Higher-order term indexing using substitution trees

BRIGITTE PIENTKA
McGill University

We present a higher-order term indexing strategy based on substitution trees for simply typed lambda-terms. There are mainly two problems in adapting first-order indexing techniques. First many operations used in building an efficient term index and retrieving a set of candidate terms from a large collection are undecidable in general for higher-order terms. Second, the scoping of variables and binders in the higher-order case presents challenges.

The approach taken in this paper is to reduce the problem to indexing linear higher-order patterns, a decidable fragment of higher-order terms, and delay solving terms outside of this fragment. We present insertion of terms into the index based on computing the most specific linear generalization of two linear higher-order patterns, and retrieval based on matching two linear higher-order patterns. Our theoretical framework maintains that terms are in $\beta\eta$-normal form, thereby eliminating the need to re-normalize and raise terms during insertion and retrieval. Finally, we prove correctness of our presented algorithms. This indexing structure is implemented as part of the Twelf system to speed up the execution of the tabled higher-logic programming interpreter.

Categories and Subject Descriptors: F.4.1 [**Theory of Computation**]: Mathematical Logic and Formal Languages; D.3.3 [**Software**]: Language Constructs and Features—*Frameworks*

General Terms: Design, Theory

Additional Key Words and Phrases: Indexing, type theory, logical frameworks

1. INTRODUCTION

First-order logic programming and theorem proving systems have developed into highly sophisticated automated reasoning systems with remarkable performance over the last decades [Ramakrishnan et al. 2001]. This success is to a large extent due to term indexing techniques, which support these systems to manage and use redundancy elimination techniques. In general, term indexing is concerned with compactly storing a large collection of terms and rapidly retrieving a set of candidate terms satisfying some property (e.g. unifiability, instance, variant, etc.) from a large collection of terms.

There are many examples where term indexing is used. In logic programming, for example, we need to select all clauses from the program whose head unifies with the current goal [Ramesh et al. 1990; Chen et al. 1994; Dawson et al. 1995; Dawson et al. 1995]. In tabled logic programming we memoize intermediate goals in a table and reuse their results later in order to eliminate redundant and infinite computation. Here we need to find all entries in the table such that the current goal is a variant or an instance of a table entry and re-use the associated answers [Ramakrishnan et al. 1999; 1995]. Similarly, in theorem proving we keep track of previously derived formulas to eliminate redundancy and detect loops [McCune 1992; Graf 1995b; Riazanov and Voronkov 2002; Hillenbrand 2003]. Since rapid retrieval and efficient storage of large collections of terms plays a central role in logic programming and in proof search in general, a variety of indexing techniques have been proposed for

first-order terms (see [Ramakrishnan et al. 2001] for a survey). However, indexing techniques for higher-order terms, i.e. terms that contain lambda-abstractions, are largely missing thereby severely hampering the performance of higher-order systems and limiting their potential applications. There are mainly two problems in adapting first-order indexing techniques. First, many operations used in building an efficient term index and retrieving a set of candidate terms from a large collection are undecidable in general in the higher-order setting. Second, the scoping of variables and binders in the higher-order case presents challenges.

In this paper, we present a higher-order term indexing technique based on substitution trees. Substitution tree indexing [Graf 1995a] is a highly successful first-order term indexing strategy which allows the sharing of common sub-expressions via substitutions. We extend this idea to the higher-order setting and present an indexing technique for higher-order terms.

The challenge in the higher-order setting is that many common operations on higher-order terms which are necessary to build and maintain substitution trees or retrieve elements from the index are undecidable in general. For example, to build a substitution tree, we compute the most specific common generalization between two terms. However, in general the most specific generalization of two terms does not exist in the higher-order setting. Similarly, retrieving all terms, which unify or match, needs to be efficient – but higher-order unification is undecidable in general. Fortunately, there exists a fragment called higher-order patterns for which checking unifiability of two terms and computing the most specific generalization between two terms is decidable [Miller 1991b; Pfenning 1991]. However, even for this fragment algorithms may not be efficient in practice [Pientka and Pfenning 2003] and are sufficiently complex that it is not obvious that they are a suitable basis for higher-order term indexing techniques.

In this paper we propose a general strategy for indexing higher-order terms where we first translate higher-order terms to linear higher-order patterns together with constraints, and second we store linear higher-order patterns together with their constraints in a substitution tree. Linear higher-order patterns refine the notion of higher-order patterns further and factor out computationally expensive parts into constraints. As we have shown in [Pientka and Pfenning 2003] many terms encountered in practice fall into this fragment. Moreover, linear higher-order pattern unification performs well in practice. In this paper, we demonstrate that linear higher-order patterns are well suited to elegantly describe term indexing operations such as computing the most specific linear generalization or checking unifiability of two terms. Moreover, we give algorithms for inserting linear higher-order patterns into an index and for retrieving a set of terms from the index such that the query is an instance of the term in the index and prove the correctness of these operations. We concentrate on simply typed terms in this paper. However, the presented techniques can be generalized to the dependently typed setting (see [Pientka 2003b]) and are in fact implemented as part of the logical framework Twelf system [Pfenning and Schürmann 1999]. We have used higher-order substitution trees to speed-up the execution of the tabled logic programming interpreter [Pientka 2002; 2005] and to facilitate the generation of small proof witnesses [Sarkar et al. 2005]. Preliminary results have been published in [Pientka 2003a], and this paper expands

the theoretical results. The theoretical framework we present can directly serve as a general foundation for higher-order term indexing in many higher-order logic programming systems such as λProlog [Nadathur and Mitchell 1999], and higher-order theorem provers such as Bedwyr [Baelde et al. 2007], Isabelle [Paulson 1986], or Leo-II [Benzmüller et al. 2008].

The paper is organized as follows: In Section 2, we present the general idea of higher-order substitution trees. In Section 3, we give the theoretical background and in Section 4 we describe higher-order substitution trees more formally. In Section 5, we give algorithms for computing the most specific linear generalization of two terms and inserting terms into the index and retrieval is discussed in Section 6. This is followed by a discussion on extending the framework to dependently-typed terms in Section 7. We conclude with a discussion of experimental results within the tabled higher-order logic programming engine in the Twelf system (Section 8) and related work (Section 9).

2. HIGHER-ORDER SUBSTITUTION TREES

We illustrate the general idea of substitution tree indexing using a first-order example and then focus on indexing of higher-order terms. In particular, we highlight some of the subtle issues concerning the interplay of bound variables and meta-variables. We will consider several examples in the logical framework Twelf [Pfenning and Schürmann 1999], which provides a higher-order logic programming engine to execute specifications to make the examples more concrete. However, the issues arising are similar in other higher-order systems which deal with lambda-terms.

Example 1. To illustrate the basic idea consider the example of equality transformations for propositional logic, described by $A \Leftrightarrow B$. In the logical framework Twelf [Pfenning and Schürmann 1999], we first declare a data-type `prop` for propositions, together with constructors for propositions such as conjunction, implication, disjunction and negation as follows.

```
prop:  type.
and:   prop -> prop -> prop.        or:   prop -> prop -> prop.
imp:   prop -> prop -> prop.        neg:  prop -> prop.
```

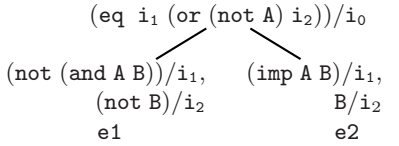
Next, we present some equivalence preserving transformations on propositions together with their encoding in Twelf using the predicate `eq` .

$A \Leftrightarrow B$	`eq : prop → prop → type.`
$e_1 : \neg(A \wedge B) \Leftrightarrow (\neg A) \vee (\neg B).$	`e1 : eq (not (and A B)) (or (not A) (not B)).`
$e_2 : (A \supset B) \Leftrightarrow (\neg A) \vee B.$	`e2 : eq (imp A B) (or (not A) B).`
$e_3 : A \supset \neg B \Leftrightarrow (\neg A) \vee (\neg B).$	`e3 : eq (imp A (not B)) (or (not A) (not B)).`

First, we define predicate `eq` which represents the judgment for equivalence preserving transformation between two propositions. Next, we represent each equivalence transformation as a clause in a logic program. To efficiently find the clause head which unifies with a given goal, we would like to store these clauses in an index, i.e. a data-structure which supports sharing of structure among these clause heads. As we see, the three transformations share quite a lot of structure, and in fact the last one is an instance of the previous one. We will use it to illustrate some

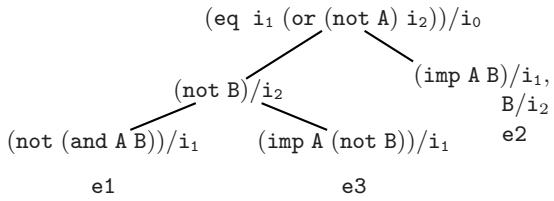
specific issues when inserting these clauses one by one into a substitution tree.

Our intention is to share common structure of terms in order to share common operations. For example, when checking whether a term U unifies with any of the given clauses, we only want to perform the comparison against the constructor `or` and the sub-term (`not A`) once. To achieve this, we compute the most specific generalization between the given terms. For example, the most specific generalization of the first and second clause stated is, `eq i1 (or (not A) i2)` where we can obtain the clause `e1` by instantiating $\mathtt{i}_1$ with (`not (and A B)`) and $\mathtt{i}_2$ with (`not B`). Similarly, we can obtain the clause `e2` by instantiating $\mathtt{i}_1$ with (`imp A B`) and $\mathtt{i}_2$ with `B`. $\mathtt{i}_0$,$\mathtt{i}_1$, $\mathtt{i}_2, \ldots$ denote meta-variables which represent holes in terms. A term can be represented as a sequence of substitutions. For example, the clause `e1` can be described as $[\![(\mathtt{not\ (and\ A\ B)})/\mathtt{i}_1, (\mathtt{not\ B})/\mathtt{i}_2]\!](\mathtt{eq\ \ i_1\ (or\ (not\ A)\ i_2)})$. A substitution tree is a tree where each node contains a set of substitutions. The tree containing the clause `e1` and `e2` is given next. The original clauses can be obtained by composing all the substitutions along one branch. To easily identify, which branch corresponds to which clause, we labelled the leafs with the name of the clause.



Let us now consider what happens, when we insert `e3`. The clause is an instance of the root of the tree, and we obtain the substitution (`imp A (not B)`)/$\mathtt{i}_1$ and (`not B`)/$\mathtt{i}_2$. Hence we call the clause head and the substitution at the root of the tree fully compatible, and we now continue to insert this substitution into one of the children. However, the substitution (`imp A (not B)`)/$\mathtt{i}_1$ and (`not B`)/$\mathtt{i}_2$ is not fully compatible with any of the substitutions in the children; hence we need to split one of the children. There are in fact two possible splits, and therefore the two possible substitution trees are shown in Figure 1.

Inserting `e3` to the left:



Inserting `e3` to the right:

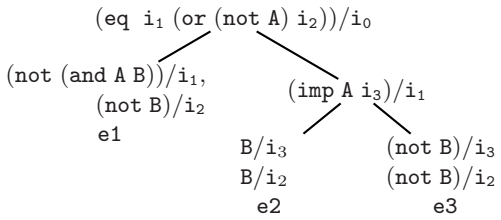


Fig. 1. Substitution tree after inserting `e3`

The choice on how to split nodes when inserting new elements in a tree must be resolved in practice. Typically, we can make more informed choices if the full set of terms we want to index is known in advance. This is for example the case when indexing clauses in a logic program, and we can employ techniques such as unification factoring [Dawson et al. 1995] to obtain an optimal tree. In theorem proving or tabled logic programming on the other hand we store intermediate sub-goals and formulas which are encountered during proof search. Consequently, elements are inserted into the index incrementally, and less information is available to make informed splitting choices. While resolving these choices in practice is important, we focus in this paper on extending substitution trees to indexing higher-order terms. This means we will concentrate on the essential operation of computing the most specific generalization of terms and substitutions in the higher-order setting.

Example 2. To illustrate the higher-order issues, we consider some well-known equivalence preserving transformation in first-order logic.

$$\begin{aligned} e_5 &: (\forall x.A(x)) \vee B \;\Leftrightarrow\; \forall x.A(x) \vee B \\ e_6 &: A \vee (\forall x.B(x)) \;\Leftrightarrow\; \forall x.A \vee B(x) \\ e_7 &: (\forall x.A(x)) \supset B \;\Leftrightarrow\; \forall x.A(x) \supset B \end{aligned}$$

Of course these stated equality preserving transformations are only valid, if certain bound variable conditions are satisfied. For example, the first translation requires that the bound variable x does not occur in B. These conditions are naturally enforced using higher-order abstract syntax where bound variables in the object language are represented by bound variables in the meta-language. We first define the constructor `forall` with type $(\texttt{i} \to \texttt{prop}) \to \texttt{prop}$. The universal quantifier $\forall x.A(x)$ is then encoded as `forall` λ`x.A x` and bound variables are represented by the λ-binder. The representation of the equivalence preserving transformations is given below.

```
e5 : eq (or    (forall λx.A x)        B        ) (forall λx.(or   (A x)   B)).
e6 : eq (or          A         (forall λx.B x) ) (forall λx.(or    A   (B x))).
e7 : eq (imp  (forall λx.A x)        B        ) (forall λx.(imp  (A x)   B)).
```

In the higher-order setting, meta-variables denote a *closed* instance of terms. The meta-variable `A` in the first translation denotes a function which is applied to `x`, `B` however denotes an atomic proposition, and hence cannot depend on the bound variable `x`. As this example illustrates, bound variable dependencies are naturally and elegantly encoded in this higher-order setting, and higher-order unification must enforce these variable dependencies. To highlight the common structure between the three transformation, we have inserted appropriate spaces.

Inspecting the three given clauses closely, we observe that they share a lot of structure. For example clause `e5` and `e6` "almost" agree on the second argument. Our goal is to share common structure of terms in order to share common operations even below a binder. This means for example that we would like to share the term `forall` λ`x.`$\Box$ where $\Box$ is instantiated with `or (A x) B` to obtain the terms `e5`, and we instantiate $\Box$ with `or A (B x)` to obtain `e6`. Finally, to obtain `e7`, we instantiate $\Box$ with with `(imp (A x) B)`. Note that in all these cases $\Box$ is instantiated with an open

term which is allowed to refer to the bound variable x. Our indexing structure supports such sharing of expressions even in the presence of binders and allows instantiations with open terms, i.e. terms which may contain bound variables. This is unlike the first-order case where holes were always instantiated with closed terms. To achieve this, we characterize holes as a closure of a meta-variable i together with a delayed substitution. This substitution precisely characterizes the dependencies we allow when instantiating the meta-variable with an open term. For example, (forall λx.□) is denoted by (forall λx.i[x/y]) where i is a meta-variable and [x/y] is a postponed substitution. When we instantiate the meta-variable i with (or (A y) B) we will apply the substitution [x/y] which essentially renames the variable y to x, and yields as a final result (forall λx.or (A x) B). In the term or □ A, the hole denoted by □ can be instantiated with a closed term. In this case we write $i[.]$ for □. Since there are no bound variables involved the postponed substitution we associate with the meta-variable i is the empty.

Associating meta-variables with a postponed substitution is a known technique from explicit substitution calculus. However instead of using the explicit substitution calculus based in de Bruijn indices [Abadi et al. 1990; Dowek et al. 1995], we use the contextual modal type theory [Nanevski et al. 2008] as a foundation which provides a high-level explanation of meta-variables. Characterizing holes in terms as a closure of meta-variable and a postponed substitution will allow us to instantiate holes using first-order replacement.

This has two key advantages: First, the term representation is more compact than other representations. Traditionally, we would represent the hole in the term forall λx.□ with the application I x where I represent the meta-variable which is subject to instantiation. This means in general, if we have n bound variables $x_1, \ldots, x_n$, the meta-variable I which is inserted must be applied to all of these variables. Second, to obtain the original term, we must instantiate I in the term forall λx.I x with the lambda-abstraction λy.or (A y) B. To compare the result of this instantiation with the original term, we must now re-normalize the term, β-reducing the created redex (λy.or (A y) B) x to [x/y](or (A y) B) = or (A x) B. The goal of treating meta-variables as closures consisting of the meta-variable itself and a postponed substitution avoids this re-normalization step, but allows direct replacement of the meta-variable with the appropriate instantiation. In an implementation with de Bruijn indices and explicit substitutions [Abadi et al. 1990; Dowek et al. 1995], the postponed renaming substitution can in fact be omitted[1]. As a consequence, we not only obtain direct in-place update of meta-variables, but also a compact representation of the term with the hole. This has important consequences for the theoretical development as well as implementations. The formal description of the operations and their correctness proofs will be substantially simpler since we do not need to consider re-normalization. Instead, we will maintain that all terms are in canonical forms. It has also important practical implications. Because we treat meta-variables as closures our formal description can be directly implemented using de Bruijn indices and explicit substitutions following similar ideas already employed for implementing higher-order unification [Abadi et al. 1990; Dowek et al. 1995].

[1]More precisely, this corresponds to shifting de Bruijn indices by zero.

To insert these three clauses into a substitution tree, we need to compute the most specific common generalization. As mentioned earlier, this problem is in general undecidable in the higher-order setting. To do this in a simple manner, we first translate terms into linear higher-order patterns [Pientka and Pfenning 2003] together with constraints. Linear higher-order patterns refine the notion of higher-order patterns [Miller 1991b; Pfenning 1991] where every meta-variable must be applied to *some* distinct bound variables in two ways: First, linear higher-order patterns require that every meta-variable occurs only once and in addition every meta-variable is applied to *all* distinct bound variables in its context.

Maintaining these two conditions yield a simple algorithm and allows us to delay the occurs check during retrieval and any other complicated conditions involving bound variable occurrences. For example, the term (`forall` λ`x.(imp p(x) q(x)))` should not unify with (`forall` λ`x.(imp (A x) B)`), because we must instantiate the meta-variable `B` with a closed term. Hence, it is invalid to replace B with `q(x)`. Checking for bound variable dependencies in the higher-order setting is as expensive as the occurs check, and requires a traversal of a term; hence it is beneficial to delay any necessary bound variable checks.

As we observe, meta-variables `A` and `B` occur more than once in all of the above clauses, and moreover the occurrence of `B` is not fully applied. For example, in `eq` (`or` (`forall` λ`x.A x`) `B`) (`forall` λ`x.(or (A x) B)` the meta-variable `B` does not depend on the bound variable `x` in (`forall` λ`x.(imp (A x) B)`) although it occurs within the scope of the binder `x`. Hence, `B` is not a linear higher-order pattern, since it is not applied to all bound variables in whose scope it occurs. In addition, the meta-variable `A` occurs twice. Before inserting the clauses into a substitution tree, we therefore first linearize terms by eliminating any duplicate occurrences of meta-variables, and replacing any meta-variable which is not fully applied with one which is. This linearization step takes as input a clause, and produces a clause which only contains linear higher-order patterns together with some constraints described by $\doteq$. The program after linearization is shown next:

`e5` : `eq` (`or` (`forall` λ`x.A x`) `B`) (`forall` λ`x.or` (`A'` `x`) (`B'` `x`)).
$\forall$`x.`(`A'` `x`) $\doteq$ (`A x`) $\wedge$ $\forall$`x.B'` `x` $\doteq$ `B`

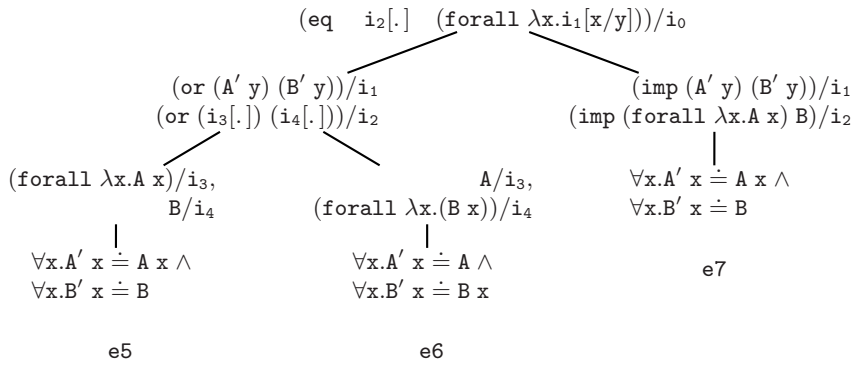
`e6` : `eq` (`or A` (`forall` λ`x.B x`)) (`forall` λ`x.or` (`A'` `x`) (`B'x`)).
$\forall$`x.`(`A'` `x`) $\doteq$ `A` $\wedge$ $\forall$`x.B'` `x` $\doteq$ (`B x`)

`e7` : `eq` (`imp` (`forall` λ`x.A x`) `B`) (`forall` λ`x.(imp` (`A'` `x`) (`B'` `x`))).
$\forall$`x.`(`A'` `x`) $\doteq$ (`A x`) $\wedge$ $\forall$`x.B'` `x` $\doteq$ `B`

We view linearization as a standardization step, which is also in a simpler form present in first-order indexing techniques. In the first-order setting, terms are linearized and duplicate occurrences of meta-variables are factored out in order to postpone the occurs check. Our notion of linear higher-order patterns establishes a criteria with the same intentions of factoring out expensive operations in the higher-order setting. Together with the linear term, we simply store variable definitions, which establish the equality between these two meta-variables.

When inspecting the translated clauses `e5`, `e6`, and `e7`, even more sharing becomes apparent. For example, the clauses `e5` and `e6` agree upon the last argument. We now compute the most specific generalization between these clauses, and can

build up a substitution tree. Each node in the substitution tree contains a set of substitutions, and variable definitions which resulted from linearizing terms are found at the leafs.



Summary. To insert a term R in a substitution tree, we first translate it into a linear higher-order pattern R', together with constraints C. Insertion of a term R', into the index can then be viewed as insertion of the substitution R'/i_0. To insert this substitution, we compute the most specific linear generalization between the given substitution and the one in the tree. This will require us to split nodes which are not fully compatible. In the remainder of the paper, we introduce the theoretical framework for simply typed lambda-terms with first-class meta-variables and formalize key operations such as computing the most-specific generalization of two terms and two substitutions. Based on this development, we define precisely criteria when two terms (or two substitutions) are compatible, and how to insert and retrieve a substitution from a given substitution tree.

3. BACKGROUND

3.1 Contextual modal type theory

In this section we introduce a simply typed lambda-calculus with first-class meta-variables which allows the instantiation of meta-variables with open terms [Nanevski et al. 2008]. Previously, we have used the contextual modal type theory as a foundation for describing linear higher-order pattern unification [Pientka and Pfenning 2003; Pientka 2003b].

$$
\begin{array}{rrcl}
\text{Types} & A, B, C & ::= & P \mid A \to B \\
\text{Normal Objects} & M, N & ::= & \lambda x.M \mid R \\
\text{Atomic Objects} & R & ::= & H \cdot S \mid u[\sigma] \\
\text{Head} & H & ::= & x \mid c \\
\text{Spines} & S & ::= & \mathsf{nil} \mid M; S \\
\text{Contexts} & \Gamma, \Psi & ::= & \cdot \mid \Gamma, x{:}A \\
\text{Substitutions} & \sigma, \tau & ::= & . \mid \sigma, N/x \mid \sigma, H /\!\!/ x \\
\text{Modal Contexts} & \Delta & ::= & \cdot \mid \Delta, u{::}P[\Psi]
\end{array}
$$

We write P for base types, and function types are denoted by $A \to B$. We enforce terms to be in normal form by exploiting a presentation technique due to

Watkins et al. [Watkins et al. 2002] and described in detail in [Nanevski et al. 2008]. While the syntax only guarantees that terms N are normal (that is, contain no β-redices), the typing rules will in addition guarantee that all well-typed terms are fully η-expanded. We will use the spine notation [Cervesato and Pfenning 2003] to describe atomic terms. The term $((c\ M_1)\ M_2)$ is for example represented in spine form as $c \cdot (M_1; M_2; \mathsf{nil})$. This is convenient when we compare, analyze and manipulate terms because we have direct access to the head c of the term. It is also close to our implementation where we represent terms using spines.

As we mentioned, we concentrate on objects in normal form. This has two reasons: First, these are the only meaningful objects in the logical framework, but in addition equality between two terms in normal form is easily decided by simply checking whether the two terms are syntactically equal up to α-renaming. Working only with normal forms in our theoretical development significantly simplifies our presentation and proofs. Moreover, it is also close to our actual implementation which maintains normal forms when inserting and retrieving terms from an index. This means we avoid re-normalization of terms in the implementation.

We follow a bi-directional type checking approach which characterizes objects in canonical form. In order to achieve this we divide the term calculus into *atomic objects* R and *normal objects* M. Contexts Γ and Ψ contain only declarations $x{:}A$ where A is normal and all terms occurring in substitutions σ are either normal (in N/x) or replace x by a head (in $H/\!\!/x$). The modal context Δ contains declarations of meta-variables $u{::}P[\Psi]$. We enforce that all meta-variables occurring in well-typed terms must be of base type, i.e. they are lowered. Because any occurrence of a meta-variable $u[\sigma]\ M$ where u is of function type $(A \to B)[\Psi]$ and M has type A can be replaced by a new meta-variable $v[\sigma, M/x]$ where v has type $B[\Psi, x{:}A]$ (see also [Dowek et al. 1995; Pientka and Pfenning 2003]), this is not a restriction.

We assume global constants are declared in a signatures Σ and never change in the course of a typing derivation. We therefore suppress the signatures throughout. Moreover, we only consider well-formed types. Typing at the level of objects is divided into four judgments:

$\Delta; \Gamma \vdash M \Leftarrow A$	Check normal object M against A
$\Delta; \Gamma \vdash R \Rightarrow P$	Synthesize base type P for atomic object R
$\Delta; \Gamma \vdash S > A \Rightarrow P$	Synthesize base type P for spine S and A
$\Delta; \Gamma \vdash \sigma \Leftarrow \Psi$	Check σ against Ψ

We always assume that Δ and Γ and the subject (M, R, or σ) are given, and that the contexts Δ and Γ are well-formed. For synthesis $R \Rightarrow P$ we assume R is given and we generate an base type P. Similarly, for $S > A \Rightarrow P$ we assume S and A to be given, but infer the type P. The typing rules are given in Figure 2.

In general, introduction forms for a type constructor break down a type when read from the conclusion to the premise until it is atomic. In our case we only have two types, base types and functions. Lambda-abstractions are checked against their type. When checking a normal object that happens to be atomic (that is, has the form R) against a type P we synthesize the type P for R. Technically, we synthesize a type P' which must be equal to P. We will leave this comparison implicit to simplify the development.

In the case where R has base type, it is either a meta-variable $u[\sigma]$ or it is of the

Normal objects

$$\frac{\Delta;\Gamma, x{:}A \vdash M \Leftarrow B}{\Delta;\Gamma \vdash \lambda x.M \Leftarrow A \to B}\ \mathsf{Lam} \qquad \frac{\Delta;\Gamma \vdash R \Rightarrow P}{\Delta;\Gamma \vdash R \Leftarrow P}\ \Rightarrow\Leftarrow$$

Atomic objects

$$\frac{\Delta;\Gamma, x{:}A, \Gamma' \vdash S > A \Rightarrow P}{\Delta;\Gamma, x{:}A, \Gamma' \vdash x \cdot S \Rightarrow P}\ \mathsf{var} \qquad \frac{c{:}A \in \Sigma \quad \Delta;\Gamma \vdash S > A \Rightarrow P}{\Delta;\Gamma \vdash c \cdot S \Rightarrow P}\ \mathsf{con}$$

$$\frac{\Delta, u{::}P[\Psi], \Delta';\Gamma \vdash \sigma \Leftarrow \Psi}{\Delta, u{::}P[\Psi], \Delta';\Gamma \vdash u[\sigma] \Rightarrow P}\ \mathsf{mvar}$$

Spines

$$\frac{\Delta;\Gamma \vdash S > B \Rightarrow P \quad \Delta;\Gamma \vdash M \Leftarrow A}{\Delta;\Gamma \vdash M\ ;\ S > A \to B \Rightarrow P}\ \mathsf{Scons} \qquad \frac{}{\Delta;\Gamma \vdash \mathsf{nil} > P \Rightarrow P}\ \mathsf{Snil}$$

Substitutions

$$\frac{\Delta;\Gamma \vdash \sigma \Leftarrow \Psi \quad \Delta;\Gamma \vdash M \Leftarrow A}{\Delta;\Gamma \vdash (\sigma, M/x) \Leftarrow (\Psi, x{:}A)}\ \mathsf{sNorm} \qquad \frac{}{\Delta;\Gamma \vdash (\cdot) \Leftarrow (\cdot)}\ \mathsf{sEmpty}$$

$$\frac{\Delta;\Gamma \vdash \sigma \Leftarrow \Psi \quad \Gamma(y) = A}{\Delta;\Gamma \vdash (\sigma, y/\!/x) \Leftarrow (\Psi, x{:}A)}\ \mathsf{sAtom{-}v} \qquad \frac{\Delta;\Gamma \vdash \sigma \Leftarrow \Psi \quad \Sigma(c) = A}{\Delta;\Gamma \vdash (\sigma, c/\!/x) \Leftarrow (\Psi, x{:}A)}\ \mathsf{sAtom{-}c}$$

Fig. 2. Bi-directional typing rules for simply-typed lambda-calculus

form $c \cdot S$ or $x \cdot S$. We first concentrate on the case where $R = c \cdot S$ or $x \cdot S$ where $S = N_1; \ldots; N_n; \mathsf{nil}$. In either case we are able to synthesize its type as follows. We first look up the type $A_1 \to \ldots A_n \to P'$ for c and x resp. Then we can synthesize a type P' for $c \cdot S$ (and $x \cdot S$ resp.), if for all i, N_i checks against A_i.

Because meta-variables are lowered, i.e. they must be of base type P, we can easily synthesize its type by looking up the type for u in the meta-variable context Δ. While $u[\sigma]$ synthesizes a type, we need the type of u, namely $P[\Psi]$ so we can check σ against Ψ. Some renaming is left implicit here, as the variables in the domain of σ should match the variables declared in Ψ.

Substitutions are constructed in two possible ways: either by M/x which replaces the variable x with a normal term M, or by $H/\!/x$ which replaces the variable x directly with a head. Substitutions $H/\!/x$ are necessary so that we can extend a given substitution with $x/\!/x$ when traversing a binding operator. We could not extend substitutions with x/x, since x is not a canonical term. The identity substitutions can now have the form $x_1/\!/x_1, \ldots, x_n/\!/x_n$.

THEOREM 3.1 DECIDABILITY OF TYPE CHECKING. *All judgments in the simply typed contextual modal type theory are decidable.*

PROOF. The typing judgments are syntax-directed and therefore decidable. □

Since our description of substitution trees relies on a concise notion of substitution, we carefully define ordinary substitution for ordinary variables and contextual substitutions for meta-variables.

3.2 Substitution on Terms

In this section we start with defining the operations of substitution on terms. The substitution function we need must construct canonical terms, since those are the

only ones that are well-formed and the only ones of interest. Hence, in places where the ordinary substitution operation would create a redex, in particular when applying the substitution $[M/x]$ to a term $x \cdot S$, we must apply the substitution $[M/x]$ to the spine S, but we also must reduce the redex $(M \cdot [M/x]S)$ which would be created. Since when applying $[M/x]$ to the spine S, we again may encounter situations which require us to contract a redex, the substitution $[M/x]$ must be hereditary. We therefore call this operation *hereditary substitution*.

This technique was first described for logical frameworks in [Watkins et al. 2002], and it only allows for objects which are in canonical form since only these are meaningful for representing object-languages in logical frameworks. Here we use this technique in the simply typed setting to ensure that terms are in $\beta\eta$-normal form thereby eliminating the need to explicitly consider $\beta\eta$-normalization when analyzing, manipulating and comparing terms. The main difficulty in defining hereditary substitutions is that this operation could easily fail to terminate. Consider for example the term which arises when computing the normal form of $(\lambda y.y\ y)\ (\lambda x.x\ x)$. Clearly, on well-typed terms this does not occur.

We define hereditary substitutions as a primitive recursive functional where we pass in the type of the variable we substitute for. This will be crucial in determining termination of the overall substitution operation. If we hereditarily substitute $[\lambda y.M/x](x \cdot S)$, then if everything is well-typed and $x : A_1 \to A_2$, then we will write $[\lambda y.M/x]_{A_1 \to A_2}(x \cdot S)$ indexing the substitution with the type for x. The substitution operation will be total since any side condition can be satisfied by α-conversion. First, we present the ordinary capture-avoiding substitution for a single variable, $[M/x]_A N$, $[M/x]_A S$, and$[M/x]_A \sigma$.

$$
\begin{array}{lcll}
[M/x]_A(\lambda y.N) & = & \lambda y.N' & \text{where } N' = [M/x]_A N \\
 & & & \text{choosing } y \notin \mathsf{FV}(M) \text{ and } y \neq x \\
[M/x]_A(u[\sigma]) & = & u[\sigma'] & \text{where } \sigma' = [M/x]_A \sigma \\
[M/x]_A(c \cdot S) & = & c \cdot S' & \text{where } S' = [M/x]_A S \\
[M/x]_A(x \cdot S) & = & \mathsf{reduce}(M : A, S') & \text{where } S' = [M/x]_A S \\
[M/x]_A(y \cdot S) & = & y \cdot S' & \text{where } y \neq x \text{ and } S' = [M/x]_A S \\
[M/x]_A(\mathsf{nil}) & = & \mathsf{nil} & \\
[M/x]_A(N; S) & = & N'; S' & \text{where } N' = [M/x]_A N \text{ and } S' = [M/x]_A S \\
[M/x]_A(\cdot) & = & \cdot & \\
[M/x]_A(\sigma, N/y) & = & (\sigma', N'/y) & \text{where } \sigma' = [M/x]_A \sigma \text{ and } N' = [M/x]_A N \\
[M/x]_A(\sigma, c/\!\!/y) & = & (\sigma', c/\!\!/y) & \text{where } \sigma' = [M/x]_A \sigma \\
[M/x]_A(\sigma, y'/\!\!/y) & = & (\sigma', y'/\!\!/y) & \text{where } \sigma' = [M/x]_A \sigma \text{ and } x \neq y' \\
[M/x]_A(\sigma, x/\!\!/y) & = & (\sigma', M/y) & \text{where } \sigma' = [M/x]_A \sigma
\end{array}
$$

Next, we concentrate on eliminating possible redices which may have been created in the case $[M/x]_A(x \cdot S)$ using the definition of $\mathsf{reduce}(M : A, S)$.

$$
\begin{array}{lcll}
\mathsf{reduce}(\lambda y.M : A_1 \to A_2, (N; S)) & = & M'' & \text{where } [N/y]_{A_1} M = M' \\
 & & & \text{and } \mathsf{reduce}(M' : A_2, S) = M'' \\
\mathsf{reduce}(R : P, \mathsf{nil}) & = & R & \\
\mathsf{reduce}(M : A, S) & & \text{fails} & \text{otherwise}
\end{array}
$$

We first compute the result of applying the substitution $[M/x]$ to the spine S which yields the spine S'. Second, we reduce any possible redices which are created using the following definition.

Substitution may fail to be defined only if substitutions into the subterms are undefined. The side conditions $y \notin \mathsf{FV}(M)$ and $y \neq x$ do not cause failure, because they can always be satisfied by appropriately renaming y. However, substitution may be undefined if we try for example to substitute an atomic term R for x in the term $x \cdot S$ where the spine S is non-empty. The substitution operation is well-founded since recursive appeals to the substitution operation take place on smaller terms with equal type A, or the substitution operates on smaller types (see the case for $\mathsf{reduce}(\lambda y.M : A_1 \to A_2, (N; S))$).

The first property states that the hereditary substitution operations terminate, independently of whether the terms involved are well-typed or not. The operation may fail if we have ill-typed terms, or yield a canonical term as a result.

Theorem 3.2 Termination.
$[M/x]_A(N)$, $[M/x]_A R$, $[M/x]_A\sigma$, $\mathsf{reduce}(M : A, S)$ terminates, either by returning a result or failing after a finite number of steps.

Proof. This can be verified by a nested induction, first on the structure of A, and second on the structure of the term we apply hereditary substitution to or the term S we apply to $M : A$ in the case for reduce. See [Watkins et al. 2002] or [Nanevski et al. 2008] for similar proofs in related calculi. □

Theorem 3.3 Substitution on Terms.

(1) *If $\Delta; \Gamma \vdash M \Leftarrow A$ and $\Delta; \Gamma, x{:}A, \Gamma' \vdash N \Leftarrow C$ and $[M/x]_A N = N'$ then $\Delta; \Gamma, \Gamma' \vdash N' \Leftarrow C$.*

(2) *If $\Delta; \Gamma \vdash M \Leftarrow A$ and $\Delta; \Gamma, x{:}A, \Gamma' \vdash R \Rightarrow P$ and $R' = [M/x]_A R$ then $\Delta; \Gamma, \Gamma' \vdash R' \Rightarrow P$.*

(3) *If $\Delta; \Gamma \vdash M \Leftarrow A$ and $\Delta; \Gamma, x{:}A, \Gamma' \vdash S > B \Rightarrow P$ and $S' = [M/x]_A S$ then $\Delta; \Gamma, \Gamma' \vdash S' > B \Rightarrow P$.*

(4) *If $\Delta; \Gamma \vdash M \Leftarrow A$ and $\Delta; \Gamma \vdash S > A \Rightarrow P$ then $\mathsf{reduce}(M : A, S) = R$ and $\Delta; \Gamma \vdash R \Rightarrow P$.*

(5) *If $\Delta; \Gamma \vdash M \Leftarrow A$ and $\Delta; \Gamma, x{:}A, \Gamma' \vdash \sigma \Leftarrow \Psi$ and $\sigma' = [M/x]_A \sigma$ then $\Delta; \Gamma, \Gamma' \vdash \sigma' \Leftarrow \Psi$.*

Proof. By simultaneous induction on the definition of substitution, structure of the type A occurring in the type annotation of the substitution $[M/x]_A$ or $\mathsf{reduce}(M : A, S)$ and the second derivation. Either we apply the substitution to a smaller term, or the type A is decreasing or the second derivation is decreasing. □

3.3 Simultaneous Substitutions

Next we define simultaneous substitution $[\sigma]M$ and $[\sigma]\tau$ by extending the ideas presented in the previous section. The substitution is again hereditary. The simultaneous substitution is only total when the substitution σ is defined on all free variables in M and τ, respectively. This will be satisfied, because simultaneous substitution is only applied when the assumptions of the theorem following this

definition are satisfied. Simultaneous substitutions commute with the term constructors, as one would expect. Just as we annotated the substitution $[M/x]_A$ with the type of the variable x to ensure termination of the hereditary substitution operation, we will annotate the simultaneous substitution σ with its domain. However, simultaneous substitution may contain substitutions such as $x/\!\!/x$ and the type of x may not always be available to extend the context annotation (see the case for $[\sigma]_\psi(\lambda y.M)$). Fortunately, it suffices to carry an approximation ψ of its domain Ψ where in fact the type for variables which will be replaced by atomic terms can be omitted. For example, given the simultaneous substitution $[x/\!\!/x, \lambda y.y/y]$ with domain $x{:}Q \to Q, y{:}P \to P$ a valid approximation is $x{:}\_, y{:}P \to P$. Allowing for approximations of contexts simply allows us here to directly extend a context with a declaration $x{:}\_$ where we do not know the type of x. We only need the type for y, because only when we replace it with $\lambda y.y$ a redex may be created, and we must rely on the reduce-operation to ensure that we return a canonical form. We would like to stress that in practice when we implement substitution, we do not need to carry this annotation ψ and it does not need to be calculated explicitely.

$$
\begin{array}{lcll}
[\sigma]_\psi(\lambda y.N) & = & \lambda y.N' & \text{where } N' = [\sigma, y/\!\!/y]_{\psi,y:\_}(N) \\
 & & & \text{choosing } y \notin \mathsf{FV}(\sigma), \mathsf{dom}(\sigma) \\
{[\sigma]_\psi(c \cdot S)} & = & c \cdot S' & \text{where } [\sigma]_\psi(S) = S' \\
{[\sigma]_\psi(x \cdot S)} & = & R' & \text{where } [\sigma]_\psi(S) = S',\ M/x \in \sigma \text{ and } x{:}A \in \psi, \\
 & & & \text{and } R' = \mathsf{reduce}(M : A, S') \\
{[\sigma]_\psi(x \cdot S)} & = & H \cdot S' & \text{where } H/\!\!/x \in \sigma \text{ and } [\sigma]_\psi(S) = S' \\
{[\sigma]_\psi(v[\tau])} & = & v[\tau'] & \text{where } \tau' = [\sigma]_\psi(\tau) \\
{[\sigma]_\psi(\mathsf{nil})} & = & \mathsf{nil} & \\
{[\sigma]_\psi(N; S)} & = & N'; S' & \text{where } N' = [\sigma]_\psi(N) \text{ and } S' = [\sigma]_\psi(S) \\
{[\sigma]_\psi(\cdot)} & = & \cdot & \\
{[\sigma]_\psi(\tau, N/y)} & = & (\tau', N'/y) & \text{where } \tau' = [\sigma]_\psi(\tau) \text{ and } N' = [\sigma]_\psi(N) \\
{[\sigma]_\psi(\tau, c/\!\!/y)} & = & (\tau', c/\!\!/y) & \text{where } \tau' = [\sigma]_\psi(\tau) \\
{[\sigma]_\psi(\tau, x/\!\!/y)} & = & (\tau', H/\!\!/y) & \text{where } \tau' = [\sigma]_\psi(\tau) \text{ and } (H/\!\!/x) \in \sigma \\
{[\sigma]_\psi(\tau, x/\!\!/y)} & = & (\tau', M/y) & \text{where } \tau' = [\sigma]_\psi(\tau) \text{ and } (M/x) \in \sigma
\end{array}
$$

The definition of simultaneous substitutions is a straightforward extension of the ordinary substitution described earlier. Simultaneous substitutions satisfy the simultaneous substitution principle.

Theorem 3.4 Simultaneous Substitution on Terms.

(1) *If* $\Delta; \Gamma \vdash \sigma \Leftarrow \Psi$ *and* $\Delta; \Psi \vdash N \Leftarrow C$ *and* $[\sigma]_\psi N = N'$ *then* $\Delta; \Gamma \vdash N' \Leftarrow C$.

(2) *If* $\Delta; \Gamma \vdash \sigma \Leftarrow \Psi$ *and* $\Delta; \Psi \vdash R \Rightarrow P$ *and* $[\sigma]_\psi R = R'$ *then* $\Delta; \Gamma \vdash R \Rightarrow P$.

(3) *If* $\Delta; \Gamma \vdash \sigma \Leftarrow \Psi$ *and* $\Delta; \Psi \vdash S > A \Rightarrow P$ *and* $[\sigma]_\psi S = S'$ *then* $\Delta; \Gamma \vdash S' > A \Rightarrow P$.

(4) *If* $\Delta; \Gamma \vdash \sigma \Leftarrow \Psi$ *and* $\Delta; \Psi \vdash \tau \Leftarrow \Theta$ *then* $\Delta; \Gamma \vdash [\sigma]_\psi \tau \Leftarrow \Theta$.

Proof. By induction on the structure of the second given derivation and theorem 3.3 (4). □

Finally, we remark that composition of hereditary substitution is written as $[\sigma]_\psi\tau$, and the standard composition principles hold (see [Nanevski et al. 2008]). We emphasize that substitutions σ are defined only on ordinary variables x and not modal variables u. Subsequently, we write id_Γ for the identity substitution $(x_1/\!\!/x_1, \ldots, x_n/\!\!/x_n)$ for a context $\Gamma = (\cdot, x_1{:}A_1, \ldots, x_n{:}A_n)$.

3.4 Contextual substitution

Meta-variables $u[\sigma]$ give rise to new contextual substitutions, which are only slightly more difficult than ordinary substitutions. To understand contextual substitutions, we take a closer look at the closure $u[\sigma]$ which describes the meta-variable. Recall that the substitution σ which is associated with every meta-variable u stands for a postponed substitution. As a consequence, we can apply σ as soon as we know which term u should stand for. Moreover, we require that meta-variables have base type P and hence, we will only substitute atomic objects for meta-variables.

Finally because of α-conversion, the variables that are substituted at different occurrences of u may be different. As a result, substitution for a meta-variable must carry a context, written as $[\![\hat{\Psi}.R/u]\!]N$ and $[\![\hat{\Psi}.R/u]\!]\sigma$ where $\hat{\Psi}$ binds all free variables in R. This complication can be eliminated in an implementation of our calculus based on de Bruijn indexes. In general, we must again ensure that the result is a canonical term, and we will define contextual substitution hereditarily following the ideas for hereditary ordinary substitutions. Just as we annotated the substitution $[M/x]_A$ with the type of the variable x, we will annotate the contextual substitution $[\![\Psi.R/u]\!]_{P[\Psi]}$ with the type of the meta-variable $P[\Psi]$. We will abbreviate $P[\Psi]$ with α for better readability.

$$
\begin{array}{lcll}
[\![\hat{\Psi}.R/u]\!]_\alpha(\lambda y.N) & = & \lambda y.N' & \text{where } N' = [\![\hat{\Psi}.R/u]\!]_\alpha N \\
[\![\hat{\Psi}.R/u]\!]_\alpha(c \cdot S) & = & c \cdot S' & \text{where } S' = [\![\hat{\Psi}.R/u]\!]_\alpha S \\
[\![\hat{\Psi}.R/u]\!]_\alpha(x \cdot S) & = & x \cdot S' & \text{where } S' = [\![\hat{\Psi}.R/u]\!]_\alpha S \\
[\![\hat{\Psi}.R/u]\!]_\alpha(u[\tau]) & = & R' & \text{where } \tau' = [\![\hat{\Psi}.R/u]\!]_\alpha\tau \text{ and } R' = [\tau'/\Psi]_\psi R \\
[\![\hat{\Psi}.R/u]\!]_\alpha(v[\tau]) & = & v[\tau'] & \text{where } \tau' = [\![\hat{\Psi}.R/u]\!]_\alpha\tau \text{ and provided } v \neq u \\
[\![\hat{\Psi}.R/u]\!]_\alpha(\mathsf{nil}) & = & \mathsf{nil} & \\
[\![\hat{\Psi}.R/u]\!]_\alpha(N;S) & = & N';S' & \text{where } N' = [\![\hat{\Psi}.R/u]\!]_\alpha N \text{ and } S' = [\![\hat{\Psi}.R/u]\!]_\alpha S \\
[\![\hat{\Psi}.R/u]\!]_\alpha(\cdot) & = & \cdot & \\
[\![\hat{\Psi}.R/u]\!]_\alpha(\tau, N/y) & = & \tau', N'/y & \text{where } \tau' = [\![\hat{\Psi}.R/u]\!]_\alpha\tau \text{ and } N' = [\![\hat{\Psi}.R/u]\!]_\alpha N \\
[\![\hat{\Psi}.R/u]\!]_\alpha(\tau, H/\!\!/y)_\alpha & = & \tau', H/\!\!/y & \text{where } \tau' = [\![\hat{\Psi}.R/u]\!]_\alpha\tau
\end{array}
$$

Applying $[\![\hat{\Psi}.R/u]\!]$ to the closure $u[\tau]$ first obtains the simultaneous substitution $\tau' = [\![\Psi.R/u]\!]\tau$, but instead of returning $R[\tau']$, it proceeds to eagerly apply τ' to R. Before τ' can be carried out, however, its domain must be renamed to match the variables in Ψ, denoted by τ'/Ψ. For example, when $\tau' = (M_1/x_1, \ldots, M_n/x_n)$ and $\Psi = (y_1{:}A_1, \ldots, y_n{:}A_n)$ then we will rename the domain of the substitution τ' by writing $\tau'/\Psi = (M_1/y_1, \ldots, M_n/y_n)$.

We note that maintaining canonical forms is easy since we enforce that every occurrence of a meta-variable must have base type. While the definition of the discussed case may seem circular at first, it is actually well-founded. The computation

of τ' recursively invokes $[\![\hat{\Psi}.R/u]\!]$ on τ, a constituent of $u[\tau]$. Then τ'/Ψ is applied to R, but applying simultaneous substitutions has already been defined without appeal to meta-variable substitution.

Substitution of a meta-variable satisfies the following contextual substitution property.

THEOREM 3.5 CONTEXTUAL SUBSTITUTION ON TERMS.

(1) *If* $\Delta; \Psi \vdash R \Leftarrow P$ *and* $(\Delta, u{::}P[\Psi], \Delta'); \Gamma \vdash N \Leftarrow C$ *and* $\alpha = P[\Psi]$ *and* $[\![\hat{\Psi}.R/u]\!]_\alpha N = N'$ *then* $(\Delta, \Delta'); \Gamma \vdash N' \Leftarrow C$.

(2) *If* $\Delta; \Psi \vdash R \Leftarrow P$ *and* $(\Delta, u{::}P[\Psi], \Delta'); \Gamma \vdash R' \Rightarrow P'$ *and* $\alpha = P[\Psi]$ *and* $[\![\hat{\Psi}.R/u]\!]_\alpha R' = R''$ *then* $(\Delta, \Delta'); \Gamma \vdash R'' \Rightarrow P'$.

(3) *If* $\Delta; \Psi \vdash R \Leftarrow P$ *and* $(\Delta, u{::}P[\Psi], \Delta'); \Gamma \vdash S > A \Rightarrow P'$ *and* $\alpha = P[\Psi]$ *and* $[\![\hat{\Psi}.R/u]\!]_\alpha S = S'$ *then* $(\Delta, \Delta'); \Gamma \vdash S' > A \Rightarrow P'$.

(4) *If* $\Delta; \Psi \vdash R \Leftarrow P$ *and* $(\Delta, u{::}P[\Psi], \Delta'); \Gamma \vdash \tau \Leftarrow \Theta$ *and* $\alpha = P[\Psi]$ *and* $\tau' = [\![\hat{\Psi}.R/u]\!]_\alpha \tau$ *then* $(\Delta, \Delta'); \Gamma \vdash \tau' \Leftarrow \Theta$.

PROOF. By simple inductions on the second given derivation, appealing to Theorem 3.4 in the case for meta-variables. □

3.5 Simultaneous contextual substitution

Just as we extended ordinary substitutions to simultaneous substitutions, we extend contextual substitution to a simultaneous contextual substitution.

$$\text{Simultaneous contextual substitutions } \theta ::= \cdot \mid \theta, \hat{\Psi}.R/u$$

We write θ for a simultaneous substitution $[\![\hat{\Psi}_1.R_1/u_1, \ldots, \hat{\Psi}_n.R_n/u_n]\!]$. We first define typing rules for simultaneous contextual substitutions.

$$\frac{}{\Delta \vdash (\cdot) \Leftarrow (\cdot)} \qquad \frac{\Delta; \Psi \vdash R \Leftarrow P \quad \Delta \vdash \theta \Leftarrow \Delta'}{\Delta \vdash (\theta, \hat{\Psi}.R/u) \Leftarrow (\Delta', u{::}P[\Psi])}$$

The new operation of substitution is compositional, but two interesting situations arise: when a variable u is encountered, and when we substitute into a lambda-abstraction. We again annotate the simultaneous contextual substitution $[\![\theta]\!]_\Delta$ with its domain.

$$\begin{array}{lcll}
[\![\theta]\!]_\Delta(\lambda y.N) & = & \lambda y.N' & \text{where } N' = [\![\theta]\!]_\Delta N \\
[\![\theta]\!]_\Delta(c \cdot S) & = & c \cdot S' & \text{where } S' = [\![\theta]\!]_\Delta S \\
[\![\theta]\!]_\Delta(x \cdot S) & = & x \cdot S' & \text{where } S' = [\![\theta]\!]_\Delta S \\
[\![\theta]\!]_\Delta(u[\sigma]) & = & R' & \text{where } \theta = (\theta_1, \hat{\Psi}.R/u, \theta_2) \text{ and } \sigma' = [\![\theta]\!]_\Delta(\sigma) \\
 & & & \text{and } R' = [\sigma']_\psi R \text{ where } u{::}P[\Psi] \in \Delta \\
[\![\theta]\!]_\Delta(\mathsf{nil}) & = & \mathsf{nil} & \\
[\![\theta]\!]_\Delta(N; S) & = & (N'; S') & \text{where } N' = [\![\theta]\!]_\Delta N \text{ and } S' = [\![\theta]\!]_\Delta S \\
[\![\theta]\!]_\Delta(\cdot) & = & \cdot & \\
[\![\theta]\!]_\Delta(\sigma, N/y) & = & (\sigma', N'/y) & \text{where } \sigma' = [\![\theta]\!]_\Delta \sigma \text{ and } N' = [\![\theta]\!]_\Delta N \\
[\![\theta]\!]_\Delta(\sigma, H /\!\!/ y) & = & (\sigma', H /\!\!/ y) & \text{where } \sigma' = [\![\theta]\!]_\Delta \sigma
\end{array}$$

We remark that the rule for substitution into a lambda-abstraction does not need to extend the substitution θ nor does it need any other restrictions. This is because the object R 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.

$$[\![\theta]\!]_\Delta(u[\sigma]) \;=\; R' \;\text{ where } \theta = (\theta_1, \hat{\Psi}.R/u, \theta_2) \text{ and } \sigma' = [\![\theta]\!]_\Delta(\sigma) \\ \text{and } R' = [\sigma']_\psi R \text{ where } u{::}P[\Psi] \in \Delta$$

This is clearly well-founded, because σ is a subexpression (so $[\![\hat{\Psi}.R/u]\!]\sigma$ will terminate) and the application of an ordinary substitution has been defined previously without reference to the new form of substitution. Similar to composition of ordinary substitution, composition for contextual substitutions holds (see [Nanevski et al. 2008]).

3.6 Pattern substitutions

An important fragment of higher-order terms, is the pattern fragment. While in general many algorithms such as unification are undecidable in the general higher-order case, these operations become decidable with suitable restrictions to patterns [Miller 1991a]. Higher-order patterns are terms where meta-variables must be applied to some distinct bound variables. In our setting, this means that substitution σ which is associated with the meta-variable $u[\sigma]$ must be a pattern substitution of the form $[x_{\phi(1)} /\!\!/ x_1, \ldots, x_{\phi(n)} /\!\!/ x_n]$. In other words the pattern substitution is just a renaming of some variables.

When we consider only closures of meta-variables together with pattern substitutions then applying the contextual substitution θ to a term M will directly yield a canonical term and it is unnecessary to annotate $[\![\theta]\!]M$ with the domain of θ. In the subsequent development, we therefore omit this annotation.

Finally, we note that applying a contextual substitution θ to a pattern substitution σ does not change σ itself, since the range of σ refers only to bound variables, while θ refers to meta-variables.

Lemma 3.6.
If $\Delta' \vdash \theta \Leftarrow \Delta$ *and* σ *is a pattern substitution, s.t.* $\Delta; \Gamma \vdash \sigma \Leftarrow \Psi$ *then* $[\![\theta]\!]_\Delta(\sigma) = \sigma$.

Proof. Induction on the structure of σ. □

To keep the algorithms for insertion and retrieval similar to the first-order setting and achieve efficient implementations, we will subsequently concentrate on linear higher-order patterns. Linear higher-order patterns enforce two conditions: First, they not only guarantee that every meta-variable is associated with *some distinct bound variables*, but with *all bound variables in whose scope the meta-variable occurs in*. If the meta-variable $u[\sigma]$ occurs in the context Ψ, then σ is a substitution with domain and range Ψ. This restriction ensures that we can avoid expensive bound variable checks when computing instantiations for the meta-variable u. We will write π_Γ for a pattern substitution. Second, we ensure that every meta-variable occurs only once in a given term. This criteria ensures that we can avoid the occurs check when we retrieve unifiable terms, and leads in general to simpler algorithms.

4. FORMALIZING HIGHER-ORDER SUBSTITUTION TREES

Higher-order substitution trees are designed for linear higher-order patterns and are built with contextual substitutions. Recall the example given earlier, where we described equality preserving transformations on logical propositions. One such transformation was the following:

$$\texttt{eq}\ \ (\texttt{or}\ \ (\texttt{forall}\ \lambda\texttt{x.A x})\ \texttt{B})\ \ (\texttt{forall}\ \lambda\texttt{x.}(\texttt{or}\ \ (\texttt{A x})\ \texttt{B})).$$

In this example, A and B denote meta-variables which are present in the original query, while x denotes an ordinary bound variable. In our contextual modal type theory, this term would be represented as follows:

$$\begin{array}{l}\texttt{eq}\cdot((\texttt{or}\quad\cdot((\texttt{forall}\cdot((\lambda\texttt{x.u}[\texttt{x}/\!/\texttt{y}])\ ;\ \mathsf{nil}))\quad;\ \texttt{v}[\cdot]\ ;\ \mathsf{nil}))\ ;\\ \quad(\texttt{forall}\cdot((\lambda\texttt{x.}(\texttt{or}\cdot(\texttt{u}[\texttt{x}/\!/\texttt{y}]\ ;\ \texttt{v}[\cdot]\ ;\ \mathsf{nil})))\ ;\ \mathsf{nil}))\\ \quad\mathsf{nil})\end{array}$$

The meta-variables $\texttt{u}[\texttt{x}/\!/\texttt{y}]$ and $\texttt{v}[\cdot]$ directly encode the dependencies which must be obeyed. The type of u is $\texttt{prop}[\texttt{x} : \texttt{i}]$ while the type of v is $\texttt{prop}[.]$. As we can see, the meta-variable $\texttt{v}[\cdot]$ is not fully applied in the term $(\lambda\texttt{x.}(\texttt{or}\cdot(\texttt{u}[\texttt{x}/\!/\texttt{y}]\ ;\ \texttt{v}[\cdot]\ ;\ \mathsf{nil})))$ because the substitution associated with the meta-variable v is empty although $\texttt{v}[\cdot]$ occurs within the scope of a lambda-binder. Moreover, the meta-variable $\texttt{u}[\texttt{x}/\!/\texttt{y}]$ occurs twice in this term. Hence we translate this term into a linear higher-order pattern:

$$\begin{array}{l}\texttt{eq}\cdot((\texttt{or}\quad\cdot((\texttt{forall}\cdot((\lambda\texttt{x.u}[\texttt{x}/\!/\texttt{y}])\ ;\ \mathsf{nil}))\qquad;\ \texttt{v}[\cdot]\ ;\ \mathsf{nil}))\ ;\\ \quad(\texttt{forall}\cdot((\lambda\texttt{x.}(\texttt{or}\cdot(\texttt{w}_1[\texttt{x}/\!/\texttt{y}]\ ;\ \texttt{w}_2[\texttt{x}/\!/\texttt{y}]\ ;\ \mathsf{nil})))\ ;\ \mathsf{nil}))\ ;\\ \quad\mathsf{nil})\\ \qquad\qquad\text{where } \forall x\,.\,\texttt{w}_1[\texttt{x}/\!/\texttt{y}] \doteq \texttt{u}[\texttt{x}/\!/\texttt{y}]\ \ \wedge\ \ \forall\texttt{x}\,.\,\texttt{w}_2[\texttt{x}/\!/\texttt{y}] \doteq \texttt{v}[\cdot]\end{array}$$

When computing the most specific generalization between two terms to build the substitution tree, we will create internal meta-variables. For example, $\texttt{i}_3[\cdot]$ and $\texttt{i}_4[\cdot]$ are internal meta-variables in

$$\begin{array}{l}\texttt{eq}\cdot((\texttt{or}\quad\cdot(\texttt{i}_3[\cdot]\ ;\ \texttt{i}_4[\cdot]\ ;\ \mathsf{nil}))\ ;\\ \quad(\texttt{forall}\cdot((\lambda x.(\texttt{or}\cdot(\texttt{w}_1[\texttt{x}/\!/\texttt{y}]\ ;\ \texttt{w}_2[\texttt{x}/\!/\texttt{y}]\ ;\ \mathsf{nil})))\ ;\ \mathsf{nil}))\ ;\\ \quad\mathsf{nil})\end{array}$$

In the definition of higher-order substitution trees we will distinguish between a modal context Δ which denotes the original meta-variables such as u, v, and w, a modal context Ω for the internal meta-variables $\texttt{i}_3$ and $\texttt{i}_4$, and a context Γ denoting ordinary variables. A higher-order substitution tree is an ordered n-ary tree.

(1) A node with a contextual substitution ρ such that $\Delta \vdash \rho \Leftarrow \Omega$ and no children is called a leaf and is a tree.

(2) If $N_1, \ldots, N_n$ are trees such that for every i, N_i has a substitution ρ_i, such that $(\Delta, \Omega_i) \vdash \rho_i \Leftarrow \Omega$, and a list of children C_i,

then a node with a contextual substitution ρ, such that $(\Delta, \Omega) \vdash \rho \Leftarrow \Omega'$, and children $N_1, \ldots, N_n$ is a tree.

For every path from the top node ρ_0 where $(\Delta, \Omega_1) \vdash \rho_0 \Leftarrow \Omega_0$ to the leaf node ρ_n, we have $\Delta \vdash [\![\rho_n]\!]([\![\rho_{n-1}]\!] \ldots \rho_0) \Leftarrow \Omega_0$. In other words, there are no internal meta-variables left after we compose all the substitutions ρ_n up to ρ_0. We assume that all meta-variables occurring in one path are unique, and are fully applied, i.e. every meta-variable $u{::}P[\Psi]$ where $\Psi = x_1{:}A_1, \ldots, x_n{:}A_n$ is applied to all the bound variables in Ψ. More formally we can define substitution trees as follows:

$$\begin{array}{lll} \text{Node N} & ::= & (\Omega \vdash \rho \twoheadrightarrow C) \\ \text{Children C} & ::= & [N, C] \mid \mathsf{nil} \end{array}$$

A tree is a node N with a contextual substitution ρ and a list of children C. If the list of children is empty, we have reached a leaf. In general, we will write $\Delta \vdash N : \Omega'$ where $N = (\Omega \vdash \rho \twoheadrightarrow C)$ which means that $(\Delta, \Omega) \vdash \rho \Leftarrow \Omega'$ and all the children N_i in C, $\Delta \vdash N_i : \Omega$. To characterize, well-formed substitution trees we employ the following judgments:

$\Delta \vdash N : \Omega$ Node N is a valid substitution tree with domain Ω
$\Delta \vdash C : \Omega$ The children C are valid substitution trees with domain Ω

We can now formalize the invariants about the domains and range of substitutions occurring in the nodes of the substitution trees. The rule Leaf is a special, but important, case of the rule Node.

$$\frac{N = \cdot \vdash \rho \twoheadrightarrow \mathsf{nil} \quad \Delta \vdash \rho \Leftarrow \Omega}{\Delta \vdash N : \Omega}\ \mathsf{Leaf}$$

$$\frac{N = \Omega_0 \vdash \rho \twoheadrightarrow C \quad C \neq \mathsf{nil} \quad \Delta, \Omega_0 \vdash \rho \Leftarrow \Omega \quad \Delta \vdash C : \Omega_0}{\Delta \vdash N : \Omega}\ \mathsf{Node}$$

$$\frac{}{\Delta \vdash \mathsf{nil} : \Omega}\ \mathsf{Empty} \qquad \frac{\Delta \vdash N : \Omega \quad \Delta \vdash C : \Omega}{\Delta \vdash [N; C] : \Omega}\ \mathsf{Children}$$

Note that there are no typing dependencies among the variables in Ω and they can be arbitrarily re-ordered. Moreover, we point out that while it is convenient to consider the simultaneous contextual substitution ρ_i in the theory, in practice we only carry the mappings which are not the identity.

The algorithms for insertion and retrieval in substitution trees are based on the most specific linear generalization (mslg) and matching. Types themselves do not play a role when computing the mslg and matcher. However, we assume the term is well-typed before it is inserted into the substitution tree, and we will show that the term can be decomposed into contextual substitutions such that their composition results in the original term.

5. INSERTION

Insertion of a term R into the index is viewed as insertion of the substitution $\hat{\Psi}.R/i_0$. Assuming that R has type P in a modal context Δ and a bound variable

context Ψ. $\hat{\Psi}.R/i_0$ is a contextual substitution such that $\Delta \vdash \hat{\Psi}.R/i_0 \Leftarrow i_0{::}P[\Psi]$. This will simplify the following theoretical development. The insertion process works by following down a path in the tree that is *compatible* with the contextual substitution ρ. To formally define insertion, we first describe the most specific linear generalization of two objects, and then show how to compute the most specific linear generalization (mslg) of two contextual substitutions.

5.1 Most specific generalization of two linear objects

Computing the most specific linear generalization of two contextual substitutions relies on finding the most specific linear generalization of two objects. Recall that we require that all objects are linear higher-order patterns and are in canonical forms. Moreover, we assume that all meta-variables are lowered and have base type. We define the computation of the most specific linear generalization of two terms next.

$$
\begin{array}{lll}
(\Delta,\Omega);\Gamma \vdash\ M_1 \sqcup M_2\ : A & \Longrightarrow M/(\Omega',\ \theta_1,\theta_2) & M \text{ is the mslg of } M_1 \text{ and } M_2 \\
(\Delta,\Omega);\Gamma \vdash\ R_1 \underline{\sqcup} R_2\ : P & \Longrightarrow R/(\Omega',\ \theta_1,\theta_2) & R \text{ is the mslg of } R_1 \text{ and } R_2 \\
(\Delta,\Omega);\Gamma \vdash\ S_1 \underline{\sqcup} S_2\ : A > P & \Longrightarrow S/(\Omega',\ \theta_1,\theta_2) & S \text{ is the mslg of } S_1 \text{ and } S_2
\end{array}
$$

If the canonical terms M_1 and M_2 have type A in modal context (Δ,Ω) and bound variable context Γ, then M is the most specific linear generalization of M_1 and M_2 such that $[\![\theta_1]\!]M$ is equal to M_1 and $[\![\theta_2]\!]M$ is equal to M_2. Moreover, θ_1 and θ_2 are contextual substitutions which map meta-variables from Ω' to the modal context (Δ,Ω). Finally, $(\Delta,\Omega');\Gamma \vdash M \Leftarrow A$. Similar invariants hold for atomic terms and spines.

We think of M_1 (R_1, or S_1) as an object which is already in the index and M_2 (R_2, or S_2) is the object to be inserted. As a consequence, only M_1 (R_1, and S_1) may refer to the internal variables in Ω, while M_2 (R_2, and S_2) only depends on Δ. In defining the most specific linear generalization, we distinguish between the internal meta-variables i and the global meta-variables u in the rules, because they play different roles. The inference rules for computing the mslg are given in Figure 3.

In the rule for lambda, we do not need to worry about capture, since meta-variables and bound variables are defined in different context. Hence, we can just traverse the body of the lambda-term. Rule a-mvar-same treats the case where both terms are meta-variables. Note that we require that both meta-variables must be the same and their associated substitutions must also be equal. In the rule a-mvar-diff-1 and a-mvar-diff-2, we just create the substitution $\hat{\Psi}.u[\pi_\Psi]/i$. In general, we would need to create $[\mathsf{id}_\Psi]^{-1}(u[\pi_\Psi])$, but since we know that π is a permutation substitution, we know that $[\mathsf{id}_\Psi]^{-1}(\pi)$ always exists. In addition, the inverse substitution of the identity is the identity.

The different roles of meta-variables u and internal meta-variables i becomes clear in the given rules. In a-mvar-diff-1 and a-mvar-diff-2 we pick a new internal meta-variable i while we re-use the internal meta-variable i in rule a-ivar. If we encounter a meta-variable u and another object R then we generalize and generate a new internal meta-variable i. However, when we encounter an internal meta-variable i and another object R, we do not generate a new internal meta-variable,

Normal linear objects

$$\frac{(\Delta,\Omega);\Gamma, x{:}A_1 \vdash M_1 \sqcup M_2 : A_2 \Longrightarrow M/(\Omega',\theta_1,\theta_2)}{(\Delta,\Omega);\Gamma \vdash \lambda x.M_1 \sqcup \lambda x.M_2 : A_1 \to A_2 \Longrightarrow \lambda x.M/(\Omega',\ \theta_1,\theta_2)}\ \textsf{a-lam}$$

$$\frac{(\Delta,\Omega);\Gamma \vdash R_1 \underline{\sqcup} R_2 : P \Longrightarrow R/(\Omega',\theta_1,\theta_2)}{(\Delta,\Omega);\Gamma \vdash R_1 \sqcup R_2 : P \Longrightarrow R/(\Omega',\ \theta_1,\theta_2)}\ \textsf{a-coe}$$

Atomic linear objects

$$\frac{u{::}P[\Psi]) \in \Delta}{(\Delta,\Omega);\Psi \vdash u[\pi_\Psi] \underline{\sqcup} u[\pi_\Psi] : P \Longrightarrow u[\pi_\Psi]/(\cdot,\cdot,\cdot)}\ \textsf{a-mvar-same}$$

$$\frac{u{::}P[\Psi] \in \Delta \quad i \text{ must be new} \quad R \neq u[\pi_\Psi]}{(\Delta,\Omega);\Psi \vdash u[\pi_\Psi] \underline{\sqcup} R : P \Longrightarrow i[\mathsf{id}_\Psi]/(i{::}P[\Psi],\ \hat{\Psi}.u[\pi_\Psi]/i,\ \hat{\Psi}.R/i)}\ \textsf{a-mvar-diff-1}$$

$$\frac{u{::}P[\Psi] \in \Delta \quad R \neq i_0[\mathsf{id}_\Psi] \text{ for some } i_0 \quad R \neq u[\pi] \quad i \text{ must be new}}{(\Delta,\Omega);\Psi \vdash R \underline{\sqcup} u[\pi_\Psi] : P \Longrightarrow i[\mathsf{id}_\Psi]/(i{::}P[\Psi],\ \hat{\Psi}.R/i,\ \hat{\Psi}.u[\pi]/i)}\ \textsf{a-mvar-diff-2}$$

$$\frac{i{::}P[\Psi] \in \Omega}{(\Delta,\Omega);\Psi \vdash i[\mathsf{id}_\Psi] \underline{\sqcup} R : P \Longrightarrow i[\mathsf{id}_\Psi]/(i{::}P[\Psi],\ \hat{\Psi}.i[\mathsf{id}_\Psi]/i,\ \hat{\Psi}.R/i)}\ \textsf{a-ivar}$$

$$\frac{(\Delta,\Omega);\Psi \vdash S_1 \underline{\sqcup} S_2 : A > P \Longrightarrow S/(\Omega',\theta_1,\theta_2) \quad x{:}A \in \Psi}{(\Delta,\Omega);\Psi \vdash x \cdot S_1 \underline{\sqcup} x \cdot S_2 : P \Longrightarrow x \cdot S/(\Omega',\theta_1,\theta_2)}\ \textsf{a-var}$$

$$\frac{(\Delta,\Omega);\Psi \vdash S_1 \underline{\sqcup} S_2 : A > P \Longrightarrow S/(\Omega',\theta_1,\theta_2) \quad c{:}A \in \Sigma}{(\Delta,\Omega);\Psi \vdash c \cdot S_1 \underline{\sqcup} c \cdot S_2 : P \Longrightarrow c \cdot S/(\Omega',\theta_1,\theta_2)}\ \textsf{a-con}$$

$$\frac{H_1 \cdot S_1 = R_1 \quad R_2 = H_2 \cdot S_2 \quad H_1 \neq H_2 \quad i \text{ must be new}}{(\Delta,\Omega);\Psi \vdash R_1 \underline{\sqcup} R_2 : P \Longrightarrow i[\mathsf{id}_\Psi]/((i{::}P[\Psi]), \hat{\Psi}.R_1/i, \hat{\Psi}.R_2/i))}\ \textsf{a-diff}$$

Normal linear spines

$$\frac{}{(\Delta,\Omega);\Psi \vdash \mathsf{nil} \underline{\sqcup} \mathsf{nil} : P > P \Longrightarrow \mathsf{nil}/(\cdot,\cdot,\cdot)}\ \textsf{a-nil}$$

$$\frac{\begin{array}{l}(\Delta,\Omega);\Psi \vdash M_1 \ \sqcup \ M_2 \ : A_1 \qquad \Longrightarrow M/(\Omega_1,\ \theta_1,\theta_2)\\ (\Delta,\Omega);\Psi \vdash S_1 \ \underline{\sqcup} \ S_2 \ : A_2 > P \ \Longrightarrow S/(\Omega_2,\ \theta_1',\theta_2')\\ \text{and } \Omega' = (\Omega_1,\Omega_2) \quad \theta = (\theta_1,\theta_1') \quad \theta' = (\theta_2,\theta_2')\end{array}}{(\Delta,\Omega);\Psi \vdash (M_1;S_1) \underline{\sqcup} (M_2;S_2) : A_1 \to A_2 > P \Longrightarrow (M;S)/(\Omega',\theta,\theta')}\ \textsf{a-cons}$$

Fig. 3. Linear most specific generalization

because i will be defined later on in the branch of the substitution tree, and we will need to continue to insert R into the tree. This is important for maintaining the invariant that any child of $(\Delta,\Omega_2) \vdash \rho \Leftarrow \Omega_1$ has the form $(\Delta,\Omega_3) \vdash \rho' \Leftarrow \Omega_2$ during insertion (see the insertion algorithm later on).

In rule a-var, a-con, and a-diff, we distinguish on the head symbol H and compute the most specific linear generalization of two objects $H_1 \cdot S_1$ and $H_2 \cdot S_2$. If H_1 and H_2 are not equal, then we generate a new internal meta-variable $i[\mathsf{id}_\Psi]$ together with the substitutions $\hat{\Psi}.(H_1 \cdot S_1)/i$ and $\hat{\Psi}.(H_2 \cdot S_2)/i$ (see a-diff rule). Otherwise, we traverse the spines S_1 and S_2 and compute the most specific linear generalization of them (see rules a-var and a-con). Finally, the rules for computing the most specific generalization of two spines are straightforward. We compute the mslg of all the sub-expressions, and just combine the substitution θ_1 and θ_1' and θ_2 and θ_2' respectively. As we require that all meta-variables occur uniquely, there are no

dependencies among Ω_1 and Ω_2.

Definition 5.1 *Compatibility of atomic objects.*
If $(\Delta, \Omega); \Psi \vdash R_1 \sqcup\!\!\underline{\sqcup} R_2 : P \Longrightarrow i[\mathsf{id}_\Psi]/(i{::}P[\Psi], \hat{\Psi}.R_1/i, \hat{\Psi}.R_2/i)$, then two atomic objects R_1 and R_2 are called incompatible. Otherwise, we call R_1 and R_2 compatible.

In other words, we call two terms compatible, if they share at least the head symbol or a lambda-prefix. We are now ready to prove correctness of the algorithm for computing the most specific linear generalization of linear higher-order patterns.

Theorem 5.2 Soundness of mslg for objects.

(1) *If* $(\Delta, \Omega); \Gamma \vdash M_1 \sqcup M_2 : A \Longrightarrow M/(\Omega', \theta_1, \theta_2)$ *and*
$(\Delta, \Omega); \Gamma \vdash M_1 \Leftarrow A$ *and* $(\Delta, \Omega); \Gamma \vdash M_2 \Leftarrow A$
then $(\Delta, \Omega) \vdash \theta_1 \Leftarrow \Omega'$ *and* $(\Delta, \Omega) \vdash \theta_2 \Leftarrow \Omega'$ *and*
$M_1 = [\![\theta_1]\!]M$ *and* $M_2 = [\![\theta_2]\!]M$ *and* $(\Delta, \Omega'); \Gamma \vdash M \Leftarrow A$.

(2) *If* $(\Delta, \Omega); \Gamma \vdash R_1 \sqcup\!\!\underline{\sqcup} R_2 : P \Longrightarrow R/(\Omega', \theta_1, \theta_2)$ *and*
$(\Delta, \Omega); \Gamma \vdash R_1 \Rightarrow P$ *and* $(\Delta, \Omega); \Gamma \vdash R_2 \Rightarrow P$
then $(\Delta, \Omega) \vdash \theta_1 \Leftarrow \Omega'$ *and* $(\Delta, \Omega) \vdash \theta_2 \Leftarrow \Omega'$ *and*
$R_1 = [\![\theta_1]\!]R$ *and* $R_2 = [\![\theta_2]\!]R$ *and* $(\Delta, \Omega'); \Gamma \vdash R \Rightarrow P$.

(3) *If* $(\Delta, \Omega); \Gamma \vdash S_1 \sqcup\!\!\underline{\sqcup} S_2 : A > P \Longrightarrow S/(\Omega', \theta_1, \theta_2)$ *and*
$(\Delta, \Omega); \Gamma \vdash S_1 > A \Rightarrow P$ *and* $(\Delta, \Omega); \Gamma \vdash S_2 > A \Rightarrow P$
then $(\Delta, \Omega) \vdash \theta_1 \Leftarrow \Omega'$ *and* $(\Delta, \Omega) \vdash \theta_2 \Leftarrow \Omega'$ *and*
$(\Delta, \Omega'); \Gamma \vdash S > A \Rightarrow P$ *and* $S_1 = [\![\theta_1]\!]S$ *and* $S_2 = [\![\theta_2]\!]S$.

Proof. Simultaneous induction on the structure of the first derivation. □

Next, we prove completeness.

Theorem 5.3 Completeness of mslg of terms.

(1) *If* $\Delta, \Omega \vdash \theta_1 \Leftarrow \Omega'$ *and* $\Delta, \Omega \vdash \theta_2 \Leftarrow \Omega'$ *and* θ_1 *and* θ_2 *are incompatible and*
$\Delta, \Omega; \Gamma \vdash M_1 \Leftarrow A$, $\Delta; \Gamma \vdash M_2 \Leftarrow A$, *and* $\Delta, \Omega'; \Gamma \vdash M \Leftarrow A$ *and*
$M_1 = [\![\theta_1]\!]M$ *and* $M_2 = [\![\theta_2]\!]M$
then there exists a contextual substitution θ_1^*, θ_2^*, *and a modal context* Ω^*, *such that* $(\Delta, \Omega); \Gamma \vdash M_1 \sqcup M_2 : A \Longrightarrow M/(\Omega^*, \theta_1^*, \theta_2^*)$ *and* $\theta_1^* \subseteq \theta_1$, $\theta_2^* \subseteq \theta_2$ *and* $\Omega^* \subseteq \Omega'$.

(2) *If* $\Delta, \Omega \vdash \theta_1 \Leftarrow \Omega'$ *and* $\Delta, \Omega \vdash \theta_2 \Leftarrow \Omega'$ *and* θ_1 *and* θ_2 *are incompatible and*
$\Delta, \Omega; \Gamma \vdash R_1 \Rightarrow P$, $\Delta; \Gamma \vdash R_2 \Rightarrow P$, *and* $\Omega', \Delta; \Gamma \vdash R \Rightarrow P$ *and*
$R_1 = [\![\theta_1]\!]R$ *and* $R_2 = [\![\theta_2]\!]R$
then there exists a contextual substitution θ_1^*, θ_2^*, *and a modal context* Ω^*, *such that* $(\Delta, \Omega); \Gamma \vdash R_1 \sqcup\!\!\underline{\sqcup} R_2 : P \Longrightarrow R/(\Omega^*, \theta_1^*, \theta_2^*)$ *and* $\theta_1^* \subseteq \theta_1$, $\theta_2^* \subseteq \theta_2$ *and* $\Omega^* \subseteq \Omega'$.

(3) *If* $\Delta, \Omega \vdash \theta_1 \Leftarrow \Omega'$ *and* $\Delta, \Omega \vdash \theta_2 \Leftarrow \Omega'$ *and* θ_1 *and* θ_2 *are incompatible and*
$(\Delta, \Omega); \Gamma \vdash S_1 > A \Rightarrow P$, $(\Delta, \Omega); \Gamma \vdash S_2 > A \Rightarrow P$, *and*
$(\Delta, \Omega'); \Gamma \vdash S > A \Rightarrow P$ *and* $S_1 = [\![\theta_1]\!]S$ *and* $S_2 = [\![\theta_2]\!]S$
then there exists a contextual substitution θ_1^*, θ_2^*, *and a modal context* Ω^*, *such that* $(\Delta, \Omega); \Gamma \vdash S_1 \sqcup\!\!\underline{\sqcup} S_2 : A \Longrightarrow S/(\Omega^*, \theta_1^*, \theta_2^*)$ *and* $\theta_1^* \subseteq \theta_1$, $\theta_2^* \subseteq \theta_2$ *and* $\Omega^* \subseteq \Omega'$.

PROOF. Simultaneous induction on the structure of M, R, and S. □

In the next section, we extend the soundness and completeness property to substitutions.

5.2 Most specific generalization of two contextual substitutions

Building on the previous algorithm for computing the most specific generalization of two linear higher-order patterns, we extend the algorithm to contextual substitutions. We begin by giving the judgments for computing the most specific linear generalization of two contextual substitutions.

$$\Delta, \Omega_1 \vdash \rho_1 \sqcup \rho_2 : \Omega_2 \Longrightarrow \rho/(\Omega, \theta_1, \theta_2) \qquad \rho \text{ is the mslg of } \rho_1 \text{ and } \rho_2$$

Intuitively, we will be able to obtain ρ_1 by composing θ_1 with ρ, and we yield ρ_2 by composing θ_2 with ρ. We assume ρ_1 and ρ_2 are well-typed, and have the domain Ω_2 and range (Ω_1, Δ).

We think of ρ_1 as the contextual substitution which is already in the index, while the contextual substitution ρ_2 is to be inserted. As a consequence, only ρ_1 will refer to the internal meta-variables in Ω_1, while ρ_2 only depends on the meta-variables in Δ. The result of the mslg are the contextual substitution θ_1 and θ_2, where $\Delta, \Omega_1 \vdash \theta_1 \Leftarrow \Omega$ and $\Delta, \Omega_1 \vdash \theta_2 \Leftarrow \Omega$. In other words, θ_1 (resp. θ_2) only replaces internal meta-variables in Ω.

First, we give the rules for computing the most specific linear generalization of two contextual substitutions.

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

$$\frac{\begin{array}{llll}(\Delta, \Omega_1) & \vdash \rho_1 \sqcup \rho_2 & : \Omega_2 & \Longrightarrow \rho \,/\, (\Omega_1', \theta_1, \theta_2) \\ (\Delta, \Omega_1); \Psi & \vdash R_1 \sqcup\!\sqcup R_2 & : P & \Longrightarrow R \,/\, (\Omega_2', \theta_1', \theta_2') \\ \Omega = (\Omega_1', \Omega_2') & \theta = (\theta_1, \theta_1') & \theta' = (\theta_2, \theta_2') & \end{array}}{(\Delta, \Omega_1) \vdash (\rho_1, \hat{\Psi}.R_1/i) \sqcup (\rho_2, \hat{\Psi}.R_2/i) : (\Omega_2, i{::}P[\Psi]) \Longrightarrow (\rho, \hat{\Psi}.R/i) \,/\, (\Omega, \theta, \theta')}$$

Note, that we are allowed to just combine the contextual substitutions θ_1 (θ_2 resp.) and θ_1' (θ_2' resp.) since we require that they refer to distinct meta-variables and all the meta-variables occur uniquely.

Similar to the compatibility of two terms, we can define the compatibility of two substitutions.

Definition 5.4 *Compatibility of contextual substitutions.*
If $(\Delta, \Omega_1) \vdash \rho_1 \sqcup \rho_2 : \Omega_2 \Longrightarrow \mathsf{id}_{\Omega_1}/(\Omega, \rho_1, \rho_2)$, then two contextual substitutions ρ_1 and ρ_2 are incompatible. Otherwise, we call ρ_1 and ρ_2 compatible.

As a consequence, if ρ_1 and ρ_2 are incompatible, then for any $\hat{\Psi}.R/i \in \rho_1$ and $\hat{\Psi}.R'/i \in \rho_2$, we know that R and R' are incompatible. Next, we prove soundness and completeness of this algorithm.

THEOREM 5.5 SOUNDNESS FOR MSLG OF SUBSTITUTIONS.
If $(\Delta, \Omega_1) \vdash \rho_1 \sqcup \rho_2 : \Omega_2 \Longrightarrow \rho/(\Omega, \theta_1, \theta_2)$ *and*
$(\Delta, \Omega_1) \vdash \rho_1 \Leftarrow \Omega_2$ *and* $(\Delta, \Omega_1) \vdash \rho_2 \Leftarrow \Omega_2$

then $(\Delta, \Omega) \vdash \rho \Leftarrow \Omega_2$, $(\Delta, \Omega_1) \vdash \theta_1 \Leftarrow \Omega$, $(\Delta, \Omega_1) \vdash \theta_2 \Leftarrow \Omega$, *and* $[\![\theta_1]\!]\rho = \rho_1$ *and* $[\![\theta_2]\!]\rho = \rho_2$

PROOF. Induction on the first derivation. □

THEOREM 5.6 COMPLETENESS FOR MSLG OF CONTEXTUAL SUBSTITUTIONS. *If* $(\Delta, \Omega) \vdash \theta_1 \Leftarrow \Omega'$ *and* $(\Delta, \Omega) \vdash \theta_2 \Leftarrow \Omega'$ *and* θ_1 *and* θ_2 *are incompatible and* $\rho_1 = [\![\theta_1]\!]\rho$ *and* $\rho_2 = [\![\theta_2]\!]\rho$ *then* $(\Delta, \Omega) \vdash \rho_1 \sqcup \rho_2 : \Omega_1 \Longrightarrow \rho/(\Omega^*, \theta_1^*, \theta_2^*)$ *such that* $\Omega^* \subseteq \Omega'$, $\theta_1^* \subseteq \theta_1$, $\theta_2^* \subseteq \theta_2$.

PROOF. Induction on the structure of ρ. □

5.3 Insertion into substitution tree

In this section we describe the final layer, namely the traversal of the substitution tree to insert a substitution δ. To insert the contextual substitution δ into a substitution tree, we need to traverse the index tree and compute at each node N with substitution ρ the mslg between ρ and δ. Recall the formal definition of substitution trees given earlier on page 18.

$$\begin{array}{lll} \text{Node N} & ::= & (\Omega \vdash \rho \twoheadrightarrow C) \\ \text{Children C} & ::= & [N, C] \mid \mathsf{nil} \end{array}$$

A tree is a node N with a contextual substitution ρ and a list of children C. If the list of children is empty, we have reached a leaf. In general, we will write $\Delta \vdash N : \Omega'$ where $N = (\Omega \vdash \rho \twoheadrightarrow C)$ which means that $(\Delta, \Omega) \vdash \rho \Leftarrow \Omega'$ and all the children N_i in C, $\Delta \vdash N_i : \Omega$.

To insert a new substitution δ into the substitution tree N, we proceed in two steps. First, we inspect all the children N_i of a parent node N, where $N_i = \Omega_i \vdash \rho_i \twoheadrightarrow C_i$ and check if ρ_i is compatible with δ. This compatibility check has three possible results:

(1) $(\Delta, \Omega_i) \vdash \rho_i \sqcup \delta : \Omega \Longrightarrow \mathsf{id}_\Omega/(\Omega', \rho_i, \delta)$:
This means ρ_i and δ are not compatible

(2) $(\Delta, \Omega_i) \vdash \rho_i \sqcup \delta : \Omega \Longrightarrow \rho_i/(\Omega_i, \mathsf{id}_{\Omega_i}, \theta_2)$
This means δ is an instance of ρ_i and we continue to insert θ_2 into the children C_i. In this case $[\![\theta_2]\!]\delta$ is equivalent to ρ_i and we call δ *fully compatible* with ρ_i.

(3) $(\Delta, \Omega_i) \vdash \rho_i \sqcup \delta : \Omega \Longrightarrow \rho'/(\Omega'', \theta_1, \theta_2)$
ρ_i and δ are compatible, but we need to replace node N_i with a new node $\Omega'' \vdash \rho' \twoheadrightarrow C'$ where C' contains two children, the child node $\Omega_i \vdash \theta_1 \twoheadrightarrow C_i$ and the child node $\cdot \vdash \theta_2 \twoheadrightarrow \mathsf{nil}$. In this case we call δ *partially compatible* with ρ_i.

The idea is to iterate through the list of children and collect all nodes which are fully compatible or at least partially compatible. If there is a fully compatible child N, we continue the insertion by considering node N. If there are no fully compatible children but a partially compatible node N we need to split N.

In general, there may be more than one fully compatible node and also more than one partially compatible node. In an implementation we need to resolve these choices. We employ a simple heuristic which orders the sets of fully and partially compatible nodes according to the number of elements the substitution contains

only counting those elements which are not mapped to themselves. A node which contains a n such elements has size n and is preferred over a node of size m if n is greater than n.

If no node is compatible, we simply create a new node with the substitution we intended to insert. To simplify the algorithms we only consider trees with at least one entry and a default identity substitution at the root. The substitution tree which contains as the only entry the term $\hat{\Psi}.R$ for example, is a tree with the default identity substitution $\hat{\Psi}.i_0[\mathsf{id}_\Psi]/i_0$ at the root and one child with the substitution $\hat{\Psi}.R/i_0$.

The filtering process to collect all nodes which are fully and partially compatible can be formalized by using the following judgment. We will distinguish between the fully compatible children, which we collect in V, and the partially compatible children, which we collect in S.

$$\begin{array}{lll} \text{Fully compatible children} & V & ::= \ \cdot \mid V, (N,\, \theta) \\ \text{Partially compatible children} & S & ::= \ \cdot \mid S,\, (N, \Omega \vdash \rho, \theta_1, \theta_2) \end{array}$$

Note that it is not quite enough to just identify the children nodes N which are fully compatible or partially compatible. Instead, we need to track more information. For example, if we identify a child node N where $N = \Omega' \vdash \rho_i \twoheadrightarrow C_i$ is fully compatible with δ, then this means that δ is an instance of ρ_i and there exists a contextual substitution θ such that $[\![\theta]\!]\rho_i = \delta$. Similarly, if we identify a child node N where $N = \Omega' \vdash \rho_i \twoheadrightarrow C_i$ is partially compatible with δ, then this means that mslg between ρ_i and δ is the contextual substitution ρ, and $\rho_i = [\![\theta_1]\!]\rho$ and $\delta = [\![\theta_2]\!]\rho$. Now we can define the following judgment:

$$\Delta \vdash C \sqcup \delta : \Omega \Longrightarrow (V, S)$$

Given some children C and a contextual substitution δ, where the domain of each child in C and of the contextual substitution δ is Ω, we can compute fully compatible children V and the partially compatible children S.

δ is a contextual substitution such that $\Delta \vdash \delta \Leftarrow \Omega$, and for all the children $C_i = (\Omega_i \vdash \rho_i \twoheadrightarrow C')$ in C, we have $\Delta, \Omega_i \vdash \rho_i \Leftarrow \Omega$. Then V and S describe all the children from C which are compatible with δ. We distinguish three cases.

$$\frac{}{\Delta \vdash \mathsf{nil} \sqcup \delta : \Omega \Longrightarrow (\cdot, \cdot)}$$

$$\frac{\Delta \vdash C \sqcup \delta : \Omega \Longrightarrow (V, S) \quad \Delta, \Omega_1 \vdash \rho_1 \sqcup \delta : \Omega \Longrightarrow \mathsf{id}_\Omega/(\Omega, \rho_1, \delta)}{\Delta \vdash [(\Omega_1 \vdash \rho_1 \twoheadrightarrow C_1), C] \sqcup \delta : \Omega \Longrightarrow (V, S)}\ NC$$

$$\frac{\Delta \vdash C \sqcup \delta : \Omega \Longrightarrow (V, S) \quad \Delta, \Omega_1 \vdash \rho_1 \sqcup \delta : \Omega \Longrightarrow \rho_1/(\Omega_1, \mathsf{id}_{\Omega_1}, \theta_2) \quad \rho_1 \neq \mathsf{id}_{\Omega_1}}{\Delta \vdash [(\Omega_1 \vdash \rho_1 \twoheadrightarrow C_1), C] \sqcup \delta : \Omega \Longrightarrow ((V,\ ((\Omega_1 \vdash \rho_1 \twoheadrightarrow C_1), \theta_2)),\ S)}\ FC$$

$$\frac{\Delta \vdash C \sqcup \delta : \Omega \Longrightarrow (V, S) \quad \Delta, \Omega_1 \vdash \rho_1 \sqcup \delta : \Omega \Longrightarrow \rho/(\Omega_2, \theta_1, \theta_2) \quad \rho \neq \rho_1 \neq \mathsf{id}_{\Omega_2}}{\Delta \vdash [(\Omega_1 \vdash \rho_1 \twoheadrightarrow C_1), C] \sqcup \delta : \Omega \Longrightarrow (V, (S\ ,\ ((\Omega_1 \vdash \rho_1 \twoheadrightarrow C_1), \Omega_2 \vdash \rho, \theta_1, \theta_2))}\ PC$$

The NC rule describes the case where the child C_i is not compatible with δ. Rule FC describes the case where δ is fully compatible with the child C_i and the rule PC describes the case where δ is partially compatible with C_i. Before

we describe the traversal of the substitution tree, we prove some straightforward soundness properties about these rules. The first lemma essentially states that δ is an instance of all nodes collected in V. Moreover, for every node N_i with substitution ρ_i in S, there exists a ρ' which is the most specific generalization of ρ_i and δ.

LEMMA 5.7 INSERTION OF SUBSTITUTION INTO TREE.
If $\Delta \vdash C \sqcup \delta : \Omega \Longrightarrow (V, S)$ *and* $\Delta \vdash \delta \Leftarrow \Omega$ *and for any* $(\Omega_i \vdash \rho_i \twoheadrightarrow C') \in C$ *with* $\Delta, \Omega_i \vdash \rho_i \Leftarrow \Omega$ *then*

(1) *for any* $(N_i, \theta_2) \in V$ *where* $N_i = (\Omega_i \vdash \rho_i \twoheadrightarrow C_i)$, *we have* $[\![\theta_2]\!]\rho_i = \delta$.
(2) *for any* $(N_i, \Omega' \vdash \rho', \theta_1, \theta_2) \in S$ *where* $N_i = (\Omega_i \vdash \rho_i \twoheadrightarrow C_i)$, *we have* $[\![\theta_2]\!]\rho' = \delta$ *and* $[\![\theta_1]\!]\rho' = \rho_i$.

PROOF. By structural induction on the first derivation and by using the previous soundness theorem for mslg of substitutions (theorem 5.5). □

Next, we show insertion of a substitution δ into a substitution tree N. The main judgment is the following:

$$\Delta \vdash N \sqcup \delta : \Omega \Longrightarrow N' \quad \text{Insert } \delta \text{ into the substitution tree } N$$

If N is a substitution tree and δ is not already in the tree, then N' will be the new substitution tree after inserting δ into N. We write $[N_i'/N_i]C$ to indicate that the i-th node N_i in the children C is replaced by the new node N_i'. Recall that the substitution δ which is inserted into the substitution tree N does only refer to meta-variables in Δ and does not contain any internal meta-variables. Therefore, a new leaf with substitution δ must have the following form: $\cdot \vdash \delta \twoheadrightarrow \mathsf{nil}$. Similarly, if we split the current node and create a new leaf $\cdot \vdash \theta_2 \twoheadrightarrow \mathsf{nil}$ (see rule "Split current node").

Create new leaf

$$\frac{\Delta \vdash C \sqcup \delta : \Omega \Longrightarrow (\cdot, \cdot)}{\Delta \vdash (\Omega' \vdash \rho \twoheadrightarrow C) \sqcup \delta : \Omega \Longrightarrow (\Omega' \vdash \rho \twoheadrightarrow (C, (\cdot \vdash \delta \twoheadrightarrow \mathsf{nil}))}$$

Recurse

$$\frac{\Delta \vdash C \sqcup \delta : \Omega \Longrightarrow (V, S) \quad N_i \in C \quad (N_i, \theta_2) \in V \quad \Delta \vdash N_i \sqcup \theta_2 \Longrightarrow N'}{\Delta \vdash (\Omega' \vdash \rho \twoheadrightarrow C) \sqcup \delta : \Omega \Longrightarrow (\Omega' \vdash \rho \twoheadrightarrow [N'/N_i]C}$$

Split current node

$$\frac{\Delta \vdash C \sqcup \delta : \Omega \Longrightarrow (\cdot, S) \quad N_i \in C \quad N_i = (\Omega_i \vdash \rho_i \twoheadrightarrow C_i) \quad (N_i, \Omega^* \vdash \rho, \theta_1, \theta_2) \in S}{\Delta \vdash (\Omega' \vdash \rho \twoheadrightarrow C) \sqcup \delta : \Omega \Longrightarrow (\Omega' \vdash \rho \twoheadrightarrow [(\Omega^* \vdash \rho \twoheadrightarrow ((\Omega_i \vdash \theta_1 \twoheadrightarrow C_i), (\cdot \vdash \theta_2 \twoheadrightarrow \mathsf{nil})))/N_i]C)}$$

The above rules always insert a substitution δ into the children C of a node $\Omega \vdash \rho \twoheadrightarrow C$. We start inserting a substitution $\hat{\Psi}.R/i_0$ into the empty substitution tree which contains the identity substitution $\hat{\Psi}.i_0[\mathsf{id}]/i_0$ and has an empty list of children. This allows us to treat insertion of a substitution δ into a substitution tree uniformly. After the first insertion, we obtain the substitution tree which contains the identity substitution $\hat{\Psi}.i_0[\mathsf{id}]/i_0$ and the child of this node contains the substitution $\hat{\Psi}.R/i_0$.

As we mentioned in the beginning, there may not be a unique substitution tree, and the order in which elements are inserted matters. More importantly, there are in fact two choices which need to be resolved when inserting a given element into a tree. The first one arises when we recursively insert a substitution into the substitution tree using the rule Recurse. This involves first computing a list of fully compatible children V. However, there may be in fact more than one fully compatible child, and hence we must choose one to recurse on. The second one arises in the split rule because there maybe be more than one partially compatible child in the set S. As mentioned earlier, these two choices must be resolved in an implementation. These choices are already present in the first-order setting, we can employ similar heuristics to resolve them. The higher-order framework we describe does not add any additional choices.

When inserting the contextual substitution θ_2 into the tree N_i, we possibly obtain a new tree N', and we must replace the old tree N_i with the new one. In practice, these updates are done destructively, and there is no need to explicitly return a tree N' and explicitly perform this replacement.

Next, we prove soundness of the insertion algorithm – we prove that in fact no matter what choice we make, we obtain a correct substitution tree. This leads us to the following soundness statement where we show that if we insert a substitution δ into the children C, then there exists a child $C_i = \Omega_i \vdash \rho_i \twoheadrightarrow C_i'$ in C and a path from ρ_i to ρ_n, where ρ_n is at a leaf such that $[\![\rho_n]\!][\![\rho_{n-1}]\!] \ldots \rho_i = \delta$.

Theorem 5.8 Soundness of insertion.
If $\Delta \vdash (\Omega' \vdash \rho' \twoheadrightarrow C) \sqcup \delta : \Omega \Longrightarrow (\Omega' \vdash \rho' \twoheadrightarrow C')$
then there exists a child $C_i = (\Omega_i \vdash \rho_i \twoheadrightarrow C')$ *and a path from* ρ_i *to* ρ_n *such that* $[\![\rho_n]\!][\![\rho_{n-1}]\!] \ldots \rho_i = \delta$.

Proof. By induction on the first derivation using the previous lemma 5.7. □

6. RETRIEVAL

In general, we can retrieve all terms from the index which satisfy a given property. This property may be finding all terms from the index which unify with a given term M or finding all terms N from the index, such that a given term M is an instance or variant of N. One advantage of substitution trees is that all these retrieval operations can be implemented with only small changes. A key challenge in the higher-order setting is to design efficient retrieval algorithms which will perform well in practice. We will show how to retrieve terms which are an instance of a given term in the index, and discuss how to modify this algorithm to retrieve all elements which are unifiable with a given term, and check for variant.

6.1 Instance checking for linear higher-order patterns

First, we present an instance checking algorithm for linear higher-order patterns. Instance checking or matching for higher-order terms was only recently proven to be decidable [Sterling 2007], but efficient algorithms do not exist for the general fragment. We therefore concentrate on linear higher-order patterns. The advantage will be that instance checking is similar to the first-order case, and easily can be modified to allow checking for unifiability of two terms.

We treat again internal meta-variables differently than global meta-variables. The principal judgment are as follows:

$$\begin{array}{lll}
\Delta_2; (\Delta_1, \Omega); \Gamma \vdash M_1 \doteq M_2 : A/(\theta, \rho) & & M_2 \text{ is an instance of } M_1 \\
\Delta_2; (\Delta_1, \Omega); \Gamma \vdash R_1 \doteqdot R_2 : P/(\theta, \rho) & & R_2 \text{ is an instance of } R_1 \\
\Delta_2; (\Delta_1, \Omega); \Gamma \vdash S_1 \doteqdot S_2 : A > P/(\theta, \rho) & & S_2 \text{ is an instance of } S_2
\end{array}$$

Again we assume that M_1 (R_1, S_1 resp.) must be well-typed in the modal context Δ, Ω and the bound variable context Γ. M_2 (R_2, S_2 resp.) are well-typed in the modal context Δ_2 and the bound variable context Γ. In other words M_1 only contains internal meta-variables and is stored in the index, while M_2 is given, and that the meta-variables occurring in M_1 are distinct from the meta-variables occurring in M_2. More formally this is stated as $(\Delta_1, \Omega); \Gamma \vdash M_1 \Leftarrow A$ and $\Delta_2; \Gamma \vdash M_2 \Leftarrow A$ and $\Delta = (\Delta_2, \Delta_1)$. ρ is the substitution for the internal meta-variables in Ω while θ is the substitution for some meta-variables in Δ_1. Moreover, we maintain that $[\![\theta, \rho]\!] M_1$ is syntactically equal to M_2. We will treat Ω as a linear context in the formal description given below. This will make it easier to prove the relationship between insertion and retrieval later on.

Normal terms

$$\frac{\Delta_2; (\Delta_1, \Omega); \Gamma, x{:}A_1 \vdash M_1 \doteq M_2 : A_2 \;/\; (\theta, \rho)}{\Delta_2; (\Delta_1, \Omega); \Gamma \vdash \lambda x.M_1 \doteq \lambda x.M_2 : A_1 \to A_2 \;/\; (\theta, \rho)}\ \mathsf{lam}$$

$$\frac{\Delta_2; (\Delta_1, \Omega); \Gamma \vdash R_1 \doteqdot R_2 : P \;/\; (\theta, \rho)}{\Delta_2; (\Delta_1, \Omega); \Gamma \vdash R_1 \doteq R_2 : P \;/\; (\theta, \rho)}\ \mathsf{coe}$$

Atomic terms

$$\frac{}{\Delta_2; (\Delta_1, i{::}P[\Gamma]); \Gamma \vdash i[\mathsf{id}_\Gamma] \doteqdot R : P \;/\; (\cdot, (\hat{\Gamma}.R/i))}\ \mathsf{meta\text{-}1}$$

$$\frac{u{::}P[\Gamma] \in \Delta_1}{\Delta_2; (\Delta_1, \cdot); \Gamma \vdash u[\pi_\Gamma] \doteqdot R : P \;/\; (\hat{\Gamma}.([\pi]^{-1}\, R/u), \cdot)}\ \mathsf{meta\text{-}2}$$

$$\frac{\Delta_2; (\Delta_1, \Omega); \Gamma \Vdash S_1 \doteqdot S_2 : A > P \;/\; (\theta, \rho) \qquad \text{where } \Sigma(H) = A \text{ or } \Gamma(H) = A}{\Delta_2; (\Delta_1, \Omega); \Gamma \vdash H \cdot S_1 \doteq H \cdot S_2 : P \;/\; (\theta, \rho)}\ \mathsf{head}$$

Spines

$$\frac{}{\Delta_2; (\Delta_1, \cdot); \Gamma \Vdash \mathsf{nil} \doteqdot \mathsf{nil} : P > P \;/\; (\cdot, \cdot)}\ \mathsf{Snil}$$

$$\frac{\begin{array}{c}\Delta_2; (\Delta_1, \Omega_1); \Gamma \vdash M_1 \doteqdot M_2 : A_1 \;/\; (\theta_1, \rho_1) \\ \Delta_2; (\Delta_1, \Omega_2); \Gamma \Vdash S_1 \doteqdot S_2 : A_2 > P \;/\; (\theta_2, \rho_2)\end{array}}{\Delta_2; (\Delta_1, \Omega_1, \Omega_2); \Gamma \Vdash (M_1; S_1) \doteqdot (M_2; S_2) : A_1 \to A_2 > P \;/\; ((\theta_1, \theta_2), (\rho_1, \rho_2))}\ \mathsf{Scons}$$

Note that we need not worry about capture in the rule for lambda expressions since meta-variables and bound variables are defined in different contexts. In the rule Scons, 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 a meta-variable $u[\pi]$ with another term R is simpler and more efficient than in the general higher-order pattern case. In particular, it does not

require a traversal of R (see rules meta-1 and meta-2).

Since the inverse of the substitution π can be computed directly and will be total, we know $[\pi]^{-1} R$ exists and can simply generate a substitution $\hat{\Gamma}.[\pi]^{-1} R/u$. The algorithm can be easily specialized to retrieve variants by requiring in the meta-2 rule that R must be $u[\pi]$. To check unifiability we need to add the dual rule to meta-2 where we unify R with an meta-variable $u[\pi]$. The only complication is that R may contain internal meta-variables which are defined later on the path in the substitution tree. Now we show soundness and completeness of the retrieval algorithm. We first show soundness and completeness of the instance algorithm for terms.

Theorem 6.1 Soundness of instance algorithm for terms.

(1) *If* $\Delta_2; (\Delta_1, \Omega); \Gamma \vdash M_1 \doteq M_2 : A/(\theta, \rho)$
where $(\Delta_1, \Omega); \Gamma \vdash M_1 \Leftarrow A$ *and* $\Delta_2; \Gamma \vdash M_2 \Leftarrow A$ *then* $[\![\theta, \rho]\!] M_1 = M_2$.

(2) *If* $\Delta_2; (\Delta_1, \Omega); \Gamma \vdash R_1 \doteqdot R_2 : P/(\theta, \rho)$
where $(\Delta_1, \Omega); \Gamma \vdash R_1 \Rightarrow P$ *and* $\Delta_2; \Gamma \vdash R_2 \Rightarrow P$ *then* $[\![\theta, \rho]\!] R_1 = R_2$.

(3) *If* $\Delta_2; (\Delta_1, \Omega); \Gamma \vdash S_1 \doteqdot S_2 > A \Rightarrow P/(\theta, \rho)$
where $(\Delta_1, \Omega); \Gamma \vdash S_1 > A \Rightarrow P$ *and* $\Delta_2; \Gamma \vdash S_2 > A \Rightarrow P$ *then* $[\![\theta, \rho]\!] S_1 = S_2$.

Proof. Simultaneous structural induction on the first derivation. □

For completeness we show that if the term M_2 is an instance of a linear term M then the given algorithm will succeed and return substitution θ^* for the meta-variables and a substitution ρ^* for the internal meta-variables occurring in M. This establishes a form of local completeness of the given retrieval algorithm. We will show later a global completeness theorem, which states that any time we compute the msgl of a term M_1 and M_2 to be M, then we can show that M_2 is in fact an instance of M. More generally, we show that any time we insert a substitution $\hat{\Gamma}.M_2/i_0$ we can also retrieve it.

Theorem 6.2 Completeness of instance algorithm for terms.

(1) *If* $(\Delta_1, \Omega); \Gamma \vdash M_1 \Leftarrow A$ *and* $\Delta_2; \Gamma \vdash M_2 \Leftarrow A$ *and*
$\Delta_2 \vdash \theta \Leftarrow \Delta_1$ *and* $\Delta_2 \vdash \rho \Leftarrow \Omega$ *and* $[\![\theta, \rho]\!] M_1 = M_2$ *then*
$\Delta_2; (\Delta_1, \Omega); \Gamma \vdash M_1 \doteq M_2 : A/(\theta^*, \rho)$ *where* $\theta^* \subseteq \theta$.

(2) *If* $(\Delta_1, \Omega); \Gamma \vdash R_1 \Rightarrow P$ *and* $\Delta_2; \Gamma \vdash R_2 \Rightarrow P$ *and*
$\Delta_2 \vdash \theta \Leftarrow \Delta_1$ *and* $\Delta_2 \vdash \rho \Leftarrow \Omega$ *and* $[\![\theta, \rho]\!] R_1 = R_2$ *then*
$\Delta_2; (\Delta_1, \Omega); \Gamma \vdash R_1 \doteqdot R_2 : P/(\theta^*, \rho)$ *where* $\theta^* \subseteq \theta$.

(3) *If* $(\Delta_1, \Omega); \Gamma \vdash S_1 > A \Rightarrow P$ *and* $\Delta_2; \Gamma \vdash S_2 > A \Rightarrow P$ *and*
$\Delta_2 \vdash \theta \Leftarrow \Delta_1$ *and* $\Delta_2 \vdash \rho \Leftarrow \Omega$ *and* $[\![\theta, \rho]\!] S_1 = S_2$ *then*
$\Delta_2; (\Delta_1, \Omega); \Gamma \vdash S_1 \doteqdot S_2 : A > P/(\theta^*, \rho)$ *where* $\theta^* \subseteq \theta$.

Proof. Simultaneous structural induction on the first typing derivation. □

6.2 Instance checking for contextual substitutions

The instance algorithm for terms can be straightforwardly extended to instances of substitutions. We define the following judgment for it:

$$\Delta_2; (\Delta_1, \Omega) \vdash \rho_1 \doteq \rho_2 : \Omega'/(\theta, \rho) \;\; \rho_2 \text{ is an instance of } \rho_1$$

We assume that ρ_1 is a contextual substitution from a modal context Ω' to a modal context Δ, Ω, and and ρ_2 is a contextual substitution from Ω' to the modal context Δ_2. Our goal is to check whether ρ_2 is an instance of ρ_1. The result of this is the contextual substitution ρ for the meta-variables in Ω and the contextual substitution θ for the meta-variables in Δ such that $[\![\theta, \sigma]\!]\rho_1$ is syntactically equal to ρ_2. Again we enforce the linearity criteria for internal meta-variables in Ω but leave it implicit for the meta-variables in Δ_1.

$$\frac{}{\Delta_2; (\Delta_1, \cdot) \vdash \cdot \doteq \cdot : \cdot/(\cdot, \cdot)}$$

$$\frac{\Delta_2; (\Delta_1, \Omega_1'') \vdash \rho_1 \doteq \rho_2 : \Omega_2/(\theta, \rho) \quad \Delta_2; (\Delta_1, \Omega_1'); \Gamma \vdash R_1 \doteq R_2 : P/(\theta', \rho')}{\Delta_2; (\Delta_1, \Omega_1', \Omega_2'') \vdash (\rho_1, \hat{\Psi}.R_1/i) \doteq (\rho_2, \hat{\Psi}.R_2/i) : (\Omega_2, i{::}P[\Gamma]))/((\theta, \theta'), (\rho, \rho')}$$

Next, we show soundness of retrieval for substitutions.

Theorem 6.3 Soundness of retrieval for substitutions.
If $\Delta_2; (\Delta_1, \Omega) \vdash \rho_1 \doteq \rho_2 : \Omega'/(\theta, \rho)$ *and* $(\Delta_1, \Omega) \vdash \rho_1 \Leftarrow \Omega'$ *and* $\Delta_2 \vdash \rho_2 \Leftarrow \Omega'$ *and* $(\Delta_1, \Delta_2) = \Delta$ *and all the variables in* Ω, Δ_1 *and* Δ_2 *are distinct then* $[\![\theta, \rho]\!]\rho_1 = \rho_2$.

Proof. Structural induction on the first derivation and using previous theorem 6.1. □

Finally, we show the global completeness of the mslg and instance algorithm which relates insertion and retrieval. We show that if the mslg of object M_1 and M_2 returns the contextual substitutions θ_1 and θ_2 together with the mslg M, then in fact the retrieval algorithm shows that M_1 is an instance of M under θ_1 and M_2 is an instance of M under θ_2. This guarantees that any time we insert a term M_2 we can in fact retrieve it. We assume here that the set of meta-variables in M_1 is distinct from the set of meta-variables in M_2 which simplifies this proof slightly, since this guarantees that the mslg M only refers to meta-variables in Ω'.

Theorem 6.4 Interaction between mslg and instance algorithm.

(1) *If* $(\Delta_1, \Omega); \Gamma \vdash M_1 \Leftarrow A$ *and* $\Delta_2; \Gamma \vdash M_2 \Leftarrow A$ *and* $(\Delta_2, \Delta_1), \Omega; \Gamma \vdash M_1 \sqcup M_2 : A \Longrightarrow M/(\Omega', \rho_1, \rho_2)$ *then* $(\Delta_1; \Omega'; \Gamma \vdash M \doteq M_1 : A/(\cdot, \rho_1)$ *and* $\Delta_2; \Omega'; \Gamma \vdash M \doteq M_2 : A/(\cdot, \rho_2)$.

(2) *If* $(\Delta_1, \Omega); \Gamma \vdash R_1 \Rightarrow P$ *and* $\Delta_2; \Gamma \vdash R_2 \Rightarrow P$ *and* $(\Delta_2, \Delta_1), \Omega; \Gamma \vdash R_1 \underline{\sqcup} R_2 : P \Longrightarrow R/(\Omega', \rho_1, \rho_2)$ *then* $\Delta_1; \Omega'; \Gamma \vdash R \doteqdot R_1 : P/(\cdot, \rho_1)$ *and* $\Delta_2; \Omega'; \Gamma \vdash R \doteqdot R_2 : P/(\cdot, \rho_2)$.

(3) *If* $(\Delta_1, \Omega); \Gamma \vdash S_1 > A \Rightarrow P$ *and* $\Delta_2; \Gamma \vdash S_2 > A \Rightarrow P$ *and* $(\Delta_2, \Delta_1), \Omega; \Gamma \vdash S_1 \underline{\sqcup} S_2 : A > P \Longrightarrow S/(\Omega', \rho_1, \rho_2)$ *then* $\Delta_1; \Omega'; \Gamma \vdash S \doteq S_1 : A > P/(\cdot, \rho_1)$ *and* $\Delta_2; \Omega'; \Gamma \vdash S \doteq S_2 : A > P/(\cdot, \rho_2)$.

Proof. Simultaneous structural induction on the first derivation. □

Theorem 6.5 Insertion and retrieval for substitutions.
If $\Delta_2, \Delta_1, \Omega \vdash \rho_1 \sqcup \rho_2 : \Omega' \Longrightarrow \rho/(\Omega'', \theta_1, \theta_2)$ *and* $(\Delta_1, \Omega) \vdash \rho_1 \Leftarrow \Omega'$ *and* $\Delta_2 \vdash \rho_2 \Leftarrow \Omega'$ *and* $(\Delta_1, \Delta_2) = \Delta$ *then* $\Delta_1; \Omega'' \vdash \rho \doteq \rho_1 : \Omega'/(\cdot, \theta_1)$ *and* $\Delta_2; \Omega'' \vdash \rho \doteq \rho_2 : \Omega'/(\cdot, \theta_2)$

Proof. Structural induction on the first derivation and use of lemma 6.4. □

Next, we show how to traverse the tree, to find a path $[\![\rho_n]\!][\![\rho_{n-1}]\!]\ldots\rho_1$ such that ρ_2 is an instance of it and return a contextual substitution θ such that $[\![\theta]\!][\![\rho_n]\!][\![\rho_{n-1}]\!]\ldots\rho_1 = \rho_2$. Traversal of the tree is straightforward.

$$\frac{\Delta, \Omega \vdash \rho \doteq \rho_2 : \Omega'/(\theta', \rho') \quad \Delta \vdash C \doteq \rho' : \Omega/\theta}{\Delta \vdash [(\Omega \vdash \rho \twoheadrightarrow C), C'] \doteq \rho_2 : \Omega'/(\theta', \theta)}$$

$$\frac{\begin{array}{r}\text{there is no derivation such that} \Delta, \Omega \vdash \rho \doteq \rho_2 : \Omega'/(\theta', \rho') \\ \Delta \vdash C' \doteq \rho : \Omega/\theta\end{array}}{\Delta \vdash [(\Omega \vdash \rho \twoheadrightarrow C), C'] \doteq \rho_2 : \Omega'/\theta}$$

THEOREM 6.6 SOUNDNESS OF RETRIEVAL.
If $\Delta \vdash C \doteq \rho' : \Omega'/\theta$ then there exists a child C_i with substitution ρ_i in C such that the path $[\![\theta]\!][\![\rho_n]\!][\![\rho_{n-1}]\!]\ldots[\![\rho_i]\!] = \rho'$.

PROOF. By structural induction on the first derivation and use of lemma 6.3. □

Finally, we show that if we insert ρ into a substitution tree and obtain a new tree, then we are able to retrieve ρ from it.

THEOREM 6.7 INTERACTION BETWEEN INSERTION AND RETRIEVAL.
If $\Delta \vdash (\Omega \vdash \rho \twoheadrightarrow C) \sqcup \rho_2 : \Omega \Longrightarrow (\Omega \vdash \rho \twoheadrightarrow C')$ then $\Delta \vdash C' \doteq \rho_2/\mathsf{id}_\Delta$.

PROOF. Structural induction on the derivation using theorem 6.5. □

7. EXTENSION TO DEPENDENTLY TYPED TERMS

Substitution trees are especially suited for indexing dependently typed terms, since they provide more flexibility than indexing techniques such as discrimination tries or path indexing techniques which only allow us to share common prefixes. To illustrate this point, we define a data-structure for lists consisting of characters and we keep track of the size of the list by using dependent types.

$$\begin{array}{ll}\mathsf{char} : \mathsf{type}. & \mathsf{list} : \mathsf{int} \rightarrow \mathsf{type}. \\ \mathsf{a} : \mathsf{char}. & \mathsf{nil} : \mathsf{list}\ 0. \\ \mathsf{b} : \mathsf{char}. & \mathsf{cons} : \Pi n{:}\mathsf{int}.\mathsf{char} \rightarrow \mathsf{list}\ n \rightarrow \mathsf{list}\ (n+1). \\ \mathsf{test} : \Pi n{:}\mathsf{int}.\mathsf{list}\ n \rightarrow \mathsf{type}. & \end{array}$$

The size of lists is an explicit argument to the predicate test. Hence test takes in two arguments, the first one is the size of the list and the second one is the actual list. The list constructor cons takes in three arguments. The first one denotes the size of the list, the second argument denotes the head and the third one denotes the tail. To illustrate, we give a few examples. We use gray color for the index arguments.

$$\begin{array}{l}\mathsf{test}\ 4\ (\mathsf{cons}\ 3\ a\ (\mathsf{cons}\ 2\ a\ (\mathsf{cons}\ 1\ a\ (\mathsf{cons}\ 0\ b\ \mathsf{nil})))) \\ \mathsf{test}\ 5\ (\mathsf{cons}\ 4\ a\ (\mathsf{cons}\ 3\ a\ (\mathsf{cons}\ 2\ a\ (\mathsf{cons}\ 1\ a\ (\mathsf{cons}\ 0\ b\ \mathsf{nil}))))) \\ \mathsf{test}\ 6\ (\mathsf{cons}\ 5\ a\ (\mathsf{cons}\ 4\ a\ (\mathsf{cons}\ 3\ a\ (\mathsf{cons}\ 2\ b\ (\mathsf{cons}\ 1\ a\ (\mathsf{cons}\ 0\ b\ \mathsf{nil}))))))\end{array}$$

If we use non-adaptive indexing techniques such as discrimination tries, we process the term from left to right and we will be able to share common prefixes. In

the given example, such a technique discriminates on the first argument, which denotes the size of the list and leads to no sharing between the second argument. The substitution tree on the other hand allows us to share the structure of the second argument. The most specific linear generalization in this example is

$$\mathsf{test}\ i_1[\mathsf{id}]\ (\mathsf{cons}\ i_2[\mathsf{id}]\ a\ (\mathsf{cons}\ i_3[\mathsf{id}]\ a\ (\mathsf{cons}\ i_4[\mathsf{id}]\ a\ (\mathsf{cons}\ i_5[\mathsf{id}]\ i_6[\mathsf{id}]\ \mathsf{nil})))).$$

This allows us to skip over the first argument denoting the size and indexing on the second argument, the actual list. It has been sometimes argued that it is possible to retain the flexibility in non-adaptive indexing techniques by reordering the arguments to test. However, this only works easily in an untyped setting and it is not clear how to maintain typing invariants in a dependently typed setting if we allow arbitrary reordering of arguments. Hence higher-order substitution trees offer an adaptive compact indexing data-structure while maintaining typing invariants.

However, there are unique challenges of designing and implementing higher-order substitution trees for dependently typed terms. For example, transforming dependently typed terms into linear higher-order patterns may result in ill-typed terms – or better linear higher-order patterns are only well-typed modulo variable definitions. Fortunately, linear higher-order patterns are still approximately well-typed. It were these complex issues which lead us to carefully formalize substitution trees in the simply typed setting.

We may think of linear terms as a representation which is only used internally, and all linear terms are well-typed modulo variable definitions. Then we can show that simple types (e.g. types where dependencies have been erased) are preserved in substitution trees, and all intermediate variables introduced are only used within this data-structure, but do not leak outside. As a consequence, we will always obtain a dependently typed term after composing the contextual substitutions in one branch of the substitution tree and solving the variable definitions.

8. IMPLEMENTATION AND EXPERIMENTAL RESULTS

We have implemented substitution trees to manage the table in the tabled higher-order logic programming engine [Pientka 2005] to permit lookup and possible insertion of terms to be performed in a single pass as part of the Twelf system [Pfenning and Schürmann 1999]. The Twelf system is an implementation of the logical framework LF together with a logic programming engine using the typed functional language SML of New Jersey 110.0.3.

In the tabled higher-order logic programming engine we store intermediate subgoals in a memoization table which is implemented as a substitution tree. Each tabled predicate has its own substitution tree. Our implementation is a high-level implementation which relies on the existing data-structure for terms and types to smoothly integrate it into the existing framework. In our implementation, de Bruijn indices are used to implement bound variables and meta-variables, and terms are implemented using spine notation as described in this paper.

The substitution tree itself is implemented as a linked tree, where a node contains as data a substitution together with the context of meta-variables occurring in the range of this substitution and a list of pointers to other substitution trees which constitute its children. The substitution at each node is implemented as a set using

C. Okasaki's implementation of red-black trees. This allows for destructive updates within the tree.

In the implementation we use two different de Bruijn indices – one for bound variables and global meta-variables (from Δ), and one for internal meta-variables (from Ω). To distinguish between bound variables and global meta-variables we in addition carry a marker. Internal existential variables i will be instantiated at a later point as we traverse the tree. Global meta-variables on the other hand are only subject to instantiation if we check whether the current subgoal is an instance of one in the table.

At the leafs of the substitution tree, we store linear residual equations, context Γ, the meta-variables occurring in the residual equations and in Γ, as well as a pointer to the answer list. We maintain that terms are kept in $\beta\eta$-normal form, as also described in this paper. Before inserting a term in the index we normalize and linearize it to obtain a standard form. In the subsequent discussion, we compare the performance of the tabled engine with and without indexing.

8.1 Experimental results

In this section, we discuss examples from three different applications which use the tabled logic programming engine in Twelf. Here we focus on an evaluation of the indexing technique. For a more in-depth discussion of tabling we refer the reader to [Pientka 2002; 2005]. All experiments were 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 instance checking, which allows us to check if a table entry subsumes the given goal, we did not observe substantial performance improvements. This is mostly due to the fact that the surrounding tabling engine can maintain stronger invariants if we support only variant checking and eliminates the need to perform a lookup on the table to retrieve answers. A similar observation has been made for tabling in the first-order logic programming engine XSB [Ramakrishnan et al. 1999]. Potentially instance checking becomes more important in theorem proving, as the experience in the first-order setting shows. We compare our tabling engine without term indexing and without linearization to the one which supports substitution tree indexing.

8.2 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 [Warren 1999]). In this example, we adapted ideas from [Warren 1999] to implement a parser for first-order formulas using higher-order abstract syntax. The simplest way to implement left and right associativity properties of implications, conjunction and disjunction is to mix right and left recursive program clauses. Clauses for conjunction and disjunction are left recursive, while the program clause for implication is right recursive. Such an implementation of the grammar is straightforward mirroring the defined properties such as left and right associativity and precedence ordering.

#tok	noindex	index	reduction in time	improvement factor
20	0.13	0.07	46%	1.85
58	2.61	1.25	52%	2.08
117	10.44	5.12	51%	2.03
178	32.20	13.56	58%	2.37
235	75.57	26.08	66%	2.90

The first column denotes the number of tokens which are parsed. This example illustrates that indexing can lead to improvements by over a factor of 2.90. 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.

8.3 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 [Davies and Pfenning 2000]. We use an implementation of the bi-directional type-checker in Twelf by F. 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	reduction time	improvement factor
sub1	3.19	0.46	86%	6.93
sub2	4.22	0.55	87%	7.63
sub3	5.87	0.63	89%	9.32
mult	7.78	0.89	89%	8.74
square1	9.08	0.99	89%	9.17
square2	9.02	0.98	89%	9.20

Not provable

example	noindex	index	reduction time	improvement factor
multNP1	2.38	0.38	84%	6.26
multNP2	2.66	0.51	81%	5.22
plusNP1	1.02	0.24	76%	4.25
plusNP2	6.48	0.85	87%	7.62
squareNP1	9.29	1.09	88%	8.52
squareNP2	9.26	1.18	87%	7.85

example	All answers noindex	index	reduction time	improvement factor
sub1	6.88	0.71	90%	9.69
sub2	3.72	0.48	87%	7.75
sub3	4.99	0.59	88%	8.46
mult	9.06	0.98	89%	9.24
square1	10.37	1.11	89%	9.34
square2	10.30	1.08	90%	9.54

As the results demonstrate indexing leads to substantial improvements by over a factor of 9. Table sizes are around 500 entries.

8.4 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	reduction time	improvement factor
mult1	10.86	6.26	57%	1.73
mult2	39.13	18.31	47%	2.14
addminus1	54.31	14.42	73%	3.77
addminus2	57.34	15.66	73%	3.66
addminus3	55.23	25.45	54%	2.17
addminus4	144.73	56.63	61%	2.55
minusadd1	1339.16	462.83	65%	2.89

As the results demonstrate, performance is improved by up to 3.77. 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.

9. RELATED WORK

We have presented a higher-order term indexing technique, called higher-order substitution trees. We only know of two other prior attempts to design and implement a higher-order term indexing technique. L. Klein [Klein 1997] 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 linear generalization of two higher-order terms does not exist in general, he suggests to maximally decompose a term into its atomic subterms. This approach results 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 need to be performed eagerly, potentially degrading the performance.

Necula and Rahul briefly discuss the use of automata driven indexing for higher-order terms in [Necula and Rahul 2001]. Their approach is to ignore all higher-order

features when maintaining the index, and return an imperfect set of candidates. Then full higher-order unification on the original terms is used to filter out the ones which are in fact unifiable in a post-processing step. They also implemented 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. The presented indexing technique is designed as a perfect filter for linear higher-order patterns. For objects which are not linear higher-order patterns, we solve variable definitions via higher-order unification, but avoid calling higher-order unification on the original term.

More recently, Theiss and Benzmüller explore higher-order term indexing in their implementation of Leo-II, a higher-order resolution theorem prover [Theiss and Benzmüller 2006]. Their approach is based on coordinate and path indexing [Stickel 1989]. The work mostly focuses on efficient data-structure of lambda-terms based on de Bruijn indices together with a path-indexing structure. Terms are not represented using spines, which means that there is no easy access to the head of a lambda-term. Moreover, there is no notion of meta-variables as closures. To implement path-indexing for lambda-terms, the authors keep track of the scope of lambda-binders using a scope number. Their indexing structure is partly designed to speed-up β-reduction, and the impact on theorem proving is not yet fully known.

Higher-order substitution trees provide a very flexible term indexing structure. As mentioned earlier, they are particular suited for dependently-typed systems, since there is no sharing of prefixes. However, in general there may be multiple ways to insert a term and no optimal substitution trees exist, and these choices must be resolved in an implementation. One may consider obtaining an even more compact tree by computing the generalization of (`and` (`not A`) (`not B`)) and (`or` (`not A`) (`not B`)) as $i[\texttt{not A}/\texttt{x}, \texttt{not B}/\texttt{y}]$ where we can obtain the first term by instantiating i with (`and x y`) and the second one by instantiating i with (`or x y`). A similar idea is employed in the first-order setting in context trees [Ganzinger et al. 2004]. In the higher-order setting however to allow this extension is non-trivial since the term $i[\texttt{not A}/\texttt{x}, \texttt{not B}/\texttt{y}]$ is outside the decidable pattern fragment. For example, if we match (`imp` (`not A`)(`not B`)) against $i[\texttt{not A}/\texttt{x}, \texttt{not B}/\texttt{y}]$, then there are four possible instantiations for i: (`imp x y`), (`imp` (`not A`) `y`), (`imp x` (`not B`)), (`imp` (`not A`) (`not B`)). This nondeterminism would significantly complicate insertion and retrieval.

10. CONCLUSION

We have presented the theoretical foundation for higher-order substitution trees based on linear higher-order patterns, developed algorithms for computing most specific generalizations of two linear higher-order patterns, and proved the correctness of inserting and retrieving elements from a substitution tree. The presented framework can serve as a general foundation for higher-order term indexing in many higher-order logic programming systems such as λProlog [Nadathur and Mitchell 1999], and higher-order theorem provers such as Bedwyr [Baelde et al. 2007], Isabelle [Paulson 1986], or Leo-II [Benzmüller et al. 2008].

We have implemented substitution trees within the Twelf system [Pfenning and Schürmann 1999] as part of the higher-order tabled logic programming engine. As

our experimental result show, this lead to an improvement of up to a factor of 10 for the tabling engine. In the tabling engine, the table is a dynamically built index, i.e. when evaluating a query we store intermediate goals encountered during proof search. We have also used our substitution tree implementation to index higher-order logic programs (see [Sarkar et al. 2005]). The use of indexing in this setting was motivated by the need to easily retrieve all clauses which unify with a current goal. This was difficult to incorporate in the existing previous implementation, and substitution trees provided a flexible way to deal with this issue. For this we build the index statically, observing the order of the clauses from the program.

Our experimental results and implementation demonstrate that indexing in the higher-order setting is feasible and beneficial in practice. However, much work remains to be done on the implementations of higher-order logic programming systems and higher-order theorem provers to explore the full impact of these techniques. In the setting of substitution tree indexing for example, we can compute an optimal substitution tree via unification factoring [Dawson et al. 1995] for a static set of terms to get the best sharing among clause heads. Hence an important question is how these results carry over to higher-order case. Another important observation is that clause heads typically do not use deeply nested terms, but are fairly shallow. This results in shallow substitution trees which are wide but not very deep. In the future, we plan to explore and optimize substitution tree indexing for indexing higher-order logic programming clauses.

Acknowledgments

I would like to thank the reviewers for their thorough and extensive comments which helped improve this work substantially.

REFERENCES

Abadi, M., Cardelli, L., Curien, P.-L., and Lévy, J.-J. 1990. Explicit substitutions. In *17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'90)*. ACM Press, 31–46.

Baelde, D., Gacek, A., Miller, D., Nadathur, G., and Tiu, A. 2007. The Bedwyr system for model checking over syntactic expressions. In *21st Conference on Automated Deduction*, F. Pfenning, Ed. Lecture Notes in Artificial Intelligence (LNAI 4603). Springer, 391–397.

Benzmüller, C., Theiss, F., Paulson, L., and Fietzke, A. 2008. Leo-II - A Cooperative Automatic Theorem Prover for Higher-Order Logic (System Description). In *4th International Joint Conference on Automated Reasoning (IJCAR'08)*. Lecture Notes in Artificial Intelligence (LNAI 5195). Springer, 162–170.

Cervesato, I. and Pfenning, F. 2003. A linear spine calculus. *Journal of Logic and Computation 13,* 5, 639–688.

Chen, T., Ramakrishnan, I. V., and Ramesh, R. 1994. Multistage indexing for speeding prolog executions. *Software – Practice and Experience 24,* 12, 1097–1119.

Davies, R. and Pfenning, F. 2000. Intersection types and computational effects. In *5th International Conference on Functional Programming (ICFP'00)*. ACM Press, 198–208.

Dawson, S., Ramakrishnan, C. R., Ramakrishnan, I. V., and Swift, T. 1995. Optimizing clause resolution: Beyond unification factoring. In *International Logic Programming Symposium*, MIT Press. 194–208.

Dawson, S., Ramakrishnan, C. R., Skiena, S., and Swift, T. 1995. Principles and practice of unification factoring. *ACM Transactions on Programming Languages and Systems 18,* 6, 528–563.

Dawson, S., Ramakrishnan, C. R., and V.Ramakrishnan, I. 1995. Design and implementation of jump tables for fast indexing of logic programs. In *International Symposium on Programming Language Implementation and Logic Programming (PLILP'95)*. Lecture Notes in Computer Science (LNCS 982). Springer, 133–150.

Dowek, G., Hardin, T., and Kirchner, C. 1995. Higher-order unification via explicit substitutions. In *10th Annual Symposium on Logic in Computer Science*, D. Kozen, Ed. IEEE Computer Society Press, 366–374.

Ganzinger, H., Nieuwenhuis, R., and Nivela, P. 2004. Fast term indexing with coded context trees. *Journal of Automated Reasoning 32,* 2, 103–120.

Graf, P. 1995a. Substitution tree indexing. In *6th International Conference on Rewriting Techniques and Applications, Kaiserslautern, Germany*. Lecture Notes in Computer Science (LNCS 914). Springer-Verlag, 117–131.

Graf, P. 1995b. *Term Indexing*. Lecture Notes in Artificial Intelligence (LNAI 1053). Springer-Verlag.

Hillenbrand, T. 2003. Citius altius fortius: lessons learned from the theorem prover Waldmeister (invited paper). In *Proceedings of the 4th International Workshop on First-Order Theorem Proving*, I. Dahn and L. Vigneron, Eds. Electronic Notes in Theoretical Computer Science, vol. 86.1. Elsevier Science.

Klein, L. 1997. Indexing für Terme höherer Stufe. Diplomarbeit, FB 14, Universität des Saarlandes, Saarbrücken, Germany.

McCune, W. 1992. Experiments with discrimination-tree indexing and path indexing for term retrieval. *Journal of Automated Reasoning 9,* 2, 147–167.

Miller, D. 1991a. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation 1,* 4, 497–536.

Miller, D. 1991b. Unification of simply typed lambda-terms as logic programming. In *8th International Logic Programming Conference*. MIT Press, 255–269.

Nadathur, G. and Mitchell, D. J. 1999. System description: Teyjus – a compiler and abstract machine based implementation of Lambda Prolog. In *16th International Conference on Automated Deduction (CADE'99)*, H. Ganzinger, Ed. Lecture Notes in Computer Science (LNCS 1632). Springer, 287–291.

Nanevski, A., Pfenning, F., and Pientka, B. 2008. Contextual modal type theory. *ACM Transactions on Computational Logic 9,* 3.

Necula, G. and Rahul, S. 2001. Oracle-based checking of untrusted software. In *28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'01)*. ACM Press, 142–154.

Paulson, L. C. 1986. Natural deduction as higher-order resolution. *Journal of Logic Programming 3*, 237–258.

Pfenning, F. 1991. Unification and anti-unification in the Calculus of Constructions. In *6th Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press, 74–85.

Pfenning, F. and Schürmann, C. 1999. System description: Twelf — a meta-logical framework for deductive systems. In *16th International Conference on Automated Deduction (CADE'99*, H. Ganzinger, Ed. Lecture Notes in Artificial Intelligence (LNAI 1632). Springer, 202–206.

Pientka, B. 2002. A proof-theoretic foundation for tabled higher-order logic programming. In *18th International Conference on Logic Programming*, P. Stuckey, Ed. Lecture Notes in Computer Science (LNCS 2401). Springer, 271 –286.

Pientka, B. 2003a. Higher-order substitution tree indexing. In *19th International Conference on Logic Programming*, C. Palamidessi, Ed. Lecture Notes in Computer Science (LNCS 2916). Springer, 377–391.

Pientka, B. 2003b. Tabled higher-order logic programming. Ph.D. thesis, Department of Computer Science, Carnegie Mellon University. CMU-CS-03-185.

Pientka, B. 2005. Tabling for higher-order logic programming. In *20th International Conference on Automated Deduction (CADE'05)*, R. Nieuwenhuis, Ed. Lecture Notes in Computer Science, (LNCS 3632). Springer, 54–68.

Pientka, B. and Pfenning, F. 2003. Optimizing higher-order pattern unification. In *19th International Conference on Automated Deduction (CADE'03), Miami, USA*, F. Baader, Ed. Lecture Notes in Artificial Intelligence (LNAI 2741). Springer, 473–487.

Ramakrishnan, I., Rao, P., Sagonas, K., Swift, T., and Warren, D. 1995. Efficient tabling mechanisms for logic programs. In *12th International Conference on Logic Programming*, L. .Sterling, Ed. MIT Press, 697–711.

Ramakrishnan, I. V., Rao, P., Sagonas, K., Swift, T., and Warren, D. 1999. Efficient access mechanisms for tabled logic programs. *Journal of Logic Programming 38,* 1 (Jan), 31–54.

Ramakrishnan, I. V., Sekar, R., and Voronkov, A. 2001. Term indexing. In *Handbook of Automated Reasoning*, A. Robinson and A. Voronkov, Eds. Vol. 2. Elsevier Science Publishers B.V., 1853–1962.

Ramesh, R., Ramakrishnan, I. V., and Warren, D. S. 1990. Automata-driven indexing of Prolog clauses. In *17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'90)*. ACM Press, 281–291.

Riazanov, A. and Voronkov, A. 2002. The design and implementation of Vampire. *AI Communications 15,* 2, 91–110.

Sarkar, S., Pientka, B., and Crary, K. 2005. Small proof witnesses for LF. In *21st International Conference on Logic Programming*, M. Gabbrielli and G. Gupta, Eds. Lecture Notes in Computer Science (LNCS 3668). Springer-Verlag, 387–401.

Sterling, C. 2007. Higher-order matching, games and automata. In *22nd Annual IEEE Symposium on Logic in Computer Science (LICS'07)*. IEEE Computer Society Press, 326–335.

Stickel, M. E. 1989. The path-indexing method for indexing terms. Tech. Rep. 473, AI Center, SRI International, 333 Ravenswood Ave., Menlo Park, CA 94025. Oct.

Theiss, F. and Benzmüller, C. 2006. Term Indexing for the LEO-II Prover. In *6th International Workshop on the Implementation of Logics*.

Warren, D. S. 1999. *Programming in tabled logic programming*. draft available from http://www.cs.sunysb.edu/w̃arren/xsbbook/book.html.

Watkins, K., Cervesato, I., Pfenning, F., and Walker, D. 2002. A concurrent logical framework I: Judgments and properties. Tech. Rep. CMU-CS-02-101, Department of Computer Science, Carnegie Mellon University.

Received August 2007 ; revised June 2008; accepted October 2008;