $a$. To restrict the search space we first discriminate on $a$ with the help of $\tau_1$. If we consider the query term $f(x, c)$ it is more efficient to first consider $\tau_2$ because this discriminates on $f$. Although, this approach increases the size of the filtered context tree exponentially, it is feasible in the case of YAGO. This is because a filtered context tree storing terms obtained from the translation of YAGO has depth at most three. This approach provides a very efficient retrieval mechanism. A similar idea is used for the implementation of relational data base systems, where an index is created for each of its arguments. For example, the tuple $(a, b, c)$ can be obtained by querying the indexing of the first argument for $a$, querying the indexing of the second argument for $b$ or querying the indexing for the third argument $c$. An implementation of this can be found, for example, in [10].

## 5.3 Summary

Filtered context trees are a powerful term indexing structure for storing large sets of terms and for efficiently performing unification, instantiation and generalization queries. In particular for the set of terms resulting from the translation of YAGO into BSHE, filtered context tree indexing enables the retrieval operations to avoid inspecting unsuccessful subtrees of the indexing. Consequently, our algorithms of the retrieval operations perform a more goal oriented search on the term index. In the beginning, SPASS was not able to load YAGO into its index within 24 hours. Now, with the integration of the new filtered context tree indexing, SPASS is able to load YAGO into its index and also to efficiently perform reasoning tasks on the clause set resulting from the translation of YAGO. SPASS saturates YAGO in less than one hour.

# 6 Engineering

In order to adjust SPASS to the new indexing technique and the calculus for BSHE, a lot of extra engineering had to be performed. We increased the maximal number of signature symbols that SPASS can handle to 19M. The parsing module was modified, so that originally quadratic manipulations on the lists of input clauses now only take linear time. Algorithms for manipulating clause sets holding SPASS's search state, such as loading the usable clauses, or sorting clause lists, were sped up from $O(n^2)$ to $O(n * \mathsf{log}(n))$. Hashmaps used in the clausification process in FLOTTER had to be extended to reduce the number of hash-conflicts. The structure for storing superterms in the sharing was changed from lists to maps. Newly derived clauses are now inserted at the first possible position with respect to weight in the list of usable clauses, instead of also considering search space depth. Finally, SPASS-YAGO skips auto-configuration and instead uses a standard complete flag setting in the input files according to our calculus (Section 4).

There is still plenty of room for speed ups via further engineering. Our motivation was not on getting a much faster system but to advance SPASS such that it can cope with the size of YAGO.

# 7 Experiments

We ran our experiments on a 4 x Intel Xeon Processor X5560 (8M Cache, 2.80 GHz) Debian Linux machine with 48 GB RAM. We compared SPASS-YAGO with iProver version 0.7 [9], E version 1.1 [14], and SPASS version 3.5 [26] including the before mentioned engineering improvements. The reason for this comparison is only to show that our new calculus, filtered context tree indexing and improved implementation advances the state of the art in automated reasoning on large ontologies. None of the above systems has been specifically designed to fit the BSHE theory created out of YAGO. All the provers were recompiled for the above 64 bit architecture to better cope with the large inputs.

First we evaluated the task of showing satisfiability of (slices of) YAGO after having removed all inconsistencies by hand on the basis of SPASS-YAGO runs. The examples are in favor of iProver, E, and SPASS 3.5 as we did not include the unique name assumption units for those provers, whereas SPASS-YAGO tests the corresponding inference rule. The results are given in Figure 7.1.

The second column shows the number of formulas (clauses), the third the time needed for saturation, and the fourth the number of additionally eventually kept clauses by SPASS-YAGO. All other provers fail on showing any of the examples due to timing constraints of 60 min for the first 4 slices and due to running out of (internal) memory (except for SPASS and E running out of time) for $S_4$ and the full set.

Note that showing satisfiability is the more difficult problem compared to actually proving queries. All provers can successfully solve queries with respect to at least one of the $S_0$-$S_4$ slices.

Since none of the other provers could handle the overall core, we only carried out the second experiment on queries using SPASS-YAGO. We evaluated the following two queries on the saturated core of YAGO, where we applied the now complete SOS strategy.

| Slices | Input size [F] | Time to saturate | Output size [F] | Other provers |
|---|---|---|---|---|
| $S_0$ | 136808 | 12.5 | +1768 | fail |
| $S_1$ | 132080 | 9.7 | +16060 | fail |
| $S_2$ | 96454 | 9.9 | +1768 | fail |
| $S_3$ | 114527 | 10.6 | +4769 | fail |
| $S_4$ | 4891235 | 37:11.1 | +24123 | fail |
| Full | 9918933 | 1:03:24.0 | +24123 | fail |

Figure 7.1: Saturating YAGO

$Q_1$ $\quad \exists x.politician(x) \ \wedge \ physicist(x) \ \wedge \ bornIn(x, Hamburg) \wedge hasSuccessor(Helmut\_Schmidt, x)$

$Q_2$ $\quad \exists x, y, z.diedIn(x, y) \ \wedge \ hasChild(x, z) \ \wedge \ bornIn(z, y) \wedge locatedIn(y, New\_York)$

The results of the querying are shown in the table below.

| Query | Derived | Kept | Proof length | Reasoning | Total |
|---|---|---|---|---|---|
| $Q_1$ | 1 | 1 | 18 | 0:00.1 | 9:37.8 |
| $Q_2$ | 9 | 0 | 6 | 0:00.1 | 9:38.3 |

The table shows the number of derived, kept clauses and the length of the proof found by SPASS-YAGO. Actually, almost all of the time is spent on loading the overall clause set, the difference between total time and reasoning time. The time for answering the queries is below one second. The difference between derived/kept clauses and proof length is the result of simplification, in particular sort simplification exploring subsort relationships. Recall that in the saturated core not all ground consequences of YAGO are explicitly represented. So the involved reasoning goes beyond simple data base style joins but involves reasoning about transitivity and subsort relationships.

# 8 Conclusion

The saturation of large ontologies is a challenge for first-order reasoning. The core of the YAGO ontology can be saturated by SPASS-YAGO in about 1 hour (Section 7) due to a new complete, sound, and terminating variant of the superposition calculus (Section 4) accompanied by filtered context tree indexing (Section 5) and improved implementations (Section 6). SPASS-YAGO significantly advances the state of the art in theorem proving on large ontologies (Section 7). It complements other efforts in this direction. The yearly CASC division on ontology reasoning [21] as well as approaches on combining theorem provers with other sources of knowledge [19] concentrate on finding proofs (answers, contradictions), not saturations, i.e. models of an overall ontology as we have studied in this paper for a core of YAGO. One of the first contributions on applying theorem proving to large ontologies is [8] where a number of engineering questions are discussed.

Most importantly, we showed that standard automated reasoning tools such as SPASS are able to cope with large ontologies such as a core of YAGO if the calculus and implementation are adopted accordingly. Currently, our implementation does not directly give answers but shows proofs. This can be straightforwardly extended to an answer mechanism. The queries we considered are solely existentially quantified. This can be extended to arbitrary quantifier prefixes, because we are considering a finite domain only. However, it needs further research in order to cope with the potential search space spanned by such a query. Here an even more refined calculus, e.g. by integrating chaining directly into the hyper resolution inference is instrumental. Finally, reasoning with respect to the confidence values attached to facts in YAGO that are ignored for this paper could be added to the calculus, e.g. in the style of a multi-valued logic aggregating formulas at their respective confidence values.

# Bibliography


[1] L. Bachmair and H. Ganzinger. Rewrite-based equational theorem proving with selection and simplification. *J. Log. Comput.*, 4(3):217–247, 1994.

[2] L. Bachmair and H. Ganzinger. Rewrite techniques for transitive relations. In *Proc. 9th IEEE Symposium on Logic in Computer Science*, pages 384–393. IEEE Computer Society Press, 1994. Short version of TR MPI-I-93-249.

[3] L. Bachmair and H. Ganzinger. Ordered chaining calculi for first-order theories of transitive relations. *Journal of the ACM (JACM)*, 45(6):1007–1049, 1998.

[4] C. G. Fermüller, A. Leitsch, U. Hustadt, and T. Tammet. Resolution decision procedures. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume 2, pages 1791–1849. Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, 2001.

[5] H. Ganzinger, R. Nieuwenhuis, and P. Nivela. Context trees. In Goré et al. [6], pages 242–256.

[6] R. Goré, A. Leitsch, and T. Nipkow, editors. *Automated Reasoning, First International Joint Conference, IJCAR 2001, Siena, Italy, June 18-23, 2001, Proceedings*, volume 2083 of *LNCS*. Springer, 2001.

[7] P. Graf. *Term Indexing*, volume 1053 of *LNCS*. Springer, 1996.

[8] I. Horrocks and A. Voronkov. Reasoning support for expressive ontology languages using a theorem prover. In J. Dix and S. J. Hegner, editors, *FoIKS: Foundations of Information and Knowledge Systems, Budapest, Hungary*, volume 3861 of *LNCS*, pages 201–218. Springer, 2006.

[9] K. Korovin. iProver - An Instantiation-Based Theorem Prover for First-Order Logic (System Description). In A. Armando, P. Baumgartner, and G. Dowek, editors, *IJCAR: The International Joint Conference on Automated Reasoning*, volume 5195 of *LNCS*, pages 292–298. Springer, 2008.

[10] T. Neumann and G. Weikum. Rdf-3x: a risc-style engine for rdf. *PVLDB*, 1(1):647–659, 2008.

[11] I. V. Ramakrishnan, R. C. Sekar, and A. Voronkov. Term indexing. In Robinson and Voronkov [13], pages 1853–1964.

[12] J. Rao and K. A. Ross. Making $B^+$-trees cache conscious in main memory. In *ACM SIGMOD International Conference on Management of Data*, pages 475–486, 2000.

[13] J. A. Robinson and A. Voronkov, editors. *Handbook of Automated Reasoning (in 2 volumes)*. Elsevier and MIT Press, 2001.

[14] S. Schulz. E - a brainiac theorem prover. *AI Communication*, 15(2-3):111–126, 2002.

[15] S. Schulz and M. P. Bonacina. On Handling Distinct Objects in the Superposition Calculus. In B. Konev and S. Schulz, editors, *Proc. of the 5th International Workshop on the Implementation of Logics, Montevideo, Uruguay*, pages 66–77, 2005.

[16] J. R. Slagle. Automatic theorem proving with built-in theories including equality, partial ordering, and sets. *J. ACM*, 19(1):120–135, 1972.

[17] F. M. Suchanek, G. Kasneci, and G. Weikum. Yago: A Core of Semantic Knowledge. In *16th international World Wide Web conference (WWW 2007)*, pages 697–706, New York, NY, USA, 2007. ACM Press.

[18] F. M. Suchanek, G. Kasneci, and G. Weikum. YAGO: A Large Ontology from Wikipedia and WordNet. *J. Web Sem.*, 6(3):203–217, 2008.

[19] M. Suda, G. Sutcliffe, P. Wischnewski, M. Lamotte-Schubert, and G. de Melo. External sources of axioms in automated theorem proving. In B. Mertsching, M. Hund, and M. Z. Aziz, editors, *KI 2009: Advances in Artificial Intelligence, 32nd Annual German Conference on AI, Paderborn, Germany, September 15-18, 2009. Proceedings*, volume 5803 of *LNCS*, pages 281–288. Springer, 2009.

[20] M. Suda, C. Weidenbach, and P. Wischnewski. On the Saturation of YAGO. Research Report MPI-I-2010-RG1-001, Max-Planck-Institut für Informatik, Saarbrücken, 2010.

[21] G. Sutcliffe. The 4th IJCAR Automated Theorem Proving System Competition - CASC-J4. *AI Communication*, 22(1):59–72, 2009.

[22] T. Tammet. Chain resolution for the semantic web. In D. A. Basin and M. Rusinowitch, editors, *IJCAR*, volume 3097 of *LNCS*, pages 307–320. Springer, 2004.

[23] T. Tammet and V. Kadarpik. Combining an inference engine with database: A rule server. In M. Schroeder and G. Wagner, editors, *RuleML*, volume 2876 of *LNCS*, pages 136–149. Springer, 2003.

[24] A. Voronkov. Merging relational database technology with constraint technology. In D. Bjørner, M. Broy, and I. V. Pottosin, editors, *Ershov Memorial Conference*, volume 1181 of *LNCS*, pages 409–419. Springer, 1996.

[25] C. Weidenbach. Combining superposition, sorts and splitting. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume 2, chapter 27, pages 1965–2012. Elsevier, 2001.

[26] C. Weidenbach, D. Dimova, A. Fietzke, M. Suda, and P. Wischnewski. SPASS Version 3.5. In R. A. Schmidt, editor, *22nd International Conference on Automated Deduction (CADE-22)*, volume 5663 of *LNAI*, pages 140–145. Springer, 2009.

Below you find a list of the most recent technical reports of the Max-Planck-Institut für Informatik. They are available via WWW using the URL `http://www.mpi-inf.mpg.de`. If you have any questions concerning WWW access, please contact `reports@mpi-inf.mpg.de`. Paper copies (which are not necessarily free of charge) can be ordered either by regular mail or by e-mail at the address below.


Max-Planck-Institut für Informatik

Library

attn. Anja Becker

Stuhlsatzenhausweg 85

66123 Saarbrücken

GERMANY

e-mail: `library@mpi-inf.mpg.de`


---

| | | |
|---|---|---|
| MPI-I-2009-RG1-002 | P. Wischnewski, C. Weidenbach | Contextual rewriting |
| MPI-I-2009-5-006 | S. Bedathur, K. Berberich, J. Dittrich, N. Mamoulis, G. Weikum | Scalable phrase mining for ad-hoc text analytics |
| MPI-I-2009-5-004 | N. Preda, F.M. Suchanek, G. Kasneci, T. Neumann, G. Weikum | Coupling knowledge bases and web services for active knowledge |
| MPI-I-2009-5-003 | T. Neumann, G. Weikum | The RDF-3X engine for scalable management of RDF data |
| MPI-I-2008-RG1-001 | A. Fietzke, C. Weidenbach | Labelled splitting |
| MPI-I-2008-5-004 | F. Suchanek, M. Sozio, G. Weikum | SOFI: a self-organizing framework for information extraction |
| MPI-I-2008-5-003 | F.M. Suchanek, G. de Melo, A. Pease | Integrating Yago into the suggested upper merged ontology |
| MPI-I-2008-5-002 | T. Neumann, G. Moerkotte | Single phase construction of optimal DAG-structured QEPs |
| MPI-I-2008-5-001 | F. Suchanek, G. Kasneci, M. Ramanath, M. Sozio, G. Weikum | STAR: Steiner tree approximation in relationship-graphs |
| MPI-I-2008-4-003 | T. Schultz, H. Theisel, H. Seidel | Crease surfaces: from theory to extraction and application to diffusion tensor MRI |
| MPI-I-2008-4-002 | W. Saleem, D. Wang, A. Belyaev, H. Seidel | Estimating complexity of 3D shapes using view similarity |
| MPI-I-2008-1-001 | D. Ajwani, I. Malinger, U. Meyer, S. Toledo | Characterizing the performance of Flash memory storage devices and its impact on algorithm design |
| MPI-I-2007-RG1-002 | T. Hillenbrand, C. Weidenbach | Superposition for finite domains |
| MPI-I-2007-5-003 | F.M. Suchanek, G. Kasneci, G. Weikum | Yago : a large ontology from Wikipedia and WordNet |
| MPI-I-2007-5-002 | K. Berberich, S. Bedathur, T. Neumann, G. Weikum | A time machine for text search |
| MPI-I-2007-5-001 | G. Kasneci, F.M. Suchanek, G. Ifrim, M. Ramanath, G. Weikum | NAGA: searching and ranking knowledge |
| MPI-I-2007-4-008 | J. Gall, T. Brox, B. Rosenhahn, H. Seidel | Global stochastic optimization for robust and accurate human motion capture |
| MPI-I-2007-4-007 | R. Herzog, V. Havran, K. Myszkowski, H. Seidel | Global illumination using photon ray splatting |
| MPI-I-2007-4-006 | C. Dyken, G. Ziegler, C. Theobalt, H. Seidel | GPU marching cubes on shader model 3.0 and 4.0 |
| MPI-I-2007-4-005 | T. Schultz, J. Weickert, H. Seidel | A higher-order structure tensor |
| MPI-I-2007-4-004 | C. Stoll, E. de Aguiar, C. Theobalt, H. Seidel | A volumetric approach to interactive shape editing |
| MPI-I-2007-4-003 | R. Bargmann, V. Blanz, H. Seidel | A nonlinear viseme model for triphone-based speech synthesis |

| | | |
|---|---|---|
| MPI-I-2007-4-002 | T. Langer, H. Seidel | Construction of smooth maps with mean value coordinates |
| MPI-I-2007-4-001 | J. Gall, B. Rosenhahn, H. Seidel | Clustered stochastic optimization for object recognition and pose estimation |
| MPI-I-2007-2-001 | A. Podelski, S. Wagner | A method and a tool for automatic veriication of region stability for hybrid systems |
| MPI-I-2007-1-003 | A. Gidenstam, M. Papatriantafilou | LFthreads: a lock-free thread library |
| MPI-I-2007-1-002 | E. Althaus, S. Canzar | A Lagrangian relaxation approach for the multiple sequence alignment problem |
| MPI-I-2007-1-001 | E. Berberich, L. Kettner | Linear-time reordering in a sweep-line algorithm for algebraic curves intersecting in a common point |
| MPI-I-2006-5-006 | G. Kasnec, F.M. Suchanek, G. Weikum | Yago - a core of semantic knowledge |
| MPI-I-2006-5-005 | R. Angelova, S. Siersdorfer | A neighborhood-based approach for clustering of linked document collections |
| MPI-I-2006-5-004 | F. Suchanek, G. Ifrim, G. Weikum | Combining linguistic and statistical analysis to extract relations from web documents |
| MPI-I-2006-5-003 | V. Scholz, M. Magnor | Garment texture editing in monocular video sequences based on color-coded printing patterns |
| MPI-I-2006-5-002 | H. Bast, D. Majumdar, R. Schenkel, M. Theobald, G. Weikum | IO-Top-k: index-access optimized top-k query processing |
| MPI-I-2006-5-001 | M. Bender, S. Michel, G. Weikum, P. Triantafilou | Overlap-aware global df estimation in distributed information retrieval systems |
| MPI-I-2006-4-010 | A. Belyaev, T. Langer, H. Seidel | Mean value coordinates for arbitrary spherical polygons and polyhedra in $\mathbb{R}^3$ |
| MPI-I-2006-4-009 | J. Gall, J. Potthoff, B. Rosenhahn, C. Schnoerr, H. Seidel | Interacting and annealing particle filters: mathematics and a recipe for applications |
| MPI-I-2006-4-008 | I. Albrecht, M. Kipp, M. Neff, H. Seidel | Gesture modeling and animation by imitation |
| MPI-I-2006-4-007 | O. Schall, A. Belyaev, H. Seidel | Feature-preserving non-local denoising of static and time-varying range data |
| MPI-I-2006-4-006 | C. Theobalt, N. Ahmed, H. Lensch, M. Magnor, H. Seidel | Enhanced dynamic reflectometry for relightable free-viewpoint video |
| MPI-I-2006-4-005 | A. Belyaev, H. Seidel, S. Yoshizawa | Skeleton-driven laplacian mesh deformations |
| MPI-I-2006-4-004 | V. Havran, R. Herzog, H. Seidel | On fast construction of spatial hierarchies for ray tracing |
| MPI-I-2006-4-003 | E. de Aguiar, R. Zayer, C. Theobalt, M. Magnor, H. Seidel | A framework for natural animation of digitized models |
| MPI-I-2006-4-002 | G. Ziegler, A. Tevs, C. Theobalt, H. Seidel | GPU point list generation through histogram pyramids |
| MPI-I-2006-4-001 | A. Efremov, R. Mantiuk, K. Myszkowski, H. Seidel | Design and evaluation of backward compatible high dynamic range video compression |
| MPI-I-2006-2-001 | T. Wies, V. Kuncak, K. Zee, A. Podelski, M. Rinard | On verifying complex properties using symbolic shape analysis |
| MPI-I-2006-1-007 | H. Bast, I. Weber, C.W. Mortensen | Output-sensitive autocompletion search |
| MPI-I-2006-1-006 | M. Kerber | Division-free computation of subresultants using bezout matrices |
| MPI-I-2006-1-005 | A. Eigenwillig, L. Kettner, N. Wolpert | Snap rounding of Bézier curves |
| MPI-I-2006-1-004 | S. Funke, S. Laue, R. Naujoks, L. Zvi | Power assignment problems in wireless communication |
| MPI-I-2005-5-002 | S. Siersdorfer, G. Weikum | Automated retraining methods for document classification and their parameter tuning |
| MPI-I-2005-4-006 | C. Fuchs, M. Goesele, T. Chen, H. Seidel | An emperical model for heterogeneous translucent objects |
| MPI-I-2005-4-005 | G. Krawczyk, M. Goesele, H. Seidel | Photometric calibration of high dynamic range cameras |
| MPI-I-2005-4-004 | C. Theobalt, N. Ahmed, E. De Aguiar, G. Ziegler, H. Lensch, M.A. Magnor, H. Seidel | Joint motion and reflectance capture for creating relightable 3D videos |
| MPI-I-2005-4-003 | T. Langer, A.G. Belyaev, H. Seidel | Analysis and design of discrete normals and curvatures |

MPI-I-2005-4-002 O. Schall, A. Belyaev, H. Seidel Sparse meshing of uncertain and noisy surface scattered data