# Towards a Substitution Tree Based Index for Higher-order Resolution Theorem Provers

Tomer Libal
Inria Saclay
Palaiseau, France
tomer.libal@inria.fr

Alexander Steen
Freie Universität Berlin
Berlin, Germany
a.steen@fu-berlin.de

## Abstract

One of the reasons that forward search methods, like resolution, are efficient in practice is their ability to utilize many optimization techniques. One such technique is subsumption and one way of utilizing subsumption efficiently is by indexing terms using substitution trees. In this paper we describe an attempt to extend such indexes for the use of higher-order resolution theorem provers. Our attempt tries to handle two difficulties which arise when extending the indexes to higher-order. The first difficulty is the need for higher-order anti-unification. The second difficulty is the closure of clauses under associativity and commutativity. We present some techniques which attempt to solve these two problems.

## 1 Introduction

Term indexing is a popular technique for speeding up computations to a broad variety of tools in computational logic, ranging from resolution based theorem provers [17] to interpreters of programming languages such as Prolog. A key aspect for optimizing such tools is to use sophisticated methods for redundancy elimination. One such method for clause-based indexes is subsumption. Forward subsumption allows to discard clauses which are less general than clauses already processed, whereas backward subsumption allows to remove stored clauses if a more general one is encountered. The idea behind both is that using more general clauses drastically reduces the search space while still being enough for completing certain tasks [9]. Due to the importance of subsumption in practical applications, there is a variety of indexing techniques that support efficient subsumption queries. We refer the reader to the survey about term indexing by Ramakrishnan et al. [16].

In a similar way to tools for first-order logic, higher-order logic tools can benefit also from term indexing for efficient subsumption. Unfortunately, higher-order matching, which is a required technique for determining subsumption, is much more complex than its first-order counter part [21]. We only know of few approaches to higher-order term indexing: Theiß and Benzmüller designed a term index [22] for the LEO-II higher-order resolution theorem prover [3]. However, their approach focuses on efficient low-level term (traversal) operations such as $\beta$-normalization and occur-checks. Additionally, term sharing is employed in order to reduce space consumption and for allowing constant-time equality checks between $\alpha$-equivalent terms.

The closest to our approach is the higher-order term indexing technique by Pientka [14]. Both approaches are based on substitution trees [6] and on propagating to the leaves those components of the indexed terms for

In: P. Fontaine, S. Schulz, J. Urban (eds.): Proceedings of the 5th Workshop on Practical Aspects of Automated Reasoning (PAAR 2016), Coimbra, Portugal, 02-07-2016, published at http://ceur-ws.org

which there is no efficient unification algorithm. One difference is the assumed term representation: In order to efficiently manage more complex type systems, such as dependent types, Pientka has chosen to represent terms using Contextual Modal Type Theory [12]. This technique allows for an elegant treatment of dependent types but requires a specially designed unification procedure [15]. Our use of the standard simple type theory [4] allows us to use type-independent unification algorithms [2]. Another difference is the technique chosen for propagating the non-pattern content: While we use an efficient algorithm for computing the pattern generalization of two non-pattern terms [2] (also made possible by our choice of term representation), Pientka's technique is based on a normalizing pre-processing step [15] which computes the non-pattern content as additional constraints and might incur an additional cost. A somewhat less crucial difference is the treatment of associativity and commutativity (AC). Since unification algorithms which incorporate this theory do not exist for higher-order logic, we deal with the problem by integrating the treatment of AC into the operations of the index. Pientka's index primary target are not terms in clausal form and therefore, this problem is not treated there. It should not be too complex, though, to integrate the ideas presented in this paper in Pientka's index in order to achieve the same AC treatment. Obvious other differences include the state of the art of Pientka's index, its implementation within the Twelf system [13], and the experimental results which are included and its rigorous representation. Our index is yet to be implemented and experimented with and we are still to provide fully rigorous presentation. Nevertheless, we believe that our approach might be more suitable for indexing higher-order terms in general higher-order theorem provers. This claim is, of course, to be justified by the implementation and experimentation of both indexing techniques within Leo-III [24].

In this paper we hence present a higher-order indexing technique for the theorem prover Leo-III which is also based on substitution trees. The differences just discussed and especially the treatment of non-pattern terms might suggest that forward and backward subsumption operations can be efficiently handled by our approach.

The main difficulty which arises when trying to store clauses in an index is that the index must be closed under the AC properties of clauses. In addition, when computing subsumed clauses, the number of literals in the clause is not as important as the fact that each literal in the subsuming clauses must generalize a literal in the subsumed one. As long as we only treat unit clauses, no special treatment is required and the technique presented in [7] can be safely extended to deal with higher-order terms. When dealing with multi-literal clauses, though, one has to compensate between optimizing the size of the index and optimizing the operations over the index. As can be seen in [23], one cannot avoid an expensive backtracking search.

We suggest a different approach which is supposed to take advantage on searching the index in parallel. In order to achieve that, we plan to store each literal independently in the index and on subsumption calls, to retrieve and compare the literals in parallel. Due to the fact that Leo-III is based on a multi-agent architecture [20], we hope that such an approach would be efficient in practice.

Since we are using substitution trees, we are still faced with the problem of using the costly higher-order unification and anti-unification procedures. In the presented work we somehow avoid this problem by using a variant of higher-order anti-unification [2] which computes pattern [11] substitutions. The use of this algorithm will allow us to maintain a substitution tree all of whose inner nodes are pattern substitutions, on which unification and anti-unification are efficient.

The definitions and properties of our index are still being investigated and some of them are not yet formally proved. Nevertheless, we hope that the arguments and examples will convince the reader about the potential of our approach for the indexing of arbitrary higher-order clauses and for the support of the forward and backward subsumption functions over this index.

In the next section we present the necessary definitions required for understanding this paper as well as the basic ideas of substitution trees and higher-order anti-unification. Following this section is the main part of the paper, in which we introduce our notion of higher-order substitution trees and define the insert, delete, retrieve and subsumption functions. We close our paper with a conclusion which also describes potential future work.

## 2 Preliminaries

In this section we present the logical language that is used throughout the paper. The language is a version of Church's simple theory of types [4] with an $\eta$-conversion rule as presented in [1] and with implicit $\alpha$-conversions. Unless stated otherwise, all terms are implicitly converted into $\beta$-normal and $\eta$-expanded form.

Let $\mathfrak{T}_{\mathfrak{o}}$ be a set of basic types, then the set of types $\mathfrak{T}$ is generated by $\mathfrak{T} := \mathfrak{T}_{\mathfrak{o}} \mid \mathfrak{T} \to \mathfrak{T}$. Let $\mathfrak{C}$ be a signature of function symbols and let $\mathfrak{V}$ be a countably infinite set of variable symbols. In our definitions and examples the symbols $u, w, x, y, z, W, X, Y, Z \in \mathfrak{V}$, and $f, g, h, k, a, b, c \in \mathfrak{C}$ are used. We sometimes use subscripts and superscripts as well. The set $\mathtt{Term}^\alpha$ of terms of type $\alpha$ is generated by $\mathtt{Term}^\alpha := f^\alpha \mid x^\alpha \mid (\lambda x^\beta.\mathtt{Term}^\gamma) \mid (\mathtt{Term}^{\beta\to\alpha}\mathtt{Term}^\beta)$ where $f \in \mathfrak{C}, x \in \mathfrak{V}$ and $\alpha \in \mathfrak{T}$ (in the abstraction, $\alpha = \beta \to \gamma$). Applications throughout the paper will be associated to the left. We will sometimes omit brackets when the meaning is clear. We will also normally omit typing information when it is not crucial for the correctness of the results. $\tau(t^\alpha) = \alpha$ refers to the type of a term. The set $\mathtt{Term}$ denotes the set of all terms. *positions* are defined as usual. We denote the subterm of $t$ at position $p$ by $t|_p$. *Bound* and *free variables* are defined as usual. Given a term $t$, we denote by $\mathtt{hd}(t)$ its *head symbol*.

*Substitutions* and their *composition* ($\circ$) are defined as usual, namely $(\sigma \circ \theta)(X) = \theta(\sigma(X))$. The *domain* and *codomain* of a substitution $\sigma$ are denoted by $\mathtt{dom}(\sigma)$ and $\mathtt{codom}(\sigma)$. The *image* of $\sigma$ is the set of all variables in $\mathtt{codom}(\sigma)$. We denote by $\sigma|_W$ the substitution obtained from substitution $\sigma$ by restricting its domain to variables in $W$. We denote by $\sigma[X \mapsto t]$ the substitution obtained from $\sigma$ by mapping $X$ to $t$, where $X$ might already exist in the domain of $\sigma$. The *join* of two substitutions $\sigma$ and $\theta$ is denoted $\sigma \bullet \theta$ (cf. [7]). We extend the application of substitutions to terms in the usual way and denote it by postfix notation. Variable capture is avoided by implicitly renaming variables to fresh names upon binding. A substitution $\sigma$ is *more general* than a substitution $\theta$, denoted $\sigma \leq \theta$, if there is a substitution $\delta$ such that $\sigma \circ \delta = \theta$. Similarly, a substitution $\sigma$ is the *most specific generalization* of substitutions $\tau$ and $\theta$ if $\sigma \leq \tau$, $\sigma \leq \theta$ and there is no other substitution $\delta$ fulfilling these properties such that $\delta > \sigma$. A substitution $\sigma$ matches a substitution $\tau$ if there is a substitution $\delta$ such that $\delta \circ \tau = \sigma$. A *complete set of matchers* between substitutions $\sigma$ and $\tau$ is a set $A$ of substitutions such that $A$ contains all the matching substitutions between $\sigma$ and $\tau$. A substitution $\sigma$ is a *renaming substitution* if $\mathtt{codom}(\sigma) \subseteq \mathfrak{V}$ and $|\mathtt{codom}(\sigma)| = |\mathtt{dom}(\sigma)|$. The predicate $\mathtt{rename}(\sigma)$ is true iff $\sigma$ is a renaming substitution. We denote the *inverse* of a renaming substitution $\sigma$ by $\mathtt{inverse}(\sigma)$.

### 2.1 Substitution Trees

This section describes substitution trees based on those defined in [6]. In order to optimize some functions on trees, the definition in [6] uses normalized terms and substitutions. As we will see, we will insert the literals of a clause independently into the index and therefore, if we normalize them as suggested in [6] the relationship between the free variables among the different literals will be lost. The main differences, therefore, between our presentation and that in [6] is that we will avoid normalizing terms and substitutions and in addition, allow terms to be of arbitrary order.

**Definition 1** (Substitution Trees). *A substitution tree is defined inductively and is either the empty tree $\epsilon$ or the tuple $(\sigma, \Pi)$ where $\sigma$ is a substitution and $\Pi$ is a set of substitution trees such that*

1. *each node in the tree is either a leaf node $(\sigma, \emptyset)$ or an inner node $(\sigma, \Pi)$ with $|\Pi| \geq 2$.*

2. *for every branch $(\sigma_1, \Pi_1), \ldots, (\sigma_n, \Pi_n)$ in a non-empty tree we have $\mathtt{dom}(\sigma_i) \cap (\mathtt{dom}(\sigma_1) \cup \ldots \cup \mathtt{dom}(\sigma_{i-1})) = \emptyset$ for all $0 < i \leq n$.*

### 2.2 Higher-order Anti-unification

Anti-unification denotes the problem of finding a generalization $t$ of two given terms $t_1$ and $t_2$, i.e. a term $t$ such that there exist substitutions $\sigma_1, \sigma_2$ such that $t\sigma_1 = t_1$ and $t\sigma_2 = t_2$. A key algorithm for the procedures which are described in the remainder of this paper is the higher-order anti-unification algorithm of Baumgartner et al. [2]. This algorithm differs from most higher-order unification and anti-unification procedures not only by being applicable to arbitrary (simply typed) higher-order terms, but also by efficiently computing very specific generalizations.

By *very specific generalization* (in contrast to the most specific one) we here mean the most specific higher-order pattern which generalizes two arbitrary higher-order terms. This pattern, however, might not be the most specific generalization of these two higher-order terms. Higher-order patterns are restricted forms of higher-order terms for which it is known that efficient unification algorithms exist [11].

Details about the higher-order pattern fragment or even the above anti-unification algorithm are not crucial for understanding this paper and are therefore omitted. It is important to note that since only most specific *pattern generalizations* are found, the size of the index described in this paper is not optimal. We will explain this point in more detail later.

The anti-unification algorithm of Baumgartner et al. is subsequently denoted by $\mathtt{msg}^*$.

**Definition 2** (Algorithm $\mathtt{msg}^*$ [2]). *The algorithm* $\mathtt{msg}^*$ *takes two arbitrary higher-order terms* $t_1$ *and* $t_2$ *as input and returns a higher-order pattern* $s$ *and substitutions* $\sigma_1$ *and* $\sigma_2$ *such that*

1. $s\sigma_1 = t_1$ *and* $s\sigma_2 = t_2$*, and*
2. *there is no other higher-order pattern* $s'$ *fulfilling the above property such that there is a non-trivial substitution* $\delta$ *where* $s\delta = s'$*.*

Baumgartner et al. showed that the algorithm $\mathtt{msg}^*$ computes a unique solution (up to renaming of free variables) and takes cubic time [2]. In this paper we are interested in a variant of this algorithm which computes substitutions, rather than terms and which is defined as follows:

**Definition 3** (Most specific pattern generalizing substitution). *Given substitutions* $\theta_1$ *and* $\theta_2$*, the substitution* $\sigma$ *is the* most specific pattern generalizing substitution *if there are substitutions* $\tau_1$ *and* $\tau_2$ *such that* $\mathtt{codom}(\sigma)$ *contains only higher-order patterns,* $\sigma \circ \tau_1 = \theta_1$*,* $\sigma \circ \tau_2 = \theta_2$ *and there is no substitution* $\sigma' > \sigma$ *fulfilling these properties.*

An algorithm for computing the most specific pattern generalizing substitution, denoted $\mathtt{msg}$, can be defined on top of $\mathtt{msg}^*$.

**Definition 4** (The algorithm $\mathtt{msg}$). *The algorithm* $\mathtt{msg}$ *takes two substitutions* $\theta_1, \theta_2$ *as input and returns a triple* $(\sigma, \tau_1, \tau_2)$ *with* $\sigma, \tau_1, \tau_2$ *as in Def. 3. To that end, let* $\mathtt{dom}(\theta_1) \cup \mathtt{dom}(\theta_2) = \{x_1, \ldots, x_n\}$*. Let* $(f(s_1, \ldots, s_n), \tau_1, \tau_2) = \mathtt{msg}^*(f(\theta_1(x_1), \ldots, \theta_1(x_n)), f(\theta_2(x_1), \ldots, \theta_2(x_n)))$ *where* $f$ *is a new function symbol of arity* $n$*. Finally, set* $\sigma := \{x_1 \mapsto s_1, \ldots, x_n \mapsto s_n\}$*.*

**Claim 5.** *Let* $\theta_1, \theta_2$ *be two substitutions and let* $(\sigma, \tau_1, \tau_2) = \mathtt{msg}(\theta_1, \theta_2)$*. Then* $\sigma$ *is a most specific pattern generalizing substitution of* $\theta_1$ *and* $\theta_2$*. Also,* $\sigma$ *is unique up to renaming of free variables.*

The algorithm $\mathtt{msg}$ takes cubic time, hence can be used to efficiently build up a substitution tree index (cf. next section).

## 3 Substitution Trees for Higher-order Clauses

In this section we will describe some modifications to first-order substitution trees which will allow us to extend them to higher-order terms.

The most obvious obstacle in extending the trees to higher-order terms is the fact that substitution trees depend on procedures for unification, anti-unification and matching. These procedures, while being both relatively efficient and unitary in classical first-order logic [10], are highly complex in higher-order logics and do not possess unique solutions any more [8, 19].

Another obstacle is the fact that since we are targeting resolution theorem provers, the terms we are going to store, retrieve, delete and check for subsumption are not mere syntactic terms but clauses with are closed under AC. In the first-order case, one can use dedicated unification algorithms which, although not unitary any more [5], are still feasible. In the higher-order case, due to the complex nature of even the syntactic unification procedure, one needs to find another approach.

Our solution to the first problem is to relax a core property of substitution trees and allow also non least general generalizations as substitutions in the nodes of the trees. Towards this end, we employ the anti-unification algorithm from section 2.2. The use of this algorithm will render our trees less optimal as nodes may now contain more general substitutions than necessary and therefore one child may be more general than another child of a node. On the other hand, the algorithm is only cubic in time complexity and is unitary.

Our approach to the second problem is to handle the AC properties of clauses not on the anti-unification or matching level, but to encode their treatment into the retrieval, insertion, deletion and subsumption functions. We obtain this by regarding each literal of a clause as an independent higher-order term and, in addition, assigning labels that are identical for all literals of the same clause.

Classical substitution trees depend on the anti-unification algorithm for treating associativity and commutativity as well as other properties required by subsumption, such as one clause being a sub-clause of the other. If such an anti-unification algorithm for higher-order term can be found, a simple extension to the trees in section 2.1 can be defined which enjoys the same definitions for insert, delete and retrieval as defined in [6]. This extension will also preserve the property of substitution trees that the index does not contain variants of substitutions already stored and the deletion function removes all variants of some input substitution.

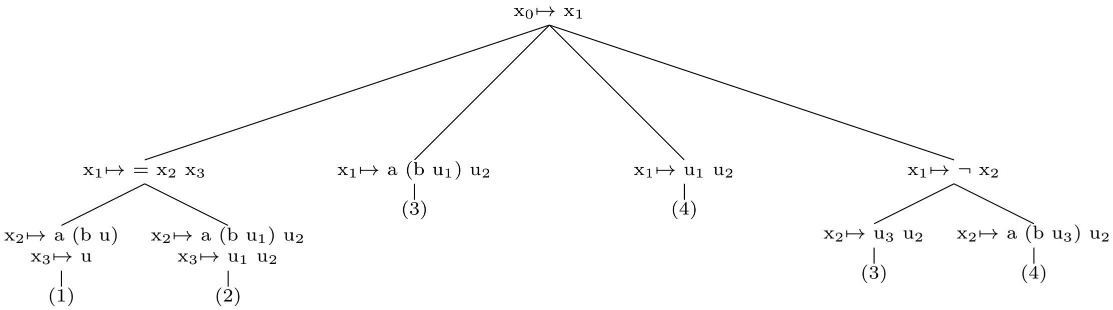


Figure 1: An example of a substitution tree

Since it is not clear if such an anti-unification procedure can be practical, we choose a different approach which will greatly change the form of the substitution trees. Our treatment of subsumption requires the substitution trees to contain all inserted literals, including those which are variants of already stored ones. Similarly, delete will only remove certain substitutions and not all variants of an input substitution. In addition, we will have all stored substitutions be associated with a set of labels which denotes the clause they are part of. In this way we can insert and retrieve the literals independently for a given clause.

We will now give the refined definition of higher-order substitution trees.

**Definition 6** (Higher-order Substitution Trees)**.** *A higher-order substitution tree is the triple $(\sigma, \Pi, L)$ where $\sigma$ is a substitution, $\Pi$ is a set of substitution trees and $L$ is a set of labels such that the following properties hold:*

- *if $L = \emptyset$, then $\Pi$ is not empty.*
- *if $\Pi$ is not empty, then $\sigma$ is a pattern substitution.*

From now on we will refer to higher-order substitution trees just as substitution trees.

The above definition makes retrieval in the tree much more efficient as the traversal of the tree will always use the algorithm `msg` and not less efficient algorithms.

Our trees will always have a root node $\{x_0 \mapsto x_1\}$ and will therefore never be empty. This is done in order to simplify the insertion operation as the root node will always be more general than any inserted term.

**Example 7.** *We use as running example, the manipulation of the index as done by Leo-III [24] when running on a variant of Cantor's surjective theorem. The first six clauses which are inserted are the following (where $\alpha$ and $\beta$ are types, $\alpha := \iota \to \beta$ and $\beta := \iota \to o$):*

*(1)* $= (a^{\alpha}\ (b^{\beta\to\iota}\ u^{\beta}))\ u^{\beta}$

*(2)* $= (a^{\alpha}\ (b^{\beta\to\iota}\ u_1^{\beta})\ u_2^{\iota})\ (u_1^{\beta}\ u_2^{\iota})$

*(3)* $\neg\ (u_3^{\beta}\ u_2^{\iota}) \vee (a^{\alpha}\ (b^{\beta\to\iota}\ u_1^{\beta})\ u_2^{\iota})$

*(4)* $(u_1^{\beta}\ u_2^{\iota}) \vee \neg\ (a^{\alpha}\ (b^{\beta\to\iota}\ u_3^{\beta})\ u_2^{\iota})$

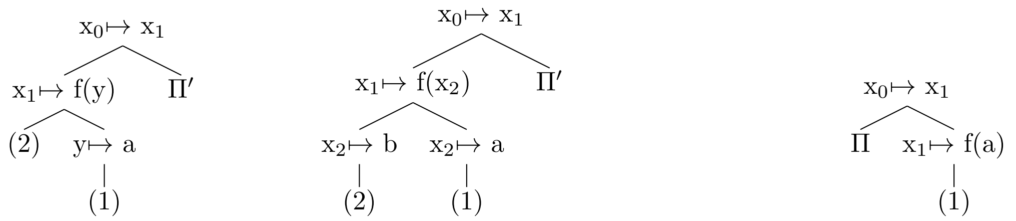
*(5)* $(a^{\alpha}\ (b^{\beta\to\iota}\ u_1^{\beta})\ u_2^{\iota}) \vee \neg\ (a^{\alpha}\ (b^{\beta\to\iota}\ u_3^{\beta})\ u_2^{\iota})$

*(6)* $(u_1^{\beta}\ u_2^{\iota}) \vee \neg\ (u_3^{\beta}\ u_2^{\iota})$

*Here, $\vee$ denotes the union of single literals. Note that clause (5) is subsumed by both clauses (3) and (4) and that clause (6) subsumes clauses (3), (4) and (5). We will use this clause set to demonstrate the insert, retrieval and delete operations of the higher-order substitution trees. Fig. 1 displays the higher-order substitution tree after the insertion of the first four clauses.*

The above example demonstrates why our trees are not optimized as for example, one of the children of node $x_0 \mapsto x_1$, the node $x_1 \mapsto a\ (b\ u_1)\ u_2$ is less general than the child $x_1 \mapsto u_1\ u_2$. Also, it demonstrates the problem of using arbitrary higher-order terms in the inner nodes of the tree as there are many possible substitutions which generalizes $x_1 \mapsto u_1\ u_2$ to $x_1 \mapsto a\ (b\ u_1)\ u_2$, but only one most specific pattern generalization, $x_0 \mapsto x_1$.

In our trees, each branch from the root of the tree to a leaf corresponds to a literal of some clause. Since we will need to know the actual substitutions at these leaves, we introduce the notation of *composed substitutions*:



(a) Variant insertion (b) Compatible insertion (c) Non-compatible (or empty) insertion

Figure 2: Cases of inserting a substitution into the index from Ex. 10.

**Definition 8** (Composed Substitutions)**.** *Given a substitution tree $T = (\sigma, \Pi, L)$ and let $\tau$ be a substitution at a leaf of $T$ such that the nodes on the branch from the root of $T$ to $\tau$ are $(\sigma_1, \ldots, \sigma_n)$. Then, the* composed substitution for $\tau$*, denoted* $\texttt{comp-sub}(\tau)$*, is given by* $\texttt{comp-sub}(\tau) = \sigma_1 \bullet \ldots \bullet \sigma_n$*.*

For example, in Fig. 1, consider the leftmost leaf labeled (1). Here, the composed substitution is given by $\{x_0 \mapsto x_1\} \bullet \{x_1 \mapsto = x_2\ x_3\} \bullet \{x_2 \mapsto a\ (b\ u), x_3 \mapsto u\} = \{x_0 \mapsto = a\ (b\ u)\ u\}$. Note that, in the context of substitution trees, $\sigma \bullet \theta$ equals $\sigma \circ \theta|_{\texttt{dom}(\sigma)}$ if $\sigma$ and $\theta$ are substitutions on a path from the root node to a leaf where $\theta$ occurs directly below (i.e. as a child node of) $\sigma$.

### 3.1 Insertion

We will define now how to insert new elements into the tree. The definition is similar to the insertion function defined in [6]. One difference is the use of `msg` for both finding variants and computing generalizations as well as adding labels to the leaves of the tree. An even more important difference is that since we store clauses and not terms, we must also store in the tree variants of existing nodes.

Given a clause labeled by $l$ for insertion, we insert each literal $t$ of the clause separately. In the following algorithm, we insert to the tree the substitution $\tau = \{x_0 \mapsto t\}$.

**Definition 9** (Insertion Function `insert`)**.** *Let $(\sigma, \Pi, L)$ be a substitution tree, $\tau$ a substitution to be inserted and $l$ the clause label of this substitution. Compute the following set $A = \{(\theta_i, \delta_i^1, \delta_i^2) \mid (\sigma_i, \Pi_i, L_i) \in \Pi, (\theta_i, \delta_i^1, \delta_i^2) = \texttt{msg}(\sigma_i, \tau)\}$. Then, $\texttt{insert}((\sigma, \Pi, L), \tau, l) = (\sigma, \Pi', L)$ where:*

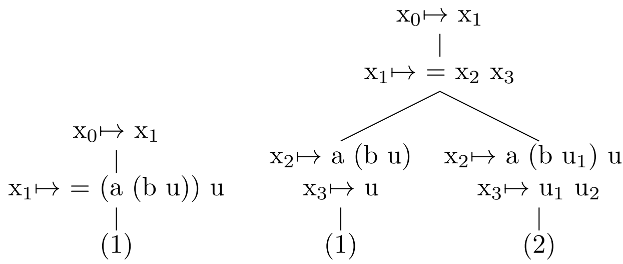
- *(Variant) if there exists $(\theta_i, \delta_i^1, \delta_i^2) \in A$ such that $\delta_i^1$ is a renaming, then $\Pi' = \Pi \setminus \{(\sigma_i, \Pi_i, L_i)\} \cup \{\texttt{insert}((\sigma_i, \Pi_i, L_i), \texttt{inverse}(\delta_i^1) \circ \delta_i^2, l)\}$.*
- *(Compatible) otherwise, if there exists $(\theta_i, \delta_i^1, \delta_i^2) \in A$ such that $\texttt{codom}(\theta_i)$ contains non-variable terms, then $\Pi' = \Pi \setminus \{(\sigma_i, \Pi_i, L_i)\} \cup \{(\theta_i, \{(\delta_i^1, \Pi_i, L_i), (\delta_i^2, \emptyset, \{l\})\})\}$.*
- *(Non compatible or empty) otherwise, let $(\theta, \delta^1, \delta^2) = \texttt{msg}(\sigma, \tau)$ and $\Pi' = \Pi \cup \{(\delta^2, \emptyset, \{l\})\}$.*

**Example 10.** *Assume we want to insert the substitution $\tau = \{x_0 \mapsto f(a)\}$ for clause (1) into the tree $(\{x_0 \mapsto x_1\}, \Pi)$.*

- *if $\Pi = \emptyset$, then we get the tree in Fig. 2c where $\Pi$ is empty.*
- *if $\Pi = \{(\{x_1 \mapsto f(y)\}, \emptyset, \{(2)\})\} \cup \Pi'$, then we have a variant node since $\texttt{msg}(\{x_1 \mapsto f(y)\}, \tau) = (\{x_1 \mapsto f(y)\}, id, \{y \mapsto a\})$ and we get the tree in Fig. 2a.*
- *if there is no variant but $\Pi = \{(\{x_1 \mapsto f(b)\}, \emptyset, \{(2)\})\} \cup \Pi'$, then we have a compatible node since $\texttt{msg}(\{x_1 \mapsto f(b)\}, \tau) = (\{x_1 \mapsto f(x_2)\}, \{x_2 \mapsto b\}, \{x_2 \mapsto a\})$ and $\texttt{codom}(\{x_1 \mapsto f(x_2)\})$ contains non variable symbols and we get the tree in Fig. 2b.*
- *if there is also no compatible child, we get the tree in Fig. 2c.*

The way we insert substitutions into the tree preserves the tree being a substitution tree.

**Claim 11.** *If $T$ is a higher-order substitution tree and $\tau$ a substitution, then $\texttt{insert}(T, \tau, l)$ is also a higher-order substitution tree.*



(a) Inserting clause (1) (b) Inserting clause (2)

Figure 3: Insertion of clauses (1) and (2) to the index from Ex. 12

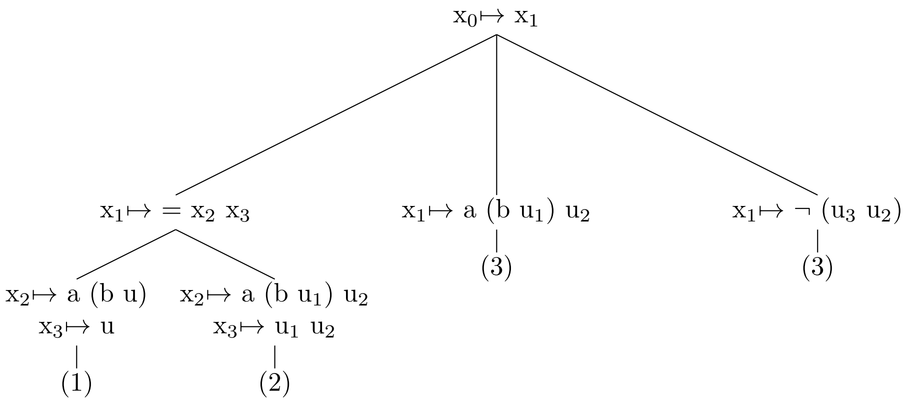


Figure 4: Insertion of clause (3) from Ex. 12

The above property will make retrieval in the tree much more efficient as the traversal of the tree will always use the algorithm `msg` and not less efficient algorithms.

We show next how to insert the clauses of our running example.

**Example 12.** *Figures 3, 4 and 1 show the state of the index after consecutive insertion of clauses (1) and (2), (3), and (4), respectively.*

Note that, in the above tree, it can be seen why the tree is not optimal when using `msg`. Although the nodes having $a\ (b\ u_1)\ u_2$ are instances of $u_1\ u_2$, the tree does not capture it and creates two separate nodes. This happens because $u_1\ u_2$ is not in the pattern fragment.

### 3.2 Deletion

While the insert function defined in the previous section did not differ much from the definition in [6], our definition of the deletion function is completely different. Deletion in first-order substitution trees serves as a logical operation and can be used to perform some limited backward subsumption. Since we store the literals of a clause independently in the tree, we need more information before we can decide if a substitution can be deleted. We will therefore define the deletion function as an optimization function which will remove from the index certain labels of clauses. Since such a deletion can leave some leaves of the tree without labels, we need to recursively optimize the tree by removing and merging nodes.

The formal definition of the deletion function `del` is given by:

**Definition 13** (Deletion Function `del`)**.** *Given a substitution tree $T = (\sigma, \Pi, L)$ and a clause to be removed labeled by $l$, the function $\texttt{del}(T, l)$ is defined as follows:*

- *Let $\Pi' = \bigcup_{T' \in \Pi} \texttt{del}(T', l)$.*
- *if $\sigma = \{x_0 \mapsto x_1\}$ (root) return $(\sigma, \Pi', L \setminus \{l\})$.*
- *else if $\Pi' = \emptyset$ and $L = \{l\}$, then return $\epsilon$.*

- *else if* $\Pi' = \{(\sigma', \Pi'', L'')\}$ *and* $|L \setminus \{l\}| = 0$*, then return* $(\sigma \circ \sigma', \Pi'', L'')$*.*

- *else return* $(\sigma, \Pi', L \setminus \{l\})$*.*

**Example 14.** *After deleting clauses (3) and (4) from the tree in Fig. 1, the result is the tree in Fig. 3b.*

### 3.3 Retrieval

Retrieval in substitution trees is used in order to retrieve all substitutions with a specific relation to some input substitution. Since we are interested in forward and backward subsumption, the relations we are interested in are for the input substitution to be less and more general than the substitutions in the tree, respectively.

In order to support associativity and commutativity of clauses, our substitution trees use a non-standard indexing mechanism where each literal is being stored independently from the other literals of the clause. This will prompt us, for each subsumption call, to try to retrieve all substitutions in the tree with a specific relation to all literals of an input clause. Since all retrieve calls can be done in parallel, Leo-III, with its multi-agent architecture, can take advantage on this approach to the associativity and commutativity of higher-order clauses. It should be noted that since we consider the literals of a clause separately, but a subsumption check requires a common substitution to be applicable to the clause, we need to gather, in addition to the labels, all substitutions that denote the relationship between the literals in the index and the literals of the input clause.

One property of higher-order terms cannot be avoided. In order to retrieve substitutions, a matching algorithm between two arbitrary higher-order substitutions must be used. We have avoided its use when inserting elements by using the anti-unification algorithm from Sec. 2.2. This algorithm will also allow us to traverse the tree when retrieving substitutions and reach a possible matching node according to the definition of substitution trees. The last action of the retrieval operation, the actual matching of a node with the input substitution, requires a stronger algorithm than `msg`. For this operation we will use a standard higher-order matching algorithm. Since the call to this algorithm is performed at most once for each stored substitution and since incomplete higher-order matching algorithms can perform very well in practice, we hope that this step will not impair much the efficiency of the trees. The use of incomplete algorithms is not essential here as a failure to match two substitutions when checking for subsumption might only increase the size of the substitution tree.

In the following, we will assume being given a (possibly incomplete) matching algorithm. Such an algorithm can be, for example, based on Huet's pre-unification procedure [8] with bounds on the depth of terms.

**Definition 15** (Matching Algorithm `match`)**.** *Given two substitutions* $\sigma$ *and* $\tau$*, then* $\texttt{match}(\sigma, \tau) = M$ *where* $M$ *is a complete set of matchers between* $\sigma$ *and* $\tau$*.*

We now describe the two supported retrieve calls, which will both return a set of labels corresponding to an input substitution. Each label will be associated to a set of substitutions. The first of such functions returns all labels of substitutions which are more general than the input argument. Intuitively, this function traverses the tree and uses `msg` for checking if the input substitution is a variant of the respective node. A formal definition is given by

**Definition 16** (Retrieval Function `g-retrieve`)**.** *Given a substitution tree* $T = (\sigma, \Pi, L)$ *and a substitution* $\tau$*,* $\texttt{g-retrieve}(T, \tau, \tau')$ *returns a set of labels associated with substitutions, defined inductively as follows (*$\tau' = \tau$ *at the initial call, but it may change during traversal):*

- *if* $\Pi = \emptyset$ *and* $M = \texttt{match}(\tau, \texttt{comp-sub}(\sigma))$ *such that* $M$ *is not empty, then return* $(L, M)$*.*

- *otherwise, return* $\{(L, \texttt{match}(\tau, \texttt{comp-sub}(\sigma))\} \cup$
  $\{\texttt{g-retrieve}((\sigma', \Pi', L'), \tau, \delta_2) \mid (\sigma', \Pi', L') \in \Pi, \texttt{msg}(\sigma', \tau') = (\theta, \delta_1, \delta_2), \texttt{rename}(\delta_1)\} \cup$
  $\{\texttt{g-retrieve}((\sigma', \emptyset, L'), \tau, \_) \mid (\sigma', \emptyset, L') \in \Pi\}$*.*

**Example 17.** *Let the substitution tree* $T$ *be that from Fig. 1 and let* $\tau = \{x_0 \mapsto (a\ (b\ u_1)\ u_2)\}$*. The execution of* $\texttt{g-retrieve}(T, \tau)$ *proceeds as follows:*

- *apply* `msg` *on all children of the root in order to find either a variant or a leaf, and obtain the two inner nodes* $(\{x_1 \mapsto (a\ (b\ u_1)\ u_2)\}, \emptyset, \{(3)\})$ *and* $(\{x_1 \mapsto (u_1\ u_2)\}, \emptyset, \{(4)\})$*.*

- *recursively apply* `g-retrieve` *on the two leaves.*

- *now apply* `match` *on the two composite substitutions* $\{x_0 \mapsto (a\ (b\ u_1)\ u_2)\}$ *and* $\{x_0 \mapsto (u_1\ u_2)\}$*, in order to obtain the two sets* $\{\delta_1 = \{u_1 \mapsto u_1, u_2 \mapsto u_2\}\}$ *and* $\{\delta_2 = \{u_1 \mapsto \lambda z.(a\ (b\ u_1)\ u_2)\}, \ldots\}$*.*

- *for each leaf, since the result is not empty, the function returns the sets* $\{((3), \{\delta_1\})\}$ *and* $\{((4), \{\delta_2, \ldots\})\}$*.*

*Note that the set of matchers can contain more than one element and even be infinite. We hope to optimize this function in the future.*

**Claim 18.** *The set returned by* `g-retrieve`$((\sigma, \Pi, L), \tau)$ *contains all the labels of substitutions* $\theta$ *which are stored in* $(\sigma, \Pi, L)$ *and which are more general than* $\tau$*. In addition, if a substitution* $\delta$ *is associated with the label of* $\theta$*, then* $\delta \circ \theta = \tau$*.*

The second retrieval function returns all substitutions which are less general than the input substitution. The tree is still traversed using the function `msg` but this time we will not be able to use `msg` to check if $\sigma$ is a variant of $\tau$. This is due to the fact that we used `msg` to check if $\tau$ is a variant of $\sigma$ by checking if $\delta_1$ is a renaming substitution for $\texttt{msg}(\sigma, \tau) = (\theta, \delta_1, \delta_2)$. This worked as both $\sigma$ and $\theta$ are pattern substitutions. If we try to check whether $\sigma$ is a variant of $\tau$, since $\tau$ might not be a pattern substitution, $\delta_2$ might not be a renaming substitution even if it is a variant. On the other hand, if $\sigma$ is a variant of $\tau$, then all the nodes in the subtrees of $\sigma$ are variants of $\tau$, so we need to use `match` only when we reach a node of which $\tau$ is not a variant and stop there.

We first introduce an utility function which gathers all literals on the leaves of a tree.

**Definition 19** (Gathering of labels)**.** *Given a substitution tree* $T = (\sigma, \Pi, L)$ *and a substitution* $\tau$*,* `gather`$(T, \tau) = (L,$ `match`$($`comp-sub`$(\sigma), \tau)) \cup \{$`gather`$(T', \tau) \mid T' \in \Pi\}$*.*

**Example 20.** *The application of* `gather`$(T, \tau)$ *where* $T$ *is the last child of the root in Fig. 1 and* $\tau = \{x_0 \mapsto (u_1\ u_2)\}$ *proceeds as follows:*

- *since there are no labels in the node, it proceed recursively on the two children.*

- *the application on the first child returns* $\{((3), A)\}$*, where* $A = \{u_1 \mapsto \lambda z.\neg(u_3\ u_2), \ldots\}$ *is the result of* `match` *on the composite function* $\{x_0 \mapsto \neg(u_3\ u_2)\}$ *and* $\tau$*.*

- *the application on the second child returns* $\{((4), A)\}$*, where* $A = \{\{u_1 \mapsto \lambda z.\neg(a\ (b\ u_3)\ u_2)\}, \ldots\}$ *is the result of* `match` *on the composite function* $\{x_0 \mapsto \neg(a\ (b\ u_3)\ u_2)\}$ *and* $\tau$*.*

- *return the union of these two sets.*

**Definition 21** (Retrieval Function `i-retrieve`)**.** *Given a substitution tree* $T = (\sigma, \Pi, L)$ *and a substitution* $\tau$*,* `i-retrieve`$(T, \tau, \tau')$ *returns a set of labels associated with substitutions, defined inductively as follows* ($\tau' = \tau$ *at the initial call, but it may change during traversal):*

- *if* `msg`$(\sigma, \tau') = (\theta, \delta_1, \delta_2)$ *such that either* $\delta_1$ *is not a renaming or* $\delta_2$ *is a renaming, and* `match`$($`comp-sub`$(\sigma), \tau) = M$ *such that* $M$ *is not empty, then return* $(L, M) \cup \{$`gather`$(T, \tau) \mid T \in \Pi\}$*.*

- *otherwise, if* $\delta_1$ *is a renaming, return* $\{$`i-retrieve`$(T', \tau, \delta_2) \mid (T') \in \Pi\}$*.*

- *otherwise, return* $\emptyset$*.*

**Example 22.** *Let the substitution tree* $T$ *be the one from Fig. 1 and* $\tau$ *be the substitution* $\{x_0 \mapsto (u_1\ u_2)\}$*. The function* `i-retrieve`$(T, \tau)$ *proceeds as follows:*

- *we first calculate* `msg`$(\{x_0 \mapsto x_1\}, \tau) = (x_0 \mapsto x_1, id, \{x_1 \mapsto (u_1\ u_2)\})$*.*

- *since id is a renaming, we recursively apply* `i-retrieve` *on all nodes.*

- *for all children, the first case of* `i-retrieve` *now holds and we start gathering all the labels in the tree.*

*For brevity, we give the example only for the last node:*

- *as mentioned, the first case now holds for this node as* `msg`$(\{x_1 \mapsto \neg x_2\}, \{x_1 \mapsto (u_1\ u_2)\}) = (\{x_1 \mapsto x_3\}, \{x_3 \mapsto \neg x_2\}, \{x_3 \mapsto (u_1\ u_2)\})$ *and therefore* $\delta_1$ *is not a renaming. On the other hand,* `match`$(\{x_0 \mapsto \neg x_2\}, \{x_0 \mapsto (u_1\ u_2)\}) = \{\{u_1 \mapsto \lambda z.\neg x_2\}, \ldots\}$ *is not empty.*

- *gather all labels on this node, as shown in Ex. 20.*

**Claim 23.** *The set returned by* $\texttt{i-retrieve}((\sigma, \Pi, L), \tau)$ *contains all the labels of substitutions which are stored in* $(\sigma, \Pi, L)$ *and which are less general than* $\tau$*. In addition, if a substitution* $\delta$ *is associated with the label of* $\theta$*, then* $\delta \circ \tau = \theta$*.*

We now show how the checks for forward and backward subsumptions of clauses can be implemented using `g-retrieve` and `i-retrieve`.

### 3.3.1 Forward Subsumption

Forward subsumption checks if an input clause is subsumed by a clause in the index. In case the clause is subsumed, no change to the index is made and the input clause is not inserted into the index.

**Definition 24** (Subsumption)**.** *A clause* $c$ *is subsumed by a clause* $d$ *if* $|c| \leq |d|$ *and there is a substitution* $\sigma$ *such that for each literal* $l$ *of* $d$ *there is a literal* $l'$ *of* $c$ *such that* $l\sigma = l'$*. Here,* $|c|$ *denotes the number of literals in* $c$*.*

Our way of treating associativity and commutativity means that for each literal of the input clause we gather all more general literals in the index with their associated substitution sets. These sets contains all substitutions which independently match the literals in the tree to the literals of the input clause. In order to detect that the input clause is subsumed by the index, we need to show two things. We first need to show that for all the literals of a clause, a more general literal is returned by the index. In addition, we need to show that for each such literal is associated a "compatible" substitution. Two substitutions are "compatible" if they agree on all variables in the intersection of their domains. This requirement means that if $d$ is a clause in the index that subsumes the input clause $c$, then if each of the literals of $d$ subsumes a literal of $c$ with a "compatible" substitution, then there is indeed a substitution $\sigma$ such that each of the literals of $d$ subsumes a literal of $c$ with $\sigma$.

It should be noted that the above technique is based on set subsumption, in contrast to multiset subsumption which is commonly used. A main reason for preferring multisets over sets, is to prevent cases where a clause is being subsumed by a larger clause, which is possible according to the above definition and leads to an incomplete search procedure. In order to restrict such cases, we will only allow subsumption of clasues which are not smaller than the subsuming clauses.

We will now give the formal definition of forward subsumption.

**Definition 25** (Forward Subsumption `fsum`)**.** *Let* $T$ *be a substitution tree and* $c = l_1 \vee \ldots \vee l_n$ *a clause. Let* $A = \{\texttt{g-retrieve}(T, \{x_0 \mapsto l_i\} \mid 0 < i \leq n\}$*. The function* $\texttt{fsum}(T, c)$ *returns true if there are in* $A$ *the labels* $(l, M_1), \ldots, (l, M_k)$ *such that* $k = |l|$ *is the number of literals of clause* $l$*,* $k \leq n$*, and for each two sets* $M_i$ *and* $M_j$ *for* $0 < i < j \leq k$*, there are substitutions* $\sigma_i \in M_i$ *and* $\sigma_j \in M_j$ *such that* $\sigma_i(x) = \sigma_j(x)$ *for all* $x \in \texttt{dom}(\sigma_i) \cap \texttt{dom}(\sigma_j)$*.*

**Example 26.** *We will follow the computation of* $\texttt{fsum}(T, c)$ *where* $T$ *is the substitution tree of Fig. 1 and* $c$ *is clause (5) from Ex. 7.*

- *We first compute the set* $A$ *which is the union of* $\texttt{g-retrieve}(T, \{x_0 \mapsto (a\ (b\ u_1)\ u_2)\})$ *and* $\texttt{g-retrieve}(T, \{x_0 \mapsto \neg(a\ (b\ u_3)\ u_2)\})$*.*

- *the first set was already computed in Ex. 17 and resulted in* $\{((3), \{\{u_1 \mapsto u_1, u_2 \mapsto u_2\}\}), ((4), \{\{u_1 \mapsto \lambda z.(a\ (b\ u_1)\ u_2)\}, \ldots\})\}$*.*

- *the second set is computed in a similar way and results in* $\{((4), \{\{u_3 \mapsto u_3, u_2 \mapsto u_2\}\}), ((3), \{\{u_3 \mapsto \lambda z.(a\ (b\ u_3)\ u_2)\}, \ldots\})\}$*.*

- *since the size of both (3) and (4) is* 2 *and the number of occurrences of the labels of each is* 2*, we are just left with checking if the matching substitutions are compatible, which is easily verified since their domains are disjoint.*

- *we return that (5) is subsumed by the index (by both (3) and (4)).*

**Claim 27.** *If* $\texttt{fsum}(T, c)$ *returns true, then there is a clause* $d$ *indexed by* $T$ *such that* $c$ *is subsumed by* $d$*.*

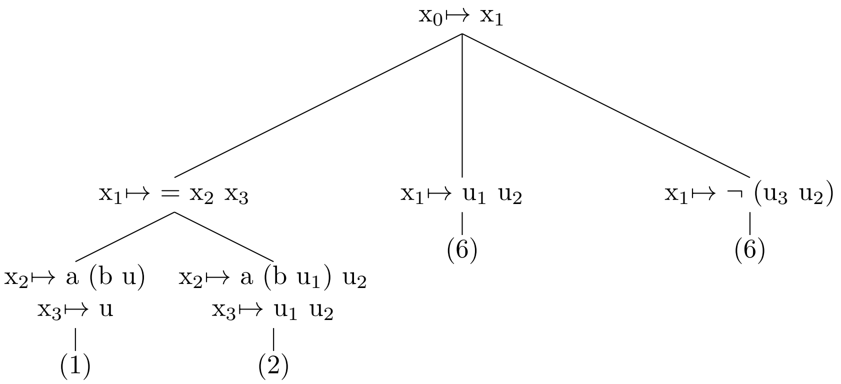


Figure 5: The final substitution tree

### 3.3.2 Backward Subsumption

Similarly to the way we treated forward subsumption, we can also define backward subsumption. When we detect that the index contains clauses which are subsumed by an input clause, we will need to modify the index in order to delete these clauses.

In order to detect all clauses in the index which are subsumed by the input clause, we need to show that the sets returned for each literal of the input clause contain the labels of the subsumed clauses and that the substitution sets associated with the same labels across sets contain ”compatible” substitutions.

**Definition 28** (Backward Subsumption `bsum`)**.** *Let $T$ be a substitution tree and $c = l_1 \vee \ldots \vee l_n$ a clause. Let $A_i = \texttt{i-retrieve}(T, \{x_0 \mapsto l_i\}$ for all $0 < i \leq n$. Label $l \in \texttt{bsum}(T, c)$ if $n \leq |l|$ and $(l, M_i) \in A_i$ for all $0 < i \leq n$ such that for each two sets $M_i$ and $M_j$ for $0 < i < j \leq n$, there are substitutions $\sigma_i \in M_i$ and $\sigma_j \in M_j$ such that $\sigma_i(x) = \sigma_j(x)$ for all $x \in \texttt{dom}(\sigma_i) \cap \texttt{dom}(\sigma_j)$.*

**Example 29.** *We will follow the computation of $\texttt{bsum}(T, c)$ where $T$ is the substitution tree of Fig. 1 and $c$ is clause (6) from Ex. 7.*

- *We first compute the sets $A_1 = \texttt{i-retrieve}(T, \{x_0 \mapsto (u_1\ u_2)\})$ and $A_2 = \texttt{i-retrieve}(T, \{x_0 \mapsto \neg(u_3\ u_2)\})$.*

- *the set $A_1$ was already computed in Ex. 22 and results in $\{((1), \{\{u_1 \mapsto \lambda z. =\ (a\ (b\ u))u\}, \ldots\}), ((2), \{\{u_1 \mapsto \lambda z. =\ (a\ (b\ u_1)\ u_2)(u_1\ u_2)\}, \ldots\}),$ $((3), \{\{u_1 \mapsto \lambda z.(a\ (b\ u_1)\ u_2)\}, \ldots\}), ((4), \{\{u_1 \mapsto \lambda z.(u_1\ u_2))\}, \ldots\}),$ $((3), \{\{u_1 \mapsto \lambda z.\neg(u_3\ u_2)\}, \ldots\}), ((4), \{\{u_1 \mapsto \lambda z.\neg(a\ (b\ u_3)\ u_2)\}, \ldots\})\}$.*

- *the set $A_2$ can be computed in a similar way and results in $\{((3), \{\{u_3 \mapsto \lambda z.(u_3\ u_2)\}, \ldots\}), ((4), \{\{u_3 \mapsto \lambda z.(a\ (b\ u_3)\ u_2)\}\}), \ldots\}$.*

- *we notice that only the labels (3) and (4) occur in both sets $A_1$ and $A_2$ and that each of the substitutions from $A_2$ can be matched with two substitutions in $A_1$.*

- *both options are compatible, the first option means that each literals of clause (6) subsumes a different literal of clauses (3) and (4) while the second option means that they subsumes the same literal in these clauses.*

- *we conclude that they are both redundant.*

- *the resulted tree after their removal was computed in Ex. 14.*

**Claim 30.** *If $l \in \texttt{bsum}(T, c)$ and the clause $d$ is labeled by $l$, then $d$ is subsumed by $c$.*

**Example 31.** *The result of inserting clause (6) to the substitution tree from the previous example can be seen in Fig. 5.*

## 4 Conclusion and Future Work

In this work, we have presented an indexing data structure for higher-order clause subsumption based on substitution trees. We make use of an efficient higher-order pattern anti-unification algorithm for calculting meaningful generalizations of two arbitrary higher-order terms. The proposed indexing method is rather limited as is does not support subsumption testing modulo associativity and commutativity which is, of course, essential for general clause subsumption. However, even the limited approach admits effective subsumption queries in certain cases. Additionally, improvements for including such AC aspects are sketched.

While the index is not size-optimal in general, we believe that the approach performs quite good in practice, especially when combined with further, orthogonal indexing techniques that could be used as a pre-test. One suitable candidate is a higher-order variant of feature vector indexing [18].

The substitution tree index as described here is planned for implementation in the Leo-III prover [24]. We hope that due to Leo-III's agent-based architecture [20], independent agents can traverse the substitution tree index in parallel. The index is mainly devised for employment in resolution-based provers, but it seems possible to generalize the approach to non-clausal-based deduction procedures.

For further work we need to investigate means of suitably enhancing `msg` to handle AC properties and other subsumption properties. Also, the matching algorithm could be improved such that it returns minimal complete sets of substitutions which can then be used for the subsumption procedure. At the current state of the index data structure, the inserted substitutions are not normalized. This is essentially a shortcoming that originates from the way we relate the matching substitutions of different literals of the same clause to each other. This results, however, in a substitution tree that contains occurrences of substitutions which are equivalent up to free variables renaming. To overcome this shortcoming, we need to find a way of keeping the substitutions normalized while still being able to relate the matchers of different literals.

### Acknowledgements

Work of the first author was funded by the ERC Advanced Grant ProofCert. The second author has been supported by the DFG under grant BE 2501/11-1 (Leo-III). We thank the reviewers for the very valuable feedback they provided.

## References

[1] Hendrik Pieter Barendregt. *The Lambda Calculus – Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics.* North-Holland, 1984.

[2] Alexander Baumgartner, Temur Kutsia, Jordi Levy, and Mateu Villaret. A variant of higher-order anti-unification. In *Leibniz International Proceedings in Informatics (LIPPICS): 24th International Conference on Rewriting Techniques and Applications (RTA 2013): RTA 2013: June 24-26 2013: Eindhoven, The Netherlands, vol. 21, p. 113-127.* Dagstuhl Publishing, 2013.

[3] Christoph Benzmüller, Lawrence C. Paulson, Nik Sultana, and Frank Theiß. The Higher-Order Prover LEO-II. *Journal of Automated Reasoning*, 55(4):389–404, 2015.

[4] Alonzo Church. A formulation of the simple theory of types. *J. Symb. Log.*, 5(2):56–68, 1940.

[5] François Fages. Associative-commutative unification. *Journal of Symbolic Computation*, 3(3):257–275, 1987.

[6] Peter Graf. Substitution tree indexing. In *Rewriting Techniques and Applications*, pages 117–131. Springer, 1995.

[7] Peter Graf and Christoph Meyer. Advanced indexing operations on substitution trees. In *CADE*, volume 1104 of *Lecture Notes in Computer Science*, pages 553–567. Springer, 1996.

[8] Gérard P. Huet. A unification algorithm for typed lambda-calculus. *Theor. Comput. Sci.*, 1(1):27–57, 1975.

[9] A. Leitsch. *The Resolution Calculus.* EATCS Monographson Theoretical Computer Science. Springer, 1997.

[10] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Trans. Program. Lang. Syst.*, 4(2):258–282, April 1982.

[11] Dale Miller. Unification of simply typed lambda-terms as logic programming. In *8th International Logic Programming Conference*, pages 255–269. MIT Press, 1991.

[12] Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. Contextual modal type theory. *Under consideration for publication in the ACM Transactions on Computation Logic*, 2005.

[13] Frank Pfenning and Carsten Schürmann. System description: Twelfa meta-logical framework for deductive systems. In *Automated DeductionCADE-16*, pages 202–206. Springer, 1999.

[14] Brigitte Pientka. Higher-order term indexing using substitution trees. *ACM Transactions on Computational Logic (TOCL)*, 11(1):6, 2009.

[15] Brigitte Pientka and Frank Pfenning. Optimizing higher-order pattern unification. In *Automated Deduction–CADE-19*, pages 473–487. Springer, 2003.

[16] I. V. Ramakrishnan, R. C. Sekar, and Andrei Voronkov. Term indexing. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning (in 2 volumes)*, pages 1853–1964. Elsevier and MIT Press, 2001.

[17] John Alan Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965.

[18] Stephan Schulz. Simple and efficient clause subsumption with feature vector indexing. In *Automated Reasoning and Mathematics*, volume 7788 of *Lecture Notes in Computer Science*, pages 45–67. Springer, 2013.

[19] Wayne Snyder and Jean H. Gallier. Higher-order unification revisited: Complete sets of transformations. *J. Symb. Comput.*, 8(1/2):101–140, 1989.

[20] Alexander Steen, Max Wisniewski, and Christoph Benzmüller. Agent-Based HOL Reasoning. In *The 5th International Congress on Mathematical Software (ICMS 2016)*, volume 9725 of *LNCS*, Berlin, Germany, 2016. Springer. To appear in 2016.

[21] Colin Stirling. Decidability of higher-order matching. *Logical Methods in Computer Science*, 5(3), 2009.

[22] Frank Theiss and Christoph Benzmüller. Term indexing for the LEO-II prover. In *IWIL-6 workshop at LPAR 2006: The 6th International Workshop on the Implementation of Logics*, Pnom Penh, Cambodia, 2006.

[23] Andrei Voronkov. Implementing bottom-up procedures with code trees: a case study of forward subsumption. Technical report, 1995.

[24] Max Wisnieski, Alexander Steen, and Christoph Benzmüller. The Leo-III project. In Alexander Bolotov and Manfred Kerber, editors, *Joint Automated Reasoning Workshop and Deduktionstreffen*, page 38, 2014.