1 Introduction

We study a generalization of first-order unification, called semi-unification, with two goals in mind:

1. Type-check functional programs relative to an improved polymorphic type discipline.
2. Decide the typability of terms in a restricted form of the polymorphic $\lambda$-calculus.

Our motivation to study semi-unification is therefore both pragmatic (in the case of 1) and theoretical (in the case of 2). The same unification problem is encountered in other areas of computer science (e.g., semi-unifiability can be used as a check for the non-termination of a rewrite rule) and in logic (e.g., if semi-unifiability is decidable then a strengthened version of a conjecture due to Kreisel is true).

1.1 The need for an improved polymorphic type discipline

A polymorphic type system, which can be used to infer types of untyped or partially typed programs, was first implemented for the language ML ([6,17]). It has proved to be very useful in programming practice and many of its aspects have been incorporated in other languages (e.g., Miranda, see [24]). It shares with Algol 68 properties of compile-time type-checking, strong typing and higher-order functions, but it also allows the definition of polymorphic functions, i.e., functions that work uniformly on arguments of different types. This has the obvious advantage of avoiding the irritating need to duplicate similar code at different times.

There are nevertheless two important limitations in the original ML type discipline. One limitation forces all the occurrences of the same $\lambda$-bound variable to have the same type. By contrast, although let $x = N$ in $M$ is considered syntactic sugar for $(\lambda x.M)N$, the ML type-checker allows the occurrences of a let-bound variable to have different types. The consequence is an occasional anomaly, illustrated by the following example (in Robin Milner’s original paper as an illustration of “the main limitation of the system” [16, page 356]).

Example 1. In a functional programming language we can write definitions of the form: $m = \lambda f.\lambda x.\lambda y.(fx, fy)$. Applying $m$ to a polymorphic function, say the polymorphic identity function $\lambda x.x : \forall a(a \rightarrow a)$, and then applying the result to $3$:INT and $tt$:BOOL, we expect the outcome to be perfectly compatible with a polymorphic type discipline; that is, we expect $m(\lambda x.x, 3, tt)$ to type-check and to return the value $(3, tt)$: (INT,BOOL). However, contrary to expectation, the ML type-checker will reject the following statement:

let $m = \lambda f.\lambda x.\lambda y.(fx, fy)$ in $m(\lambda x.x, 3, tt)$

because it attempts to assign the same type to all the occurrences of the $\lambda$-bound $f$.

Recursive definitions in ML can be written as equations, i.e., in the form $F = \cdots F \cdots F \cdots$. The ML type discipline also restricts all the occurrences of a recursively defined function $F$ on both sides of its definition to have the same type. $F$ may be a single identifier or a pattern of identifiers. This leads to situations that contradict what are arguably minimal requirements of a uniform polymorphic type discipline.

Example 2. Consider the functions $f$ and $g$:

$$f \ x = x$$
$$g \ y = f \ 3$$

In ML as in other languages, there are the necessary constructs to define functions simultaneously. If $f$ and $g$ are so defined, by

1. If we take the recursive definition of a function $F$ to be the result of applying a fixpoint operator (obeying an appropriate axiom) to an abstraction $\lambda F(\cdots F \cdots F \cdots )$), then this restriction is a consequence of the monomorphic treatment of $\lambda$-bound variables. But we can also produce the recursive definition of $F$ by using a fixpoint constructor $\text{fix}$ without change in the operational semantics of the language (we simply introduce a reduction rule for $\text{fix}$ as a constructor which is equivalent to the axiom for $\text{fix}$ as an operator). With the latter approach we can consider $\text{fix}$-bound variables and $\lambda$-bound variables independently, the first polymorphically and the second monomorphically (or vice-versa).

2. Partly supported by NSF grant CCR-8601592.

3. On leave from the Institute of Mathematics, Univ of Warsaw. Partly supported by a grant of the Polish Ministry of Science and Higher Education, No. RP.1.09.

4. Partly supported by a grant of the Polish Ministry of Science and Higher Education, No. RP.1.09.
writing for example:

(1) letrec ((f x), (g y)) = (x, (f 3))

then the ML type-checker assigns the following types to f and g:

(2) \( f : \text{INT} \rightarrow \text{INT} \)

(3) \( g : \forall \alpha (\alpha \rightarrow \text{INT}) \)

But this is not the expected type assignment — or, in any case, it is different from the one obtained from the ML type-checker by taking the definition of the two functions sequentially (first f and then g), namely:

(4) \( f : \forall \alpha (\alpha \rightarrow \alpha) \)

(5) \( g : \forall \alpha (\alpha \rightarrow \text{INT}) \)

which of course mentions the type we want for the polymorphic identity f. The type assignment in (2) is the result of the ML type-checker assigning the same type to all the occurrences of f on both sides of the recursive definition in (1). The contrast between (2) and (3) can be made sharper: If f and g are defined simultaneously together with a third function h that uses f at a different type, say BOOL, by writing for example:

(4) letrec ((f x), (g y), (h z)) = (x, (f 3), (f h))

the result is a program that is not typable at all in ML.

Other programming examples which exhibit very similar typing anomalies of ML are in the references [2,7,11,12,18]. The difficulties encountered in all these examples are the result of assigning different types to operationally equivalent program phrases. The difficulties can be bypassed by appropriate re-coding, but not without the duplication of equivalent code. The point here is that we can remedy the problems exhibited within the ML type discipline, but only at the price of foregoing polymorphism.

Mycroft in [18] (and we independently in [11]) proposed an extension of the ML type discipline which allows "universally polymorphic recursion" (so that the definition of a function F can contain recursive calls to F at different types). The same extension, with further justification, was also considered in [7] and [13]. For later reference, let ML/1 be the extension proposed in [11,18]. The universally polymorphic recursion of ML/1 can handle the difficulty of Example 2 above but not the difficulty of Example 1.

Among other things in this paper, we propose a further extension of ML and ML/1, called ML/2, which allows different occurrences of the same \( \lambda \)-bound variable to have different types, all instances of the same generic (universal) type. ML/2 handles the difficulties of both examples above and, moreover, has a uniform type discipline in the sense that operationally equivalent program phrases are assigned the same type.

The precise definition of ML/2 is given in Section 2.

1.2 Typability in a restricted polymorphic \( \lambda \)-calculus

There are many important features of the (Girard-Reynolds) polymorphic \( \lambda \)-calculus, the most salient being Girard's characterization that the number-theoretic functions representable in this calculus are exactly the recursive functions which can be proved total in second-order arithmetic (see [4,5,22]). An outstanding open problem is whether the set of terms of the polymorphic \( \lambda \)-calculus is recursive. Instead of attacking the problem in its full generality, several people have suggested a ramification by "rank" with a view to prove (or disprove) that the set of terms of the "polymorphic \( \lambda \)-calculus of rank \( k \)" for a given fixed \( k \geq 0 \) is recursive. All attempts to solve the problem even in this restricted form have been unsuccessful.

In [14], and again in [15], the authors attempted without success to prove that the "polymorphic \( \lambda \)-calculus of rank 2" is recursive. In this paper we consider a restricted polymorphic \( \lambda \)-calculus, denoted \( \Lambda_2 \), which is our own (equivalent) formulation of the "polymorphic \( \lambda \)-calculus of rank 2".

We adopt the "Curry view" of the polymorphic \( \lambda \)-calculus, in which the (untyped) terms of the \( \lambda \)-calculus are assigned type expressions involving universal quantifiers, although the same problems can be raised again in relation to the "Church view" where terms and types are defined simultaneously to produce typed terms. Hence, instead of proving (or disproving) that the set of terms of the polymorphic \( \lambda \)-calculus of rank 2 is recursive, we want to decide whether an arbitrary (untyped) \( \lambda \)-term can be assigned a type in \( \Lambda_2 \). (The "Curry view" is also the one adopted in [14] and [15].)

The precise definition of \( \Lambda_2 \) is given in Section 2.

1.3 What is Semi-Unification?

Let \( \Sigma \) be a signature consisting of one binary function symbol \( + \) and a finite set \( C \) of constant symbols, \( C = \{c_1, \ldots, c_6\} \). Let \( X = \{x_0, x_1, \ldots\} \) be a countably infinite set of variables. \( T \) denotes the set of all terms over \( \Sigma \) and \( X \). A substitution is a function \( S : X \rightarrow T \) such that \( \{x \in X | S(x) \neq x\} \) is finite. Every substitution extends in a natural way to a \( \Sigma \)-homomorphism \( S : T \rightarrow T \).

An instance \( \Gamma \) of the Semi-Unification Problem is a finite subset of \( T \times T \). A substitution \( S \) is a solution of instance \( \Gamma = \{(t_1, u_1), \ldots, (t_n, u_n)\} \) iff there are substitutions \( R_1, \ldots, R_n \) such that:

\[
R_1(S(t_1)) = S(u_1), \ldots, R_n(S(t_n)) = S(u_n)
\]

1The extension ML/1 is called ML* in [11] and [12]. But the names ML* and ML++ are also used in [8] to denote very different extensions, so it seemed best to adopt altogether new names.

2The notation \( FX = M \) is syntactic sugar for \( \lambda x. F(\lambda x. M) \).
Each pair \((t, u)\) in such an instance \(\Gamma\) is called an inequality. The Semi-Unification Problem (henceforth abbreviated SUP) is the problem of deciding, for any instance \(\Gamma\), whether \(\Gamma\) has a solution.\(^8\)

The same formulation of SUP in the special case when every instance contains exactly one inequality is considered in the problem of deciding, for any instance \(r\), whether \(r\) has a for a rewrite rule to be non-terminating.

The same formulation of SUP in the general case is studied in [19] in relation to a strengthened version of a conjecture due to Kreisel, stated as follows: “There exists a computable function \(f\) such that if Peano Arithmetic proves \(A(\theta^n)\) in \(k\) steps for all \(n \leq f(k, A)\), where \(\theta^n\) is the \(n\)-th natural number, then it also proves \(\forall x A(x)\).” If SUP is decidable, then the conjecture is true. Very similar, it appears, is a study (unpublished) undertaken some ten years ago, about the relative speed of proofs in Frege systems with substitution, where the question “Can you decide whether a wff is provable within a certain number of steps?” is countered in functional programming languages restricted to one inequality. Despite its very simple and natural formulation, to this date no solution for SUP in general is known.

None of these studies proves that SUP is decidable (or undecidable) in its full generality. Some of them succeed in proving SUP is decidable in the special case when every instance is restricted to one inequality. Despite its very simple and natural formulation, to this date no solution for SUP in general is known.

1.4 The main results of the paper
The pure \(\lambda\)-terms are defined by \(M ::= x \mid (MN) \mid (\lambda x.M)\).

We also consider a \(\lambda\)-calculus augmented with special forms encountered in functional programming languages — these are the augmented \(\lambda\)-terms defined by the grammar:

\[
M ::= a \mid x \mid (MN) \mid (\lambda x.M) \mid (\text{let } z = N \text{ in } M)((\text{if } M \text{ then } N \text{ else } P))(\text{fix } x.M)
\]

where \(a\) ranges over the finite set of symbols in a first-order signature, and \(\text{let-in, if-then-else}\) and \(\text{fix}\) are constructors that will be given their standard interpretations.\(^6\)

A pure term \(M\) is typable in \(\Lambda_2\) iff there is a derivation in \(\Lambda_2\) which assigns a type to \(M\) (more details are in Section 2).

We define similarly what it means for an augmented term to be typable in ML/1 or in ML/2 (again, more details are in Section 2).

We use \(\Lambda_2\) (resp., ML/1 or ML/2) to denote both a type-inference system for pure terms (resp., for augmented terms) and the set of augmented terms typable in that system.

\(\Lambda_2\) is polynomial-time reducible to SUP, in symbols \(\Lambda_2 \leq SUP\), if there is a polynomial-time translation \(\Phi\) from pure terms to instances of SUP such that the pure term \(M\) is in \(\Lambda_2\) iff the instance \(\Phi(M)\) has a solution. If there is instead a polynomial-time translation \(\Psi\) from instances of SUP to pure terms such that the instance \(\Gamma\) has a solution iff the pure term \(\Phi(\Gamma)\) is in \(\Lambda_2\), we write SUP \(\leq \Lambda_2\).

We define similarly the following reductions:

\[
\text{ML/1} \leq \text{ML/2}, \quad \text{ML/2} \leq \text{SUP}, \quad \text{SUP} \leq \text{ML/1},
\]

with “augmented terms” instead of “pure terms” in the previous paragraph.

Although ML/1 \(\subseteq\) ML/2, a proof is needed to show that ML/1 \(\leq\) ML/2. If an augmented term \(M\) is typable in ML/1, then \(M\) is necessarily typable in ML/2, but the converse is not always true.

Theorem A. ML/1 \(\leq\) ML/2.

Theorem B. ML/2 \(\leq\) SUP.

Weaker versions of Theorem B are essentially proved in [7] and [11], and both imply that ML/1 \(\leq\) SUP.

Theorem C. SUP \(\leq\) ML/1.

The reduction in Theorem B is in fact from SUP to a lean fragment of ML/1: Every instance \(\Gamma\) is translated into an augmented term \(M\) containing no \(\text{let-in, no if-then-else, and exactly one fix in outermost position.}\)

Theorems A, B and C, show that SUP is the appropriate algebraic characterization for typability of augmented terms in both ML/1 and ML/2, as expressed by the following.

Corollary. SUP \(\leq\) ML/1 \(\leq\) ML/2 \(\leq\) SUP.

Theorem D. \(\Lambda_2 \subseteq\) SUP.

There is strong evidence that SUP is decidable in general. So far, we have failed to prove this conjecture, but the following are partial results.

Theorem E. It is decidable whether an instance \(\Gamma\) of SUP containing exactly one inequality has a solution.

Theorem E was independently established in [10] and [19].

Theorem F. If \(\Gamma = \{(t_1,u_1), \ldots, (t_n,u_n)\}\) is an instance of SUP where for every \(i \in \{1, \ldots, n\}\) and every \(x \in X\), \(x\) occurs at most once in \(t_i\) (\(x\) may occur in different \(t_i\)'s), then it is decidable whether \(\Gamma\) has a solution.
Typability of augmented terms in standard ML is reducible to SUP. The decision procedures in Theorems E and F have exponential time-complexities. Given that typability in standard ML is a PSPACE-hard problem (see [9]), there can be no polynomial-time decision procedure for SUP (unless it turns out that P = PSPACE).\footnote{In [10] it is claimed that there is a polynomial-time decision procedure for the special case of Theorem E. We have not checked the details. However, even if their claim is correct, it does not contradict our claim, based on [9], that no polynomial-time decision procedure exists for SUP in general, unless P = PSPACE.}

## 2 Type-Inference Systems

The conventional interpretation of polymorphism in programming languages has been universal quantification. Thus, for the exponential time-complexities. Given that typability in standard ML/2, only universal types need to be considered (which include as a proper subset the open types).

However, once we are committed to include a polymorphic treatment of λ-bindings, we need to consider more complicated types. The open types are defined by the grammar:

\[ \tau ::= \beta \mid \sigma \mid (\tau \rightarrow \tau') \]

where \( \beta \) ranges over the ground types \{0, BOOL\} and \( \sigma \) ranges over type variables. The universal types are expressions of the form \( \forall \sigma_1 \ldots \forall \sigma_n : \tau \), where \( n \geq 0 \) and \( \tau \) is an open type. Let \( S_0 \) and \( S_1 \) denote the open types and the universal types, respectively. The set \( S_2 \) of types considered in ML/2 is the set of all expressions of the form:

\[ \forall \beta \exists ! (\forall \sigma_1 : \tau_1 \ldots \forall \sigma_n : \tau_n \rightarrow \tau) \]

where \( \beta \) and \( \sigma \) denote sequences of (zero or more) type variables, \( \tau, \tau_1, \ldots, \tau_n \in S_0 \) and \( n \geq 0 \). It is clear that \( S_0 \subseteq S_1 \subseteq S_2 \). The type-inference system for ML/2 is shown in Figure 1. We follow standard notation and terminology. An assertion is an expression of the form \( A \vdash M : \tau \) where \( A \) is an environment (a finite set \( \{x_1 : \sigma_1, \ldots, x_n : \sigma_n\} \) associating at most one type \( \sigma \) with each variable \( x \)). \( M \) a term and \( \tau \) a type. In such an assertion, the \( \sigma \)'s are called the environment types, and \( \tau \) the assigned or derived type.

The essential difference between ML/1 and ML/2 is in the ABS rule. Whereas in the ABS rule of ML/1 (and, a fortiori, in the ABS rule of ML) the discharged environment type \( \sigma \) must be an open type, there is no such restriction in the ABS rule of ML/2. As a result, derived types in ML/2 can no longer be restricted to be universal. Closer examination of the inference rules of ML/2 will show that the class \( S_2 \) of types is the simplest class of derived types which is compatible with a polymorphic treatment of λ-bindings. Thus, when we go from ML to ML/2, we enrich the type discipline just enough to rectify the anomalies discussed in the Introduction.

When we set up \( \Lambda_2 \) we are not limited by such considerations. Rather, we define the largest fragment of the full polymorphic λ-calculus which we can reduce to SUP.

It is natural to define the entire hierarchy \( \{\Lambda_k\} \) first, of which \( \Lambda_2 \) is only one level. The types considered in the full polymorphic λ-calculus are defined by:

\[ \tau ::= \alpha \mid (\forall \alpha \tau) \mid (\sigma \rightarrow \tau) \]

where \( \alpha \) ranges over an infinite set of type variables. (We are now concerned about typability of pure terms and we do not need to include type constants in type expressions.) We classify types according to the following induction. First, we define: \( R(0) \) (open types) and then, for all \( k \geq 0 \), define \( R(k + 1) \) as the least set such that:

\[ R(k + 1) \supseteq R(k) \cup \{ (\sigma \rightarrow \tau) \mid \sigma \in R(k), \tau \in R(k + 1) \} \cup \{ (\forall \alpha \sigma) \mid \sigma \in R(k + 1) \} \]

The set of all types is: \( R(\omega) = \bigcup R(k) k \in \omega \). We say that \( R(k) \) is the set of types of rank \( k \).

The rank of the assertion \( A \vdash M : \tau \) where \( A = \{x_1 : \sigma_1, \ldots, x_n : \sigma_n\} \) is the rank of the type \( \sigma_1 \rightarrow \cdots \rightarrow \sigma_n \rightarrow \tau \).

For every \( k \geq 0 \), we define \( \Lambda_k \) as the fragment of the polymorphic λ-calculus which assigns types of rank \( \leq k \) to terms. A precise definition of \( \Lambda_k \) is given in Figure 2; it is the usual type

<table>
<thead>
<tr>
<th>RULE</th>
<th>FORMULA</th>
<th>MEANING</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONST</td>
<td>( A \vdash a : \sigma )</td>
<td>( \sigma \in S_0 ) closed type of a</td>
</tr>
<tr>
<td>VAR</td>
<td>( A \vdash \alpha : \sigma )</td>
<td>( (\alpha : \sigma) \in A )</td>
</tr>
<tr>
<td>INST</td>
<td>( A \vdash M : \forall \alpha.\sigma )</td>
<td>( \sigma \in S_0 )</td>
</tr>
<tr>
<td>GEN</td>
<td>( A \vdash M : \sigma )</td>
<td>( \sigma \notin \text{FV}(A) )</td>
</tr>
<tr>
<td>APP</td>
<td>( A \vdash M : \sigma \rightarrow \tau, A \vdash N : \sigma )</td>
<td>( A \vdash (M N) : \tau )</td>
</tr>
<tr>
<td>ABS</td>
<td>( A[\alpha : \sigma] \vdash M : \tau )</td>
<td>( A \vdash (\lambda x M) : \sigma \rightarrow \tau )</td>
</tr>
<tr>
<td>LET</td>
<td>( A \vdash N : \sigma, A[\alpha : \sigma] \vdash M : \tau )</td>
<td>( A \vdash \text{let } x = N \text{ in } M : \tau )</td>
</tr>
<tr>
<td>COND</td>
<td>( A \vdash M : \text{BOOL}, A \vdash N : \sigma, A \vdash P : \sigma )</td>
<td>( A \vdash (\text{if } M \text{ then } N \text{ else } P) : \sigma )</td>
</tr>
<tr>
<td>FIX</td>
<td>( A[\alpha : \sigma] \vdash M : \sigma )</td>
<td>( A \vdash (\text{fix } x M) : \sigma )</td>
</tr>
</tbody>
</table>
inference system of the polymorphic λ-calculus with the sole restriction that all assertions in a derivation are of rank k.

\[
\begin{align*}
\text{VAR} & : A \vdash x : \sigma \\
\text{INST} & : A \vdash M : \forall a.\sigma \\
\text{GEN} & : A \vdash M : \sigma, A \vdash M : \forall a.\sigma \\
\text{APP} & : A \vdash M : \sigma \rightarrow \tau, A \vdash N : \sigma \\
\text{ABS} & : A[x : \sigma] \vdash M : \tau \\
\end{align*}
\]

Figure 2. System \( \Lambda_{\Delta} \): all assertions are of rank \( k \).

For the rest of the paper we only deal with system \( \Lambda_{\Delta} \).

3 About the Proofs

The proofs of Theorems A, B, and D are inductions on the length of derivations in systems ML/1, ML/2 and \( \Lambda_{\Delta} \), respectively. In the case of Theorems B and D, derivations are also transformed into appropriately defined canonical forms. There are two essential parts in this transformation of derivations into canonical forms. The first consists in the elimination of the INST and GEN rules, which in turn requires an appropriate reformulation of the other inference rules, with a view to make all derivations "deterministic." The second part in our transformation is to displace quantifiers in type expressions "without changing their semantics". Once the derivation of a term with its assigned type semantics". Once the derivation of a term with its assigned type.

3.1 Two Conditions to Satisfy

A term \( t \in T \) can be viewed as a (non-empty and finite) binary tree, denoted \( \text{tree}(t) \), together with an appropriate labelling of the nodes of \( \text{tree}(t) \), and a binary tree is viewed as a subset of \( \{L, R\}^* \) satisfying appropriate closure properties. Let \( \text{interior}(t) \) and \( \text{frontier}(t) \) abbreviate \( \text{interior}(\text{tree}(t)) \) and \( \text{frontier}(\text{tree}(t)) \), respectively. The partial function that labels the nodes of \( \text{tree}(t) \) is \( \text{label}(t) : \{L, R\}^* \rightarrow X \).

A subterm of \( t \in T \) is specified by a node \( p \in \text{tree}(t) \), denoted \( \text{subterm}(t, p) \). If \( p \notin \text{tree}(t) \) then \( \text{subterm}(t, p) = \emptyset \).

Lemma 1 Let \( \text{id} : T \rightarrow T \) be the identity substitution. If \( \Gamma \) is an instance of SUP such that for every inequality \( (t, u) \in \Gamma \):

(A) \( \text{tree}(t) \subseteq \text{tree}(u) \),

(B) \( \forall p \in \text{frontier}(t) \) \[ \text{label}(t, p) = \text{label}(t, q) \Rightarrow \text{subterm}(u, p) = \text{subterm}(u, q) \],

then \( \Gamma \) has a solution and, in particular, \( \text{id} \) is a solution of \( \Gamma \).

Proof: Straightforward consequence of the definitions. □

3.2 Extended Labelling of Terms

We assume the existence of coding functions \( \langle \cdot, \cdot \rangle : \omega \times \omega \rightarrow \omega, L : \omega \rightarrow \omega, \) and \( R : \omega \rightarrow \omega \) (the latter two written in postfix notation), such that:

(1) \( \forall i \in (\omega, \omega) \)((iL, iR) = i] \),

(2) \( \forall j, k, j', k' \in \omega \)(\( (j, k) = (j', k') \) \( \Leftrightarrow j = j' \) \( \land k = k' \)),

where \( \omega \) is the range of \( \langle \cdot, \cdot \rangle \). The set \( I(\Gamma) \) of initial indices of \( \Gamma \) is:

\[ I(\Gamma) = \{ i \mid z_i \in \text{var}(\Gamma) \} \]

The set \( I_{\text{ext}}(\Gamma) \) of external indices of \( \Gamma \) is:

\[ I_{\text{ext}}(\Gamma) = \{ ip \mid i \in I(\Gamma) \land p \in \{ L, R \}^* \} \]

and the set \( I_{\text{int}}(\Gamma) \) of internal indices is the smallest such that:

\[ I_{\text{int}}(\Gamma) \supseteq \{ (j, k) \mid j, k \in I(\Gamma) \cup I_{\text{ext}}(\Gamma) \} \]

We make two further requirements:

(3) \( I(\Gamma) \cap I_{\text{ext}}(\Gamma) = \emptyset, I(\Gamma) \cup I_{\text{ext}}(\Gamma) = \emptyset, I_{\text{int}}(\Gamma) \cap I_{\text{int}}(\Gamma) = \emptyset, \)

(4) \( \forall i, j \in I(\Gamma) \cup I_{\text{ext}}(\Gamma) \cup I_{\text{int}}(\Gamma) \forall a \in \{ L, R \} \)

\[ \{ ia = ja \Leftrightarrow i = j \} \]

If \( y \in X \) and \( p \in \{ L, R \}^* \), we write \( y_p \) to denote the variable \( x_p \), where \( y = x_i \); in particular, we have \( y_e = y \) (\( e \) is the empty string).

We use the coding functions \( \langle \cdot, \cdot \rangle, L, \) and \( R \) to extend the labelling function \( \text{label}(t, \cdot) \) to \( \text{label}^*(t, \cdot) : \{ L, R \}^* \rightarrow X \), for any \( t \in T \), by requiring:

- \( \forall p \in \text{interior}(t) \)

\[ \text{label}^*(t, pL) = z_i \] and \( \text{label}^*(t, pR) = z_j \)

\[ \Rightarrow \text{label}^*(t, p) = x_{i,j} \]
The function \( \text{label}^*(t, \cdot) \) is a total function on \( \{L, R\}^* \).

We can always choose variable indeces appearing in an arbitrary instance \( \Gamma \) of SUP so that conditions (3) and (4) are satisfied. In particular, if \( \Gamma \) satisfies (3) and (4) and \( S : T \rightarrow T \) is a substitution, we require that \( S \) be such that \( S(\Gamma) \) also satisfy (3) and (4). Although variables can now be labels of internal nodes of terms viewed as binary trees, when we write \( S(\Gamma) \) we mean that \( S \) only acts on variables in \( \text{var}(\Gamma) \), i.e., on variables labelling leaf nodes. There is no problem in requiring that \( S(\Gamma) \) also satisfy (3) and (4), because there is an infinite supply of variable names.

We say that a substitution \( S \) is an expansion substitution if \( \text{label}(S(y), p) = y_p \) for every \( y \in X \) and every \( p \in \text{frontier}(S(y)) \). We say that instance \( \Gamma' \) is an expansion of instance \( \Gamma \) iff \( \Gamma' = S(\Gamma) \) for some expansion substitution \( S \), i.e.,

\[
\Gamma' = \{(S(t), S(u)) \mid (t, u) \in \Gamma\}
\]

Define the operation \( \text{expand} \) on instances of SUP as follows.

**OPERATION expand:**

1. **input:** instance \( \Gamma \), variable \( y \in X \);
2. \( S := \text{id} \circ f(y, y_0) \);
3. \( \Gamma' := S(\Gamma) \);
4. **output:** instance \( \Gamma' \).

**Lemma 2** If \( \Gamma = \{(t_1, t_2), \ldots, (t_{2n-1}, t_{2n})\} \) is an instance of SUP satisfying conditions (3) and (4), \( S : T \rightarrow T \) is an expansion substitution, and \( \Gamma' = S(\Gamma) \), then:

1. \( \Gamma' \) also satisfies condition (3),
2. \( T(\Gamma') \cup T_{\text{int}}(\Gamma) \cup T_{\text{int}}(\Gamma') \cup T_{\text{int}}(\Gamma'') \), which implies that \( \Gamma' \) also satisfies (4),
3. for every term \( t \) of \( \Gamma \) and every \( p \in \{L, R\}^* \), \( \text{label}^*(t, p) = \text{label}^*(S(t), p) \).

**Proof:** Straightforward from the definitions.

**3.3 Reformulation of the Two Conditions**

Let \( \Gamma \) be an instance of SUP, with the extended labelling defined above. We define equivalence relations \( \approx_0, \approx_1, \approx_2, \ldots \) on \( X \) relative to \( \Gamma \) as follows. First, we set:

\[
y \approx_0 z \iff y = z
\]

for all \( y, z \in X \). Proceeding inductively, assume we have defined \( \approx_0, \approx_1, \ldots, \approx_k \) for some \( k \geq 0 \). The relation \( \approx_{k+1} \) (not yet \( \approx_{k+1} \)) is:

\[
y \approx_{k+1} z \iff \exists \nu \in \text{Sup}(\Gamma) \text{ such that } \nu_y = \nu_z
\]

The sole exception of substitution \( \nu \) in Subsection 3.3. We do not need to know that \( \nu \) also satisfies (3) and (4).

\[
\approx_{k+1} z \iff y \approx_0 z \text{ for all } y, z \in X.
\]

**OR**

\[
(\exists (t, u) \in \Gamma)(\exists p, q \in \{L, R\}^*) \text{ such that } \text{label}^*(t, p) \approx_k \text{label}^*(t, q) \land y = \text{label}^*(u, p) \land z = \text{label}^*(u, q)
\]

for all \( y, z \in X \). In general, \( \approx_{k+1} \) is not yet an equivalence because it fails to be transitive. The desired \( \approx_{k+1} \) is the reflexive and transitive closure of \( \approx_{k+1} \). Having defined \( \approx_k \), for every \( k \geq 0 \), we finally define \( \approx \) as:

\[
y \approx z \iff (\exists k \geq 0)(y \approx_k z)
\]

for all \( y, z \in X \).

We define the substitution \( \nu \) by setting, for all \( x_i \in X \), \( \nu(x_i) = x_k \) where \( k = \min\{j|z_j \approx x_i\} \). The effect of \( \nu \) is simply to rename all variables in the same equivalence class with a same variable.

We reformulate conditions (A) and (B) of Subsection 3.1 as follows.

**Lemma 3** If \( \Gamma = \{(t_1, t_2), \ldots, (t_{2n-1}, t_{2n})\} \) is an instance of SUP such that:

1. (A) (Vodd it \{l, \ldots, r\})
   \[\text{tree}(t_i) \subseteq \text{tree}(t_{i+1})\]
2. (B) (Viy, j \in \{1, \ldots, 2n\})(Vp,q \in \{L, R\}^*)
   \[
   \text{label}^*(t_i, p) = \text{label}^*(t_j, q) \Rightarrow \text{subtree}(t_i, p) = \text{subtree}(t_j, q),
   \]
then \( \nu(\Gamma) \) satisfies conditions (A) and (B) of Lemma 1, i.e., \( \nu \) is a solution of \( \Gamma \).

**Proof:** (A1) = (A). The non-trivial part is to show that (A1) and (B1) together imply (B).

**3.4 The Measure \( \Delta \)**

Let \( \Gamma = \{(t_1, t_2), \ldots, (t_{2n-1}, t_{2n})\} \) be an instance of SUP. For every variable \( z \in X \) we define the set of terms \( \delta(\Gamma, z) \) as follows. If \( z \not\in \text{var}(\Gamma) \) then \( \delta(\Gamma, z) = \emptyset \), and if \( z \in \text{var}(\Gamma) \) then:

\[
\delta(\Gamma, z) = \{\text{subterm}(t_i, p) \mid (t_i, t_{i+1}) \in \Gamma, \ p \in \text{tree}(t_i), \ z = \text{label}(t_{i+1}, p)\}
\]

For every \( z \), \( \delta(\Gamma, z) \) contains finitely many terms \( \neq \emptyset \). The measure \( \Delta \) is:

\[
\Delta(\Gamma, z) = |\text{interior}(\bigcup\{\text{tree}(u) \mid u \in \delta(\Gamma, z)\})|.
\]

This definition is meaningful because the union of binary trees

\footnote{In general, \( \nu \) is a substitution with finite support. Definitions and proofs are not affected.}
(i.e., their least upper bound in the poset of binary trees partially ordered under \( \leq \)) is again a binary tree. To cover the case when \( \delta(\Gamma, z) = \emptyset \) in the definition of \( \Delta(\Gamma, z) \), we pose \( \text{interior}(\emptyset) = \emptyset \).

Finally, we define:

\[
\Delta(\Gamma) = \sum \{ \Delta(\Gamma, z) \mid z \in X \}.
\]

**Lemma 4** Under the hypothesis of Theorem F, \( \Delta(\Gamma) = 0 \) iff \( \Gamma \) satisfies (A1) and (B1) of Lemma 3.

**Proof:** Straightforward. \( \blacksquare \)

### 3.5 A Procedure to Decrease the Value of \( \Delta \)

The procedure \( A \) expands a given instance \( \Gamma \) of SUP by applying \( \text{expand} \) repeatedly.

**PROCEDURE A:**
1. input: instance \( \Gamma \) and variable \( y \in \text{var}(\Gamma) \);
2. \( \Gamma' := \Gamma := \Gamma' \);
3. \( Z := \{ y \} \);
4. repeat \( y' := \text{choice}(Z) \);
5. \( \Gamma' := \Gamma'' \);
6. \( \Gamma'' := \text{expand}(\Gamma', y') \);
7. \( Z := \{ Z \} \cup \{ z \in X \mid z \neq y_L, z \neq y_R, \Delta(\Gamma'', z) > \Delta(\Gamma', z) \} \)
8. until \( Z = \emptyset \);
9. output: instance \( \Gamma'' \).

Procedure \( A \) is non-deterministic, because in line 4 the function \( \text{choice} \) arbitrarily chooses a variable from the set \( Z \).

**Lemma 5** Let \( \Gamma \) be an instance of SUP which has a solution. If \( \Delta(\Gamma) \neq 0 \), then there is \( y \in \text{var}(\Gamma) \) and a computation of procedure \( A \) on input \( (\Gamma, y) \) which:

1. terminates,
2. applies the operation \( \text{expand} \) at most \( |\text{var}(\Gamma)| \) times,
3. returns as output an instance \( \Gamma'' \) such that \( |\text{var}(\Gamma'')| \leq 2 \cdot |\text{var}(\Gamma)| \), and
4. returns as output an instance \( \Gamma'' \) such that \( \Delta(\Gamma'') = \Delta(\Gamma) - 1 \).

**3.6 The Decision Procedure**

If instance \( \Gamma \) has a solution, then procedure \( B \) below expands \( \Gamma \) to another instance \( \tilde{\Gamma} \) such that \( \Delta(\tilde{\Gamma}) = 0 \).

**PROCEDURE B:**
1. input: instance \( \Gamma \);
2. \( \tilde{\Gamma} := \Gamma \);
3. \( Z := \{ z \in X \mid \Delta(\tilde{\Gamma}, z) \neq 0 \} \);
4. while \( Z \neq \emptyset \) do
5. begin \( y := \text{choice}(Z) \);
6. \( \Gamma := \tilde{\Gamma} \);
7. run procedure \( A \) on input \( (\Gamma, y) \) to produce an instance \( \Gamma' \) (if and when \( A \) terminates);
8. \( Z := \{ z \in X \mid \Delta(\tilde{\Gamma}, z) \neq 0 \} \) end
9. output: instance \( \tilde{\Gamma} \).

The size \( |\Gamma| \) of an instance \( \Gamma = \{(t_1, t_2), \ldots, (t_{2m-1}, t_{2m})\} \) is:

\[
|\Gamma| = \sum \{ |t| \mid t \in \{t_1, t_2, \ldots, t_{2m} \} \}.
\]

**Lemma 6** If instance \( \Gamma \) of SUP has a solution, there is a computation of \( B \) on input \( \Gamma \) which terminates and applies operation \( \text{expand} \) less than \( |\Gamma| \cdot 2^{|\Gamma|^2} \) times. Moreover, the output \( \tilde{\Gamma} \) of the computation has a solution and, under the hypothesis of Theorem F, satisfies conditions (A1) and (B1) of Lemma 3.

**Proof:** This easily follows from Lemma 5. \( \blacksquare \)

Theorem F is an immediate consequence of Lemma 6.
References


