Higher-order substitution tree indexing

Brigitte Pientka*

Department of Computer Science
Carnegie Mellon University
bp@cs.cmu.edu

Abstract. We present a higher-order term indexing strategy based on substitution trees. The strategy is based in linear higher-order patterns where computationally expensive parts are delayed. Insertion of terms into the index is based on computing the most specific linear generalization of two linear higher-order patterns. Retrieving terms is based on matching two linear higher-order patterns. This indexing structure is implemented as part of the Twelf system to speed-up the execution of the tabled higher-logic programming interpreter. Experimental results show substantial performance improvements, between 100% and over 800%.

1 Introduction

Efficient term indexing techniques have resulted in dramatic speed improvements in all major first-order logic programming and theorem proving systems and have been crucial to their success. Over the last years, different indexing techniques have been proposed for first-order terms (see [19] for a survey). However, indexing techniques for higher-order terms are missing thereby severely hampering the performance of higher-order reasoning systems such as Twelf [13], λProlog [8] or Isabelle [11].

In this paper, we present a higher-order term indexing technique based on substitution trees. Substitution tree indexing [3] is highly successful in the first-order setting and allows the sharing of common sub-expressions via substitutions. Given the following terms,pred (h (g a)) (g b) a (term 1), pred (h (g b)) (g b) a (term 2))and the term (3) pred (h (g c)) (g b) c , we compute their most specific common generalization, which is pred (h (g *1)) (g b) *2 where *1 and *2 are placeholders. We then obtain the first term, by substituting a for the placeholders *1 and *2. The second term is recovered by substituting b for *1 and a for *2. Finally, the third term is obtained by substituting c for *1 and *2. The common structure of these three terms can be shared by storing substitutions in a tree (see Figure 1). To obtain the first term, we compose the substitutions in the left-most branch. Similarly, composing all the substitutions in the right-most branch yields the third term.

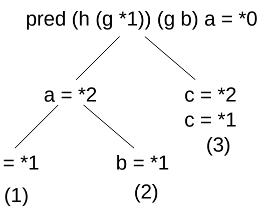


Fig. 1. Substitution tree

* This work has been partially supported by NSF Grant CCR-9988281 "*Logical and Meta-Logical Frameworks*" and by a *Siebel Scholarship*.

Extending this idea to the higher-order setting poses several challenges: First, building a substitution tree relies on computing the most specific generalization of two terms. However in the higher-order setting, the most specific generalization of two terms does not exist in general. Second, retrieving all terms, which unify or match, needs to be efficient – but higher-order unification is undecidable in general. Although decidable fragments, called higher-order patterns [7, 12], exist, algorithms based on them may not be efficient enough in practice [17]. Hence it is not obvious that they are a suitable basis for higher-order term indexing techniques.

In this paper, we present substitution tree indexing for higher-order terms based on linear higher-order patterns [17]. Linear higher-order patterns refine the notion of higher-order patterns further and factor out any computationally expensive parts. As shown in [17], many terms encountered fall into this fragment and linear higher-order pattern unification performs well in practice. In this paper, we give algorithms for computing the most specific generalization of two linear higher-order patterns, for inserting and retrieving terms from the index. This indexing structure is implemented as part of the Twelf system [13] to speed-up the execution of the tabled logic programming interpreter [15]. Experimental results show substantial performance improvements.

The paper is organized as follows: First, we give a theoretical framework for the dependently-typed lambda calculus. In Section 3, we present the general idea of higher-order substitution trees. In Section 4, we give algorithms for computing the most specific generalization of two terms and inserting terms into the index. Retrieval is discussed in Section 5. In Section 6, we present some experimental results comparing the tabled higher-order logic programming interpreter with and without indexing.

2 Dependently typed lambda calculus

In this section, we introduce a conservative extension of LF [5] where we have existential variables as first-class objects [17]. We distinguish between ordinary variables denoted by x and existential variables denoted by u. c and a are constants, which are declared in a signature. In the presentation, we suppress some routine details such as signatures. Although we use the dependently typed lambda-calculus, the presented indexing techniques can be easily adopted for the simply-typed case.

$$
\begin{array}{lrcl}
\text{Kinds} & K & ::= & \mathsf{type} \mid \Pi x{:}A.\,K \\
\text{Families} & A,B,C & ::= & a \mid A\,M \mid \Pi x{:}A_1.\,A_2 \\
\text{Objects} & M,N & ::= & c \mid x \mid u[\sigma] \mid \lambda x{:}A.\,M \mid M_1\,M_2 \\
\text{Substitutions} & \sigma,\tau & ::= & \cdot \mid \sigma, M/x \\
\text{Contexts} & \Gamma,\Psi & ::= & \cdot \mid \Gamma, x{:}A \\
\text{Modal Contexts} & \Delta & ::= & \cdot \mid \Delta, u{::}(\Psi{\vdash}A)
\end{array}
$$

Existential variable u are viewed as modal variables of type A in the context Ψ and are declared in a modal context Δ, while ordinary variables are declared in the context Γ. Note that the substitution σ is part of the syntax of existential variables. The principal judgments are listed below. As usual, we omit similar judgments on types and kinds and all judgments concerning definitional equality.

$\Delta; \Gamma \vdash M : A$	Object M has type A
$\Delta \vdash \theta : \Delta'$	Modal substitution θ matches Δ'
$\Delta; \Gamma \vdash \sigma : \Psi$	Substitution σ matches context Ψ
$\vdash \Delta \ \mathsf{mctx}$	Δ is a valid modal context
$\Delta \vdash \Psi \ \mathsf{ctx}$	Ψ is a valid context

We will tacitly rename bound variables, and maintain that contexts and substitutions declare no variable more than once. Note that substitutions σ are defined only on ordinary variables x and not existential variables u. We write id_Γ for the identity substitution $(x_1/x_1, \ldots, x_n/x_n)$ for a context $\Gamma = (\cdot, x_1{:}A_1, \ldots, x_n{:}A_n)$. We will use π for a substitution which may permute the variables, i.e $\pi = (x_{\Phi(1)}/x_1, \ldots, x_{\Phi(n)}/x_n)$ where Φ is a total permutation defined on the elements from a context $\Gamma = (\cdot, x_1{:}A_1, \ldots, x_n{:}A_n)$. We only consider well-typed substitutions, so π must respect possible dependencies in its domain. We also streamline the calculus slightly by always substituting simultaneously for all ordinary variables. This is not essential, but saves some tedium in relating simultaneous and iterated substitution. Moreover, it is also closer to the actual implementation where we use de Bruijn indices and postpone explicit substitutions. Our convention is that substitutions as defined operations on expressions are written in prefix notation $[\sigma]P$ for an object, family, kind, or substitution P. These operations are capture-avoiding as usual. Moreover, we always assume that all free variables in P are declared in σ. Ordinary substitutions are defined in a standard manner, and we omit the details for the sake of brevity (for a more detailed account see [17]). Substitutions that are part of the syntax are written in postfix notation, $u[\sigma]$. Note that such explicit substitutions occur only for existential variables u defined in Δ. Substitutions θ are defined on existential variables. We write θ for a simultaneous substitution $[\![M_1/u_1, \ldots, M_n/u_n]\!]$ where $u_1, \ldots, u_n$ are distinct existential variables. The new operation of substitution is compositional, but two interesting situations arise: when a variable u is encountered, and when we substitute into a λ-abstraction (or a dependent type Π respectively).

Substitutions

$$
\begin{aligned}
[\![\theta]\!](\cdot) &= \cdot \\
[\![\theta]\!](\sigma, N/y) &= ([\![\theta]\!]\sigma, [\![\theta]\!]N/y)
\end{aligned}
$$

Objects

$$
\begin{aligned}
[\![\theta]\!]c &= c \\
[\![\theta]\!]x &= x \\
[\![\theta_1, M/u, \theta_2]\!](u[\sigma]) &= [[\![\theta_1, M/u, \theta_2]\!]\sigma]M \\
[\![\theta]\!](N_1\, N_2) &= ([\![\theta]\!]N_1)\, ([\![\theta]\!]N_2) \\
[\![\theta]\!](\lambda y{:}A.\, N) &= \lambda y{:}[\![\theta]\!]A.\, [\![\theta]\!]N
\end{aligned}
$$

We remark that the rule for substitution into a λ-abstraction does not require a side condition. This is because the object M is defined in a different context, which is accounted for by the explicit substitution stored at occurrences of u.

Finally, consider the case of substituting into a closure, which is the critical case of this definition.

$$[\![\theta_1, M/u, \theta_2]\!](u[\sigma]) = [[\![\theta_1, M/u, \theta_2]\!]\sigma]M$$

This is clearly well-founded, because σ is a subexpression (so $[\![\theta_1, M/u, \theta_2]\!]\sigma$ will terminate) and application of an ordinary substitution is defined without reference to the new form of substitution. The modal substitutions are compositional and the usual substitution properties hold (see [16] for a detailed discussion). We write id_Δ for the identity modal substitution $(u_1[\mathsf{id}_{\Psi_1}]/u_1, \ldots, u_n[\mathsf{id}_{\Psi_n}]/u_n)$ for a modal context $\Delta = (u_1{::}\Psi_1{\vdash}A_1, \ldots, u_n{::}\Psi_n{\vdash}A_n)$. The typing rules are given below.

Modal substitutions

$$\frac{}{\Delta \vdash (\cdot) : (\cdot)} \qquad \frac{\Delta \vdash \theta : \Delta' \quad \Delta; [\![\theta]\!]\Psi \vdash M : [\![\theta]\!]A}{\Delta \vdash (\theta, M/u) : (\Delta', u{::}\Psi{\vdash}A)}$$

Ordinary substitutions

$$\frac{}{\Delta; \Gamma \vdash (\cdot) : (\cdot)} \qquad \frac{\Delta; \Gamma \vdash \sigma : \Psi \quad \Delta; \Gamma \vdash M : [\sigma]A}{\Delta; \Gamma \vdash (\sigma, M/x) : (\Psi, x{:}A)}$$

Objects

$$\frac{}{\Delta; \Gamma, x{:}A, \Gamma' \vdash x : A} \qquad \frac{\Delta, u{::}(\Psi{\vdash}A), \Delta'; \Gamma \vdash \sigma : \Psi}{\Delta, u{::}(\Psi{\vdash}A), \Delta'; \Gamma \vdash u[\sigma] : [\sigma]A}$$

$$\frac{\Delta; \Gamma, x{:}A_1 \vdash M : A_2}{\Delta; \Gamma \vdash \lambda x{:}A_1..\, M : \Pi x{:}A_1.\, A_2} \qquad \frac{\Delta; \Gamma \vdash M_1 : \Pi x{:}A_2.\, A_1 \quad \Delta; \Gamma \vdash M_2 : A_2}{\Delta; \Gamma \vdash M_1\, M_2 : [\mathsf{id}_\Gamma, M_2/x]A_1}$$

Modal Context

$$\frac{}{\vdash (\cdot)\ \mathsf{mctx}} \qquad \frac{\vdash \Delta\ \mathsf{mctx} \quad \Delta \vdash \Psi\ \ \mathsf{ctx} \quad \Delta; \Psi \vdash A : \mathsf{type}}{\vdash (\Delta, u{::}(\Psi{\vdash}A))\ \mathsf{mctx}}$$

Ordinary Context

$$\frac{}{\Delta \vdash (\cdot)\ \mathsf{ctx}} \qquad \frac{\Delta \vdash \Psi\ \mathsf{ctx} \quad \Delta; \Psi \vdash A : \mathsf{type}}{\Delta \vdash (\Psi, x{:}A)\ \mathsf{ctx}}$$

We only consider objects in $\beta\eta$-normal form and require that all terms which are not functions must be of atomic type. In particular, existential variables must also be of atomic type. This is accomplished by a technique called *lowering*. Lowering replaces a variable $u{::}(\Psi{\vdash}\Pi x{:}A_1.\, A_2)$ by a new variable $u'{::}(\Psi, x{:}A_1{\vdash}A_2)$. This process is repeated until all existential variables have a type of the form $\Psi \vdash b\, N_1 \ldots N_k$. For a more detailed account we refer the reader to [17, 9, 16]. Here, we just present the refined grammar for objects:

$$\text{Normal Objects } U ::= \lambda x.\, U \mid u[\sigma] \mid H\ U_1\ \ldots U_n$$

The head H is either a constant c or a variable x. Note that type labels on λ-abstraction may be omitted for canonical forms. To illustrate the notation, we give a sample signature together with a set of terms we want to index. We define a type family exp for expressions and a predicate pred which takes in three arguments, where each argument is an expression. In addition, we give constants a, b, c and constructors f, g, h for building expressions of this small language.

```
exp: type.          pred: exp -> exp -> exp -> type.
a: exp.             h: exp -> exp.
b: exp.             g: exp -> exp.
c: exp.             f: (exp -> exp) -> exp.
```

Next, we give four examples of the predicate pred:

$$\begin{array}{rl} \cdot;\cdot \vdash \mathsf{pred}\ (\mathsf{h}\ (\mathsf{g}\ \mathsf{b}))\ (\mathsf{g}\ \mathsf{b})\ \mathsf{a} & (1) \\ \cdot;\cdot \vdash \mathsf{pred}\ (\mathsf{h}\ (\mathsf{g}\ \mathsf{b}))\ (\mathsf{g}\ \mathsf{b})\ \mathsf{b} & (2) \\ v{::}\cdot\vdash\mathsf{exp};\cdot \vdash \mathsf{pred}\ (\mathsf{h}\ (\mathsf{h}\ \mathsf{b}))\ (\mathsf{g}\ \mathsf{b})\ (\mathsf{f}\ \lambda x.v[\cdot]) & (3) \\ u{::}x{:}\mathsf{exp}\vdash\mathsf{exp};\cdot \vdash \mathsf{pred}\ (\mathsf{h}\ (\mathsf{h}\ \mathsf{c}))\ (\mathsf{g}\ \mathsf{b})\ (\mathsf{f}\ \lambda x.u[x/x]) & (4) \end{array}$$

The first two predicates do not refer to any existential variables, while predicate (3) and (4) contain existential variables v and u respectively which are bound in the modal context. Note that all terms are well-typed in the context $\Delta = u{::}x{:}\mathsf{exp}\vdash\mathsf{exp}, v{::}\cdot\vdash\mathsf{exp}$ by weakening. Using these examples, we will briefly highlight some of the subtle issues concerning the interplay of bound and existential variables. The third predicate (3) refers to the existential variable v. Note that v is associated with the empty substitution, although it occurs in the context of a bound variable x. This means that any instantiation we find for v is not allowed to depend on the bound variable x. In contrast, predicate (4) refers to an existential variable u, which is associated with the identity substitution. This means that any instantiation we find for u may depend on the bound variable x. We come back to this point in Section 3.1.

3 Higher-order substitution trees

We start by giving the general idea of higher-order substitution tree. To build a higher-order substitution tree, we proceed in two steps: First, we standardize the terms and convert terms into linear higher-order patterns. Second, we represent terms as a sequence of substitutions, which are stored in a tree.

3.1 Standardization: Linear higher-order pattern

To get the maximum structure sharing across different indexed terms it is important to use variables in a consistent manner. In first-order term indexing we therefore standardize the term before inserting a term into an index. In addition, first-order term indexing strategies often employ linearization. When converting the term into standard form every occurrence of an existential variable is represented as a distinct standardized existential variable. Together with the linear term, we then also store variable definitions, which establish the equality between these two variables. One of the reasons for using linearization is efficiency. Non-linear terms may have multiple occurrences of the same existential variable and therefore requiring to check whether different substitutions for the same variable are consistent. Since such consistency checks can be potentially expensive and may lead to performance degradation, most first-order indexing techniques rely on a post-processing step to carry out the consistency checks.

To design a higher-order indexing technique, we will extend this notion of linearization and standardization. We will require that terms are linear higher-order patterns. Higher-order patterns [7, 12] are terms where every existential variable must be applied to *some* distinct bound variables. Linear higher-order patterns [17] impose some further restrictions on the structure of terms: First, all existential variables must occur only once. This allows us to delay any expensive consistency checks. Second, all existential variables must be applied to *all* distinct bound variables. This eliminates any computationally expensive checks involving bound variables. This observation to restrict higher-order patterns even further to patterns where existential variables must be applied to all bound variables has also been made by Hanus and Prehofer [4] in the context of higher-order functional logic programming. While Hanus and Prehofer syntactically disallow terms which are not fully applied, we translate any term into a linear higher-order pattern together with some variable definitions. Any consistency checks are delayed after linear higher-order pattern unification has succeeded. As was shown in [17], performance of unification is improved substantially by this technique.

To illustrate, let us consider the previous term $\mathsf{pred}\ (\mathsf{h}\ (\mathsf{g}\ \mathsf{b}))\ (\mathsf{g}\ \mathsf{b})\ (\mathsf{f}\ \lambda x.v[\cdot])$. The existential variable $v[\cdot]$ occurs only once, but is not applied to all distinct bound variables, since v is not allowed to depend on the bound variable x. Therefore $v[\cdot]$ is a higher-order pattern, but it is not linear. We can enforce that every existential variable occurs only once and is applied to *all* bound variables, by translating it into a linear higher-order pattern:

$$\mathsf{pred}\ (\mathsf{h}\ (\mathsf{g}\ \mathsf{b}))\ (\mathsf{g}\ \mathsf{b})\ (\mathsf{f}\ \lambda x.u[x/x])$$

together with a variable definition,

$$\forall x{:}\mathsf{exp}.u[x/x] \stackrel{D}{=} v[.]$$

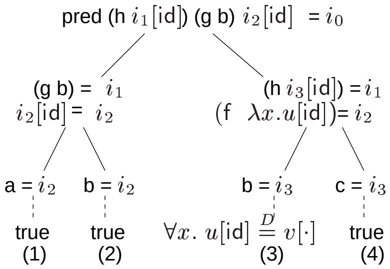
where u is a new existential variable which is applied to the bound variable x. Similarly to first-order standardization and linearization, linearization in the higher-order case remains quite straightforward and can be done by traversing the term once. The main difference is that the postponed variable definitions may be more complex.

$$\text{Variable Definitions}\quad D ::= \mathsf{true} \mid u[\mathsf{id}_\Psi] \stackrel{D}{=} U \mid D_1 \wedge\ D_2 \mid \forall x{:}A.D$$

3.2 Higher-order substitution trees: an example

In this section, we show the substitution tree for the set of terms given earlier. In each node, we store a set of substitutions, which we write here as $U = i$ where

U is a linear higher-order pattern and i is an internal existential variable and i will always be applied to all bound variables it may depend on. Therefore it will always be associated with the identity substitution id.



Note that variable definitions are stored at the leafs. By composing the substitutions in the right-most branch for example we get

$$[\![\mathsf{c}/i_3]\!][\![\mathsf{f}\ \lambda x.\ u[\mathsf{id}]/i_2,\ (\mathsf{h}\ i_3[\mathsf{id}])/i_1]\!][\![\mathsf{pred}\ (\mathsf{h}\ i_1[\mathsf{id}])\ (\mathsf{g}\ \mathsf{b})\ i_2[\mathsf{id}]/i_0]\!]i_0[\mathsf{id}]$$

which represents the term (4) $\mathsf{pred}\ (\mathsf{h}\ (\mathsf{h}\ \mathsf{c}))\ (\mathsf{g}\ \mathsf{b})\ (\mathsf{f}\ \lambda x.\ u[\mathsf{id}])$. Note that in the implementation we can omit the identity substitution $i_2[\mathsf{id}]/i_2$ and maintain as an invariant that the substitutions in each node can be extended appropriately. This will simplify the theoretical treatment slightly.

In the following development we will distinguish between internal existential variables i which are defined in the modal context Σ and "global" existential variables u and v which are defined in the modal context Δ. A higher-order substitution tree is an ordered tree and is defined as follows:

1. A tree is a leaf node with substitution ρ such that $\Delta \vdash (\mathsf{id}_\Delta, \rho) : (\Delta, \Sigma)$.
2. A tree is a node with substitution ρ such that $(\Delta, \Sigma) \vdash (\mathsf{id}_\Delta, \rho) : (\Delta, \Sigma')$ and children nodes $N_1, \ldots, N_n$ where each child Node N_i has a substitution ρ_i such that $(\Delta, \Sigma_i) \vdash (\mathsf{id}_\Delta, \rho_i) : (\Delta, \Sigma)$.

For every path from the top node ρ_0 where $(\Delta, \Sigma_1) \vdash (\mathsf{id}_\Delta, \rho_0) : (\Delta, \Sigma_0)$ to the leaf node ρ_n, we have $\Delta \vdash [\![id_\Delta, \rho_n]\!]([\![\mathsf{id}_\Delta, \rho_{n-1}]\!] \ldots (\mathsf{id}_\Delta, \rho_0)) : (\Delta, \Sigma_0)$. In other words, there are no internal existential variables left after we compose all the substitutions ρ_n up to ρ_0. Note that there are no typing dependencies among the variables in Σ and they can be arbitrarily re-ordered. Moreover, it is convenient to consider the extended modal substitution $(\mathsf{id}_\Delta, \rho_i)$ in the theory although in an implementation, we do not need to carry around explicitly the modal substitution id_Δ, but can always assume that any substitution ρ_i can be extended appropriately.

4 Insertion into index

Insertion of a term U into the index is viewed as insertion of the substitution U/i_0. Assuming that U has type A in a modal context Δ and a bound variable context Γ, U/i_0 is a modal substitution such that $\Delta \vdash U/i_0 : i{::}(\Gamma{\vdash}A)$. We will call the modal substitution which is going to be inserted into the index ρ.

The insertion process works by following down a path in the tree that is *compatible* with the modal substitution ρ. To formally define insertion, we show how to compute the most specific linear generalization (msg) of two modal substitutions and describe the most specific linear generalization of two terms. We start by giving the judgments for computing the msg of two modal substitutions and the msg of two linear higher-order patterns.

$$(\Delta, \Sigma_1) \vdash \rho_1 \sqcup \rho_2 : \Sigma_2 \Longrightarrow \rho/(\Sigma, \theta_1, \theta_2) \qquad \rho \text{ is the msg of } \rho_1 \text{ and } \rho_2$$
$$(\Delta, \Sigma); \Gamma \vdash L_1 \sqcup L_2 : A \Longrightarrow L/(\Sigma',\ \theta_1, \theta_2)\ L \text{ is the msg of } L_1 \text{ and } L_2$$

If $(\Delta, \Sigma_1) \vdash (\mathsf{id}_\Delta, \rho_1) : (\Delta, \Sigma_2)$ and $(\Delta, \Sigma_1) \vdash (\mathsf{id}_\Delta, \rho_2) : (\Delta, \Sigma_2)$ then ρ is the msg of ρ_1 and ρ_2. Moreover, ρ is a modal substitution such that $[\![\mathsf{id}_\Delta, \theta_1]\!]\rho$ is syntactically equal to ρ_1 and $[\![\mathsf{id}_\Delta, \theta_2]\!]\rho$ is syntactically equal to ρ_2. We think of ρ_1 as the modal substitution which is already in the index, while the modal substitution ρ_2 is to be inserted. The result of the msg are the modal substitution θ_1 and θ_2, where $\Delta, \Sigma_1 \vdash \theta_1 : \Sigma$ and $\Delta, \Sigma_1 \vdash \theta_2 : \Sigma$. In other words, θ_1 (resp. θ_2) only replaces internal existential variables in Σ.

If the terms L_1 and L_2 have type A in modal context (Δ, Σ) and bound variable context Γ, then L is the most specific generalization of L_1 and L_2 such that $[\![\mathsf{id}_\Delta, \theta_1]\!]L$ is syntactically equal to L_1 and $[\![\mathsf{id}_\Delta, \theta_2]\!]L$ is syntactically equal to L_2. Moreover, θ_1 and θ_2 are modal substitutions which map existential variables from Σ' to the modal context (Δ, Σ). We think of L_1 as an object which is already in the index and L_2 is the object to be inserted. First, we give the rules for computing the most specific generalization of two modal substitutions.

$$\frac{}{(\Delta, \Sigma) \vdash \cdot : \cdot \Longrightarrow \cdot/(\cdot, \cdot, \cdot)}$$

$$\frac{\begin{array}{c}(\Delta, \Sigma_1) \vdash \rho_1 \sqcup \rho_2 : \Sigma_2 \Longrightarrow \rho/(\Sigma, \theta_1, \theta_2) \\ (\Delta, \Sigma_1); \Gamma \vdash L_1 \sqcup L_2 : \tau \Longrightarrow L/(\Sigma', \theta_1', \theta_2')\end{array}}{(\Delta, \Sigma_1) \vdash (\rho_1, L_1/i) \sqcup (\rho_2, L_2/i) : (\Sigma_2, i{::}(\Gamma{\vdash}\tau)) \Longrightarrow (\rho, L/i)/((\Sigma, \Sigma'), (\theta_1, \theta_1'), (\theta_2, \theta_2'))}$$

Note, that we are allowed to just combine the modal substitutions θ_1 (θ_2 resp.) and θ_1' (θ_2' resp.) since we require that they refer to distinct existential variables and all the existential variables occur uniquely. Computing the most specific generalization of two modal substitutions relies on finding the most specific generalization of two objects.

$$\frac{(\Delta, \Sigma); \Gamma, x{:}A_1 \vdash L_1 \sqcup L_2 : A_2 \Longrightarrow L/(\Sigma', \theta_1, \theta_2)}{(\Delta, \Sigma); \Gamma \vdash \lambda x.L_1 \sqcup \lambda x.L_2 : A_1 \to A_2 \Longrightarrow \lambda x.L/(\Sigma',\ \theta_1, \theta_2)}$$

$$\frac{u{::}(\Psi{\vdash}a) \in \Delta}{(\Delta, \Sigma); \Gamma \vdash u[\pi] \sqcup u[\pi] : a \Longrightarrow u[\pi]/(\cdot, \cdot, \cdot)}\ (*)$$

$$\frac{u{::}\Psi{\vdash}a \in (\Delta, \Sigma) \quad i \text{ must be new}}{(\Delta, \Sigma); \Gamma \vdash u[\pi] \sqcup L : a \Longrightarrow i[\mathsf{id}_\Gamma]/(i{::}\Gamma{\vdash}a,\ u[\pi]/i,\ L/i)}\ (1a)$$

$$\frac{i{::}\Gamma{\vdash}a \in (\Delta,\Sigma)}{(\Delta,\Sigma);\Gamma \vdash i[\mathsf{id}_\Gamma] \sqcup L : a \Longrightarrow i[\mathsf{id}_\Gamma]/(i{::}\Gamma{\vdash}a,\ i[\mathsf{id}_\Gamma]/i,\ L/i)}\ (1b)$$

$$\frac{u{::}(\Psi{\vdash}a) \in \Delta}{(\Delta,\Sigma);\Gamma \vdash L \sqcup u[\pi] : a \Longrightarrow i[\mathsf{id}_\Gamma]/(i{::}\Gamma{\vdash}a,\ L/i,\ u[\pi]/i)}\ (2)$$

$$\frac{\begin{array}{c}(\Delta,\Sigma);\Gamma \vdash L_1' \sqcup L_1'' : A_1 \Longrightarrow L_1/(\Sigma_1,\theta_1',\theta_1'')\\ \vdots \\ (\Delta,\Sigma);\Gamma \vdash L_n' \sqcup L_n'' : A_n \Longrightarrow L_n/(\Sigma_n,\theta_n',\theta_n'')\\ L = H\ L_1 \ldots L_n \quad \Sigma = (\Sigma_1,\ldots \Sigma_n) \quad \theta_1 = (\theta_1',\ldots,\theta_n') \quad \theta_2 = (\theta_1'',\ldots,\theta_n'')\end{array}}{(\Delta,\Sigma);\Gamma \vdash H\ L_1'' \ldots L_n'' \sqcup H\ L_1' \ldots L_n' : a \Longrightarrow L/(\Sigma,\theta_1,\theta_2)}\ (3a)$$

$$\frac{H_1 \neq H_2 \quad i \text{ must be new} \quad \theta_1 = (H_1\ L_1' \ldots L_n'/i) \quad \theta_2 = (H_2\ L_1'' \ldots L_n''/i)}{(\Delta,\Sigma);\Gamma \vdash H_1\ L_1' \ldots L_n' \sqcup H_2\ L_1'' \ldots L_n'' : a \Longrightarrow i[\mathsf{id}_\Gamma]/((i{::}\Gamma{\vdash}a),\theta_1,\theta_2)}\ (3b)$$

Note in the rule for lambda, we do not need to worry about capture, since existential variables and bound variables are defined in different context. Rule (*) treats the case where both terms are existential variables. Note that we require that both existential variables must be the same and their associated substitutions must also be equal. In rule (1) and (2), we just create the substitution $u[\pi]/i$. In general, we would need to create $[\mathsf{id}_\Gamma]^{-1}(u[\pi])$, but since we know that π is a permutation substitution, we know that $[\mathsf{id}_\Gamma]^{-1}(\pi)$ exists. In addition the inverse substitution of the identity is the identity. Note that we distinguish between the internal existential variables i and the "global" existential variables u in the rules (1a) and (1b). The key distinction is that we pick a new internal existential variable i in rule (1a) while we re-use the internal existential variable i in rule (1b). This is important for maintaining the invariant that any child of $(\Delta,\Sigma_2) \vdash (\mathsf{id}_\Delta,\rho) : (\Delta,\Sigma_1)$ has the form $(\Delta,\Sigma_3) \vdash (\mathsf{id}_\Delta,\rho') : (\Delta,\Sigma_2)$.

In rule (3a) and (3b) we distinguish two cases. If H_1 and H_2 are not equal, then we generate a new internal existential variable $i[\mathsf{id}_\Gamma]$ together with the substitutions $H_1\ L_1' \ldots L_n'/i$ and $H_2\ L_1'' \ldots L_n''/i$. Otherwise, we compute the most specific generalization for each L_i' and L_i'' and combine all the substitutions together. This is allowed since we require that all existential variables occur uniquely, and hence there are no dependencies among Σ_i.

To insert a new substitution ρ_2 into the substitution tree N where node N has the substitution $\Delta,\Sigma \vdash \rho : \Sigma'$, we proceed in two steps. First, we inspect all the children N_i of a parent node N, where N_i contains a modal substitution $\Delta,\Sigma_i{\vdash}\rho_i : \Sigma$ and check if ρ_i is compatible with ρ_2. This compatibility check has three possible result.

1. $(\Delta,\Sigma_i) \vdash \rho_i \sqcup \rho_2 : \Sigma \Longrightarrow \mathsf{id}_\Sigma/(\Sigma,\rho_i,\rho_2)$
   This means ρ_i and ρ_2 are not compatible. As a result we create a new leaf with substitution ρ_2 and add it to the children of N.
2. $(\Delta,\Sigma_i) \vdash \rho_i \sqcup \rho_2 : \Sigma \Longrightarrow \rho_1/(\Sigma_i,\mathsf{id}_{\Sigma_i},\theta_2)$
   In this case $[\![\mathsf{id}_\Delta,\theta_2]\!]\rho_2$ is syntactically equivalent to ρ_i. This means ρ_2 is an

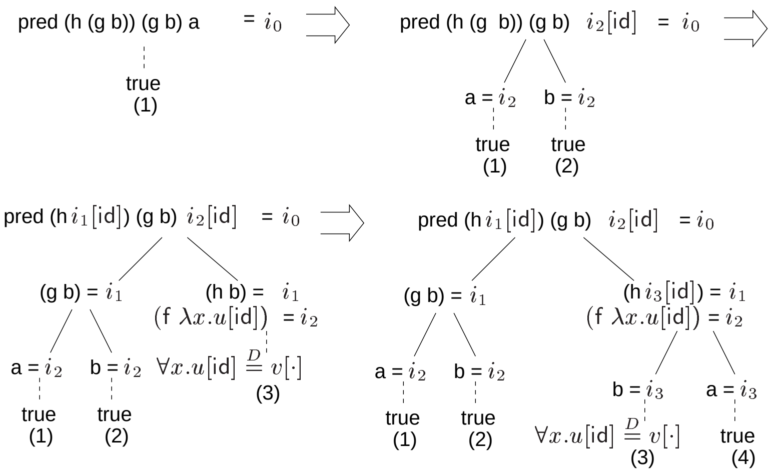
instance of ρ_i and we will continue to insert θ_2 into the children C_i of the node N_i.

3. $(\Delta, \Sigma_i) \vdash \rho_i \sqcup \rho_2 : \Sigma \Longrightarrow \rho' / (\Sigma'', \theta_1, \theta_2)$
ρ_i and ρ_2 are compatible and ρ' is more general than ρ_i and ρ_2. Then we replace node N_i with a new node with the substitution ρ' and two children, where the first child contains substitution θ_1 and has children C_i and the second child has the substitution θ_2 with no children.

We illustrate this process by an example. Assume we have the following two modal substitutions:

$$\rho_1 = [\![(g\ a)/i_1,\ f\ (\lambda x.u[\mathsf{id}])/i_2,\ (h\ c)/i_3]\!]$$
$$\rho_2 = [\![(g\ b)/i_1,\ c/i_2,\ (h\ a)/i_3]\!]$$

Then, the most specific linear common generalization of the substitution is computed as follows. We compute the most specific linear generalization for each element. Hence the resulting most specific common substitution ρ of ρ_1 and ρ_2 is $[\![(g\ \ i_4[\mathsf{id}])/i_1,\ \ i_2[\mathsf{id}]/i_2,\ \ (h\ \ i_5[\mathsf{id}])/i_3]\!]$. In addition, ρ_1 will be split into the most specific common substitution ρ and the substitution $\theta_1 = [\![a/i_4,\ f\ (\lambda x.u[\mathsf{id}])/i_2,\ c/i_5]\!]$, s.t. $[\![\theta_1]\!]\rho = \rho_1$. Similarly ρ_2 will be split into ρ and the substitution $\theta_2 = [\![b/i_4,\ c/i_2,\ a/i_5]\!]$, s.t. $[\![\theta_2]\!]\rho = \rho_2$. Next, we illustrates the process of inserting the terms (1) to (4) from page 5 into a higher-order substitution tree. As mentioned earlier in the implementation we do not explicitly carry around the identity substitution $i_2[\mathsf{id}]/i_2$ and we will omit them in this figure.



5 Retrieval

In general, we can retrieve all terms from the index which satisfy some property (e.g unifiability, instance, variant). One advantage of substitution trees is that all these retrieval operations can be implemented with only small changes. We adapt here the assignment for linear higher-order patterns described in [17] to check whether the term L_2 is an instance of the term L_1. The principal judgment for normal objects is defined as follows:

$$(\Delta, \Sigma); \Gamma \vdash L_1 \doteq L_2 \;/ (\theta, \rho) \; L_2 \text{ is an instance of } L_1$$

Again we assume that L_1 and L_2 must be well-typed in the modal context Δ, Σ and the bound variable context Γ. Let Δ_1 be the existential variables in L_1. Then θ is a substitution for the existential variables in Δ_1 and ρ is the substitution for the internal existential variables in Σ then $[\![\theta, \rho]\!]L_1$ is syntactically equal to L_2.

$$\frac{(\Delta, \Sigma); \Gamma, x{:}A \vdash L_1 \doteq L_2 \;/\; (\theta, \rho)}{(\Delta, \Sigma); \Gamma \vdash \lambda x{:}A.L_1 \doteq \lambda x{:}A.L_2 \;/\; (\theta, \rho)}\; lam$$

$$\frac{}{(\Delta, \Sigma); \Gamma \vdash i[\mathsf{id}_\Gamma] \doteq L \;/\; (\cdot, (L/i))}\; exL1 \qquad \frac{u{::}\Psi{\vdash}a \in \Delta}{(\Delta, \Sigma); \Gamma \vdash u[\pi] \doteq L \;/\; (([\pi]^{-1}\, L/u), \cdot)}\; exL2$$

$$\frac{\begin{array}{c}(\Delta, \Sigma); \Gamma \vdash L'_1 \doteq L''_1 \;/\; (\theta_1, \rho_1) \\ \vdots \\ (\Delta, \Sigma); \Gamma \vdash L'_n \doteq L''_n \;/\; (\theta_n, \rho_n)\end{array}}{(\Delta, \Sigma); \Gamma \vdash H L'_1 \ldots L'_n \doteq H L''_1 \ldots L''_n \;/\; ((\theta_1, \ldots, \theta_n), (\rho_1 \ldots \rho_n))}\; app$$

Note that we need not worry about capture in the rule for lambda expressions since existential variables and bound variables are defined in different contexts. In the rule *app*, we are allowed to union the two substitutions θ_1 and θ_2, as the linearity requirement ensures that the domains of both substitutions are disjoint. Note that the case for matching an existential variable $u[\pi]$ with another term L is simpler and more efficient than in the general higher-order pattern case. In particular, it does not require a traversal of L (see rules *exL1* and *exL2*). Since the inverse of the substitution π can be computed directly and will be total, we know $[\pi]^{-1}\, L$ exists and can simply generate a substitution $[\pi]^{-1}\, L/u$. The algorithm can be easily specialized to retrieve variances by requiring in the *exL2* rule that L must be $u[\pi]$. To check unifiability we need to add the dual rule to *exL2* where we unify L with an existential variable $u[\pi]$. The only complication is that L may contain internal existential variables which are defined later on the path in the substitution tree.

This instance algorithm for terms can be straightforwardly extended to instances of substitutions. Let $\rho' = [\![U_1/i_k, \ldots, U_n/i_n]\!]$ be associated with node N' and $\rho'' = [\![V_1/i_k, \ldots, V_n/i_n]\!]$, then we match each U_k against V_k and the final result is $\theta = [\![\theta_1, \ldots, \theta_n]\!]$ and $\rho = [\![\rho_1, \ldots, \rho_n]\!]$. In the next step, we continue to retrieve all children C' of node N' such that ρ is an instance of C'.

6 Experimental Results

In this section, we discuss examples from three different applications which use the tabled logic programming engine in Twelf [14, 15]. Here we focus on an evaluation of the indexing technique. All experiments are done on a machine with the following specifications: 1.60GHz Intel Pentium Processor, 256 MB RAM. We are using SML of New Jersey 110.0.3 under Linux Red Hat 7.1. Times are measured in seconds. All the examples use variant checking as a retrieval mechanism. Although we have implemented subsumption checking, we did not observe substantial performance improvements using subsumption. A similar observation has been made for tabling in the first-order logic programming engine XSB [18]. Potentially subsumption checking becomes more important in theorem proving, as the experience in the first-order setting shows.

6.1 Parsing of first-order formulae

Parsing and recognition algorithms for grammars are excellent examples for tabled evaluation, since we often want to mix right and left recursion (see also [20]). In this example, we adapted ideas from Warren [20] to implement a parser for first-order formulas using higher-order abstract syntax.

#tok	noindex	index	speed-up
20	0.13	0.07	85%
58	2.61	1.25	108%
117	10.44	5.12	103%
178	32.20	13.56	137%
235	75.57	26.08	190%

The first column denotes the number of tokens which are parsed. This example illustrates that indexing can lead to speed-ups to over 190%. In fact, the more tokens need to be parsed and the longer the tabled logic programming engine runs, the larger the benefits of indexing. The table grows up to over 4000 elements in this example. This indicates that indexing prevents to some extent program degradation due to large tables and longer run-times.

6.2 Refinement type-checker

In this section, we discuss experiments with a bi-directional type-checking algorithm for a small functional language with intersection types which has been developed by Davies and Pfenning [1]. We use an implementation of the bi-directional type-checker in *Elf* by Pfenning. The type-checker is executable with the original logic programming interpreter, which performs a depth-first search. However, redundant computation may severely hamper its performance as there are several derivations for proving that a program has a specified type.

We give several examples which are grouped in three categories. In the first category, we are interested in finding the first answer to a type checking problem and once we have found the answer execution stops. The second category contains example programs which are not well-typed and the implemented type-checker rejects these programs as not well-typed. The third category are examples where we are interested in finding all answer to the type-checking problem.

First answer

example	noindex	index	speed-up
sub1	3.19	0.46	593%
sub2	4.22	0.55	663%
sub3	5.87	0.63	832%
mult	7.78	0.89	774%
square1	9.08	0.99	817%
square2	9.02	0.98	820%

Not provable

example	noindex	index	speed-up
multNP1	2.38	0.38	526%
multNP2	2.66	0.51	422%
plusNP1	1.02	0.24	325%
plusNP2	6.48	0.85	662%
squareNP1	9.29	1.09	752%
squareNP2	9.26	1.18	685%

All answers

example	noindex	index	speed-up
sub1	6.88	0.71	869%
sub2	3.72	0.48	675%
sub3	4.99	0.59	746%
mult	9.06	0.98	824%
square1	10.37	1.11	834%
square2	10.30	1.08	854%

As the results demonstrate indexing leads to substantial improvements by over 800%. Table sizes are not as large as in the parsing example, however we accessed the table often.

6.3 Evaluating Mini-ML expression via reduction

In the third experiment we use an implementation which evaluates expressions of a small functional language via reduction. The reduction rules are highly non-deterministic containing reflexivity and transitivity rules.

example	noindex	index	speed-up
mult1	10.86	6.26	73%
mult2	39.13	18.31	114%
addminus1	54.31	14.42	277%
addminus2	57.34	15.66	266%
addminus3	55.23	25.45	117%
addminus4	144.73	56.63	155%
minusadd1	1339.16	462.83	189%

As the results demonstrate, performance is improved by up to 277%. Table size was around 500 entries in the table. The limiting factor in this example is not necessarily the table size but the large number of suspended goals which is over 6000. This may be the reason why the speed-up is not as large as in the refinement type-checking example.

7 Related work and conclusion

We have presented a term indexing technique, called higher-order substitution trees for dependently-typed terms which may contain λ-abstraction. This indexing technique is based on linear higher-order patterns. We only know of two other attempts to design and implement a higher-order term indexing technique. L. Klein [6] developed in his master's thesis a higher-order term indexing technique for simply typed terms where algorithms are based on a fragment of Huet's higher-order unification algorithm, the simplification rules. Since the most specific generalization of two higher-order terms does not exist in general, he suggests to maximally decompose a term into its atomic subterms. This approach result in larger substitution trees and stores redundant substitutions. In addition, he does not use explicit substitutions leading to further redundancy in the representation of terms. As no linearity criteria is exploited, the consistency checks are necessary.

Necula and Rahul briefly discuss the use of automata driven indexing for higher-order terms in [10]. Their approach is restricted to a fragment of LF and essentially ignores all higher-order features when maintaining the index, and return an imperfect set of candidates where full higher-order unification on the original term is used to filter out the ones which are in fact unifiable in a post-processing step. Their approach is also based on Huet's unification algorithm, which is highly nondeterministic. Although, they have achieved substantial speed-up for their application in proof-carrying code, it is not as general as the technique we have presented here.

So far, we have implemented and successfully used higher-order substitution trees in the context of higher-order tabled logic programming. Another interesting use is in indexing the actual higher-order logic program. While the table is a dynamically built index, indexing of program clauses can be done statically. Although the general idea of substitution trees is also applicable in this setting there are several important optimizations, such as unification factoring [2], we plan do adapt to the higher-order setting in the future.

Acknowledgments

This work could not have been accomplished without the insights, persistence and critical feedback of my advisor Frank Pfenning.

References

1. Rowan Davies and Frank Pfenning. Intersection types and computational effects. In *Proceedings of the International Conference on Functional Programming (ICFP 2000), Montreal, Canada*, pages 198–208. ACM Press, 2000.
2. Steve Dawson, C. R. Ramakrishnan, Steve Skiena, and Terrance Swift. Principles and practice of unification factoring. *ACM Transactions on Programming Languages and Systems*, 18(6):528–563, 1995.
3. Peter Graf. Substitution tree indexing. In *Proceedings of the 6th International Conference on Rewriting Techniques and Applications, Kaiserslautern, Germany*, Lecture Notes in Computer Science (LNCS) 914, pages 117–131. Springer-Verlag, 1995.
4. Michael Hanus and Christian Prehofer. Higher-order narrowing with definitional trees. *Journal of Functional Programming*, 9(1):33–75, 1999.

5. Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.
6. Lars Klein. Indexing für Terme höherer Stufe. Diplomarbeit, FB 14, Universität des Saarlandes, Saarbrücken, Germany, 1997.
7. Dale Miller. Unification of simply typed lambda-terms as logic programming. In *Eighth International Logic Programming Conference*, pages 255–269, Paris, France, June 1991. MIT Press.
8. Gopalan Nadathur and Dale Miller. An overview of λProlog. In Kenneth A. Bowen and Robert A. Kowalski, editors, *Fifth International Logic Programming Conference*, pages 810–827, Seattle, Washington, August 1988. MIT Press.
9. Aleksander Nanevski, Brigitte Pientka, and Frank Pfenning. A modal foundation for meta-variables. In *2nd ACM SIGPLAN Workshop on Mechanized Reasoning about Languages with variable binding (Merlin), Uppsala, Sweden*. to appear, August 2003.
10. G. Necula and S. Rahul. Oracle-based checking of untrusted software. In *28th ACM Symposium on Principles of Programming Languages (POPL'01)*, pages 142–154, 2001.
11. Lawrence C. Paulson. Natural deduction as higher-order resolution. *Journal of Logic Programming*, 3:237–258, 1986.
12. Frank Pfenning. Unification and anti-unification in the Calculus of Constructions. In *Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 74–85, Amsterdam, The Netherlands, July 1991.
13. Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206, Trento, Italy, July 1999. Springer-Verlag Lecture Notes in Artificial Intelligence (LNAI) 1632.
14. Brigitte Pientka. Memoization-based proof search in LF: an experimental evaluation of a prototype. In *Third International Workshop on Logical Frameworks and Meta-Languages (LFM'02), Copenhagen, Denmark*, Electronic Notes in Theoretical Computer Science (ENTCS), 2002.
15. Brigitte Pientka. A proof-theoretic foundation for tabled higher-order logic programming. In P. Stuckey, editor, *18th International Conference on Logic Programming, Copenhagen, Denmark*, Lecture Notes in Computer Science (LNCS), 2401, pages 271 –286. Springer-Verlag, 2002.
16. Brigitte Pientka. *Tabled higher-order logic programming*. PhD thesis, Department of Computer Sciences, Carnegie Mellon University, 2003. forthcoming.
17. Brigitte Pientka and Frank Pfennning. Optimizing higher-order pattern unification. In F. Baader, editor, *19th International Conference on Automated Deduction, Miami, USA*, Lecture Notes in Artificial Intelligence (LNAI) 2741, pages 473–487. Springer-Verlag, July 2003.
18. I. V. Ramakrishnan, P. Rao, K. Sagonas, T. Swift, and D. Warren. Efficient access mechanisms for tabled logic programs. *Journal of Logic Programming*, 38(1):31–54, Jan 1999.
19. I. V. Ramakrishnan, R. Sekar, and A. Voronkov. Term indexing. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume 2, pages 1853–1962. Elsevier Science Publishers B.V., 2001.
20. David S. Warren. *Programming in tabled logic programming*. draft available from http://www.cs.sunysb.edu/w̃arren/xsbbook/book.html, 1999.