COMPUTATIONAL CONSEQUENCES AND PARTIAL SOLUTIONS OF A GENERALIZED UNIFICATION PROBLEM

(Partial Report)

A.J. Kfoury*
Dept of Computer Science
Boston University

J. Tiuryn†
Dept of Computer Science
Washington State Univ

P. Urzyczyn‡
Institute of Mathematics
University of Warsaw

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 λ-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 types.

There are nevertheless two important limitations in the original **ML** type discipline. One limitation forces all the occurrences of the same λ-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 $\lambda x.x : \forall\alpha(\alpha \to \alpha)$, 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:

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

because it attempts to assign the same type to all the occurrences of the λ-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.[1] 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:

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

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 λ-bound variables. But we can also produce the recursive definition of F by using a fixpoint constructor **fix** without change in the operational semantics of the language (we simply introduce a reduction rule for **fix** as a constructor which is equivalent to the axiom for **fix** as an operator). With the latter approach we can consider **fix**-bound variables and λ-bound variables independently, the first polymorphically and the second monomorphically (or vice-versa).

*Partly supported by NSF grant CCR-8601592 .

†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.I.09 .

‡Partly supported by a grant of the Polish Ministry of Science and Higher Education, No. RP.I.09 .

writing for example[2]:

(1) **letrec** $((f\ x),(g\ y)) = (x,(f\ 3))$

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

(2) f: INT $\longrightarrow$ INT $\qquad\qquad$ $g : \forall\alpha(\alpha \longrightarrow \text{INT})$

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:

(3) $f : \forall\alpha(\alpha \longrightarrow \alpha) \qquad\qquad g : \forall\alpha(\alpha \longrightarrow \text{INT})$

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** $((f\ x),(g\ y),(h\ z)) = (x,(f\ 3),(f\ tt))$

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 by-passed 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.[3]

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 λ-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.[4]

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

1.2 Typability in a restricted polymorphic λ-calculus

There are many important features of the (Girard-Reynolds) *polymorphic λ-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 λ-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 λ-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 λ-calculus of rank 2" is recursive. In this paper we consider a restricted polymorphic λ-calculus, denoted Λ_2, which is our own (equivalent) formulation of the "polymorphic λ-calculus of rank 2".

We adopt the "Curry view" of the polymorphic λ-calculus, in which the (untyped) terms of the λ-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 λ-calculus of rank 2 is recursive, we want to decide whether an arbitrary (untyped) λ-term can be assigned a type in Λ_2. (The "Curry view" is also the one adopted in [14] amd [15].)

The precise definition of Λ_2 is given in Section 2.

1.3 What is Semi-Unification?

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

An instance Γ of the Semi-Unification Problem is a finite subset of $\mathcal{T} \times \mathcal{T}$. 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:

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

[3] Milner and his co-workers were aware of the difficulty of the first example but, it seems, not that of the second. They nevertheless chose not to include a polymorphic treatment of λ-bindings because they did not know whether it would be otherwise possible to perform automatic type reconstruction [16, page 356].

[2] The notation **letrec** $FX = M$ is syntactic sugar for **let** F = **fix** $F.\lambda X.M$

[4] The 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.

Each pair (t, u) in such an instance Γ is called an *inequality*. The *Semi-Unification Problem* (henceforth abbreviated SUP) is the problem of deciding, for any instance Γ, whether Γ has a solution.[5]

The same formulation of SUP in the special case when every instance contains exactly one inequality is considered in [20] and [10], where semi-unifiability is studied as a sufficient condition 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 $A(0^{(n)})$ in k steps for all $n \leq f(k, A)$, where $0^{(n)}$ 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 also reduced to SUP [23].

Different but essentially equivalent formulations of SUP are also presented in earlier analyses. In [7] and [11], the authors reduce the problem of type reconstruction in $\mathbf{ML}/1$ to somewhat more complicated forms of SUP.

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* λ*-terms* are defined by $M ::= x \mid (M\ N) \mid (\lambda x\ M)$. We also consider a λ-calculus augmented with special forms encountered in functional programming languages — these are the *augmented* λ*-terms* defined by the grammar:

$$M ::= a \mid x \mid (M\ N) \mid (\lambda x\ M) \mid (\textbf{let } x = N \textbf{ in } M) \mid (\textbf{if } M \textbf{ then } N \textbf{ else } P) \mid (\textbf{fix } x\ M)$$

where a ranges over the finite set of symbols in a first-order signature, and **let-in, if-then-else** and **fix** are constructors that will be given their standard interpretations.[6]

A pure term M is *typable* in Λ_2 iff there is a derivation in Λ_2 which assigns a type to M (more details are in Section 2).

[5] In earlier announcements of the work presented in this paper, what is here defined as SUP was called *Generalized Unification Problem*, abbreviated GUP. We now follow the terminology adopted by several other researchers.

[6] Our definition of augmented terms does not include the pairing constructor (necessary to write the examples in Subsection 1.1) as well as various other constructors of functional languages, because the type discipline proposed here is not affected by their presence and can be examined relative to a minimal functional language.

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

We use Λ_2 (resp., $\mathbf{ML}/1$ or $\mathbf{ML}/2$) to denote both a type-inference system for pure terms (resp., for augmented terms) and the set of augmented terms typable in that system.

Λ_2 is *polynomial-time reducible* to SUP, in symbols $\Lambda_2 \leq$ SUP, if there is a polynomial-time translation Φ from pure terms to instances of SUP such that the pure term M is in Λ_2 iff the instance $\Phi(M)$ has a solution. If there is instead a polynomial-time translation Ψ from instances of SUP to pure terms such that the instance Γ has a solution iff the pure term $\Psi(\Gamma)$ is in Λ_2, we write SUP $\leq \Lambda_2$.

We define similarly the following reductions:

$$\mathbf{ML}/1 \leq \mathbf{ML}/2, \qquad \mathbf{ML}/2 \leq \text{SUP}, \qquad \text{SUP} \leq \mathbf{ML}/1,$$

with "augmented terms" instead of "pure terms" in the previous paragraph.

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

Theorem A. $\mathbf{ML}/1 \leq \mathbf{ML}/2$.

Theorem B. $\mathbf{ML}/2 \leq$ SUP.

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

Theorem C. SUP $\leq \mathbf{ML}/1$.

The reduction in Theorem B is in fact from SUP to a lean fragment of $\mathbf{ML}/1$: Every instance Γ is translated into an augmented term M containing no **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 $\mathbf{ML}/1$ and $\mathbf{ML}/2$, as expressed by the following.

Corollary. SUP $\leq \mathbf{ML}/1 \leq \mathbf{ML}/2 \leq$ SUP.

Theorem D. $\Lambda_2 \leq$ 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* Γ *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* Γ *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).[7]

2 Type-Inference Systems

The conventional interpretation of polymorphism in programming languages has been universal quantification. Thus, for the polymorphism of standard **ML** and again that of **ML**/1, 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:

$$\tau \ ::= \ b \mid \alpha \mid (\tau \to \tau') \ ,$$

where b ranges over the ground types $\{\mathbf{0}, \mathrm{BOOL}\}$ and α ranges over type variables. The *universal types* are expressions of the form $\forall\alpha_1 \cdots \forall\alpha_n.\tau$, where $n \geq 0$ and τ 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:

$$\forall\vec{\beta}((\forall\vec{\alpha}_1\ \tau_1) \to \cdots \to (\forall\vec{\alpha}_n\ \tau_n) \to \tau)$$

where $\vec{\alpha}_i$ and $\vec{\beta}$ 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 σ with each variable x), M a term and τ a type. In such an assertion, the σ's are called the *environment types*, and τ 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 σ 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.

[7] 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.

CONST	$A \vdash a : \sigma$	$\sigma \in S_0$ closed type of a
VAR	$A \vdash x : \sigma$	$(x : \sigma) \in A$
INST	$\dfrac{A \vdash M : \forall\alpha.\sigma}{A \vdash M : \sigma[\alpha := \tau]}$	$\tau \in S_0$
GEN	$\dfrac{A \vdash M : \sigma}{A \vdash M : \forall\alpha.\sigma}$	$\alpha \notin \mathrm{FV}(A)$
APP	$\dfrac{A \vdash M : \sigma \to \tau, \quad A \vdash N : \sigma}{A \vdash (M\ N) : \tau}$	
ABS	$\dfrac{A[x : \sigma] \vdash M : \tau}{A \vdash (\lambda x\ M) : \sigma \to \tau}$	
LET	$\dfrac{A \vdash N : \sigma, \quad A[x : \sigma] \vdash M : \tau}{A \vdash (\mathbf{let}\ x = N\ \mathbf{in}\ M) : \tau}$	
COND	$\dfrac{A \vdash M : \mathrm{BOOL}, \quad A \vdash N : \sigma, \quad A \vdash P : \sigma}{A \vdash (\mathbf{if}\ M\ \mathbf{then}\ N\ \mathbf{else}\ P) : \sigma}$	
FIX	$\dfrac{A[x : \sigma] \vdash M : \sigma}{A \vdash (\mathbf{fix}\ x\ M) : \sigma}$	

Figure 1. System **ML**/2: all environment types in S_1, all derived types in S_2.

When we set up Λ_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 Λ_2 is only one level. The types considered in the full polymorphic λ-calculus are defined by:

$$\tau ::= \ \alpha \mid (\forall\alpha\ \tau) \mid (\sigma \to \tau)$$

where α 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: $R(0) = \{\text{open types}\}$ and then, for all $k \geq 0$, define $R(k+1)$ as the least set such that:

$$\begin{aligned} R(k+1) \ \supseteq \ & R(k) \ \cup \\ & \{\ (\sigma \to \tau) \mid \sigma \in R(k),\ \tau \in R(k+1)\ \} \ \cup \\ & \{\ (\forall\alpha\ \sigma) \mid \sigma \in R(k+1)\ \} \end{aligned}$$

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 $\sigma_1 \to \cdots \to \sigma_n \to \tau$.

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

inference system of the polymorphic λ-calculus with the sole restriction that all assertions in a derivation are of rank k.

VAR	$A \vdash x : \sigma$	$(x : \sigma) \in A$
INST	$\dfrac{A \vdash M : \forall\alpha.\sigma}{A \vdash M : \sigma[\alpha := \tau]}$	
GEN	$\dfrac{A \vdash M : \sigma}{A \vdash M : \forall\alpha.\sigma}$	$\alpha \notin \mathrm{FV}(A)$
APP	$\dfrac{A \vdash M : \sigma \to \tau, \quad A \vdash N : \sigma}{A \vdash (M\ N) : \tau}$	
ABS	$\dfrac{A[x : \sigma] \vdash M : \tau}{A \vdash (\lambda x\ M) : \sigma \to \tau}$	

Figure 2. System Λ_k: all assertions are of rank k .

For the rest of the paper we only deal with system Λ_2.

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 Λ_2, 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".[8] 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 is in canonical form, it is relatively straightforward to read from it the corresponding instance of SUP.

The proof of Theorem C is a construction of programs that "simulate" the finitely many inequalities in a given instance Γ of SUP, such that the constructed programs are typable iff Γ has a solution.

We say more about the approach of our examination of SUP. The discussion is simplified if we assume that terms in instances of SUP do not contain constant symbols, i.e., $C = \emptyset$.

3.1 Two Conditions to Satisfy

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

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

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

(A) $\mathrm{tree}(t) \subseteq \mathrm{tree}(u)$,

(B) $(\forall p, q \in \mathrm{frontier}(t))$
$[\mathrm{label}(t, p) = \mathrm{label}(t, q) \Longrightarrow \mathrm{subterm}(u, p) = \mathrm{subterm}(u, q)]$,

then Γ *has a solution and, in particular,* id *is a solution of* Γ.

Proof: Straightforward consequence of the definitions. ∎

3.2 Extended Labelling of Terms

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

(1) $(\forall i \in \langle\omega, \omega\rangle)[\langle iL, iR\rangle = i]$,

(2) $(\forall j, k, j', k' \in \omega)[\langle j, k\rangle = \langle j', k'\rangle \Longleftrightarrow j = j' \text{ and } k = k']$,

where $\langle\omega, \omega\rangle$ is the range of $\langle\cdot,\cdot\rangle$. The set of variables appearing in the terms of an instance Γ of SUP is denoted *var*(Γ). The set $\mathcal{I}(\Gamma)$ of *initial indeces* of Γ is:

$$\mathcal{I}(\Gamma) = \{i \mid x_i \in \mathit{var}(\Gamma)\}$$

The set $\mathcal{I}_{ext}(\Gamma)$ of *external indeces* of Γ is:

$$\mathcal{I}_{ext}(\Gamma) = \{ip \mid i \in \mathcal{I}(\Gamma) \text{ and } p \in \{L, R\}^+\}$$

and the set $\mathcal{I}_{int}(\Gamma)$ of *internal indeces* is the smallest such that:

$$\mathcal{I}_{int}(\Gamma) \supseteq \{\langle j, k\rangle \mid j, k \in \mathcal{I}(\Gamma) \cup \mathcal{I}_{int}(\Gamma)\}$$

We make two further requirements:

(3) $\mathcal{I}(\Gamma) \cap \mathcal{I}_{ext}(\Gamma) = \emptyset, \mathcal{I}(\Gamma) \cap \mathcal{I}_{int}(\Gamma) = \emptyset, \mathcal{I}_{ext}(\Gamma) \cap \mathcal{I}_{int}(\Gamma) = \emptyset$,

(4) $(\forall i, j \in \mathcal{I}(\Gamma) \cup \mathcal{I}_{ext}(\Gamma) \cup \mathcal{I}_{int}(\Gamma))(\forall a \in \{L, R\})$
$[\, ia = ja \iff i = j \,]$.

If $y \in X$ and $p \in \{L, R\}^*$, we write y_p to denote the variable x_{ip} where $y = x_i$; in particular, we have $y_\epsilon = y$ (ϵ is the empty string).

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

- $(\forall p \in \mathit{interior}(t))$
$[\mathit{label}^*(t, pL) = x_i \text{ and } \mathit{label}^*(t, pR) = x_j$
$\Longrightarrow \mathit{label}^*(t, p) = x_{\langle i,j\rangle}]$

[8] The elimination of the INST and GEN rules, to get rid of the non-determinism they introduce in derivations in polymorphic type-inference systems, is not a new idea. It was already used by others in one way or another, e.g., in the references [1,3]. In the case of ML/2, our elimination of INST and GEN is quite similar to that of [1], although ours is made more complicated by the richer type discipline.

- $(\forall p \in \mathit{frontier}(t))(\forall q \in \{L,R\}^*)$
  $[\mathit{label}^*(t,p) = x_i \implies \mathit{label}^*(t,pq) = x_{iq}]$

The function $\mathit{label}^*(t,\cdot)$ is a total function on $\{L,R\}^*$.

We can always choose variable indeces appearing in an arbitrary instance Γ of SUP so that conditions (3) and (4) are satisfied. In particular, if Γ satisfies (3) and (4) and $S : \mathcal{T} \to \mathcal{T}$ is a substitution, we require that S be such that $S(\Gamma)$ also satisfy (3) and (4).[9] 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 $\mathit{var}(\Gamma)$, 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* iff $\mathit{label}(S(y),p) = y_p$ for every $y \in X$ and every $p \in \mathit{frontier}(S(y))$. We say that instance Γ' is an *expansion* of instance Γ iff $\Gamma' = S(\Gamma)$ for some expansion substitution S, i.e.,

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

Define the operation *expand* on instances of SUP as follows.

OPERATION *expand*:
1. **input**: instance Γ, variable $y \in X$;
2. $S := id[y \mapsto f(y_L, y_R)]$;
3. $\Gamma' := S(\Gamma)$;
4. **output**: instance Γ'.

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

1. *Γ' also satisfies condition (3),*
2. *$\mathcal{I}(\Gamma) \cup \mathcal{I}_{\mathrm{ext}}(\Gamma) \cup \mathcal{I}_{\mathrm{int}}(\Gamma) = \mathcal{I}(\Gamma') \cup \mathcal{I}_{\mathrm{ext}}(\Gamma') \cup \mathcal{I}_{\mathrm{int}}(\Gamma')$, which implies that Γ' also satisfies (4),*
3. *for every term t_i of Γ and every $p \in \{L,R\}^*$, $\mathrm{label}^*(t_i,p) = \mathrm{label}^*(S(t_i),p)$.*

Proof: Straightforward from the definitions. ∎

3.3 Reformulation of the Two Conditions

Let Γ 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 Γ 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 $\sim_{k+1}$ (not yet $\approx_{k+1}$) is:

$y \sim_{k+1} z \iff$

$y \approx_k z$

OR

$(\exists (t,u) \in \Gamma)(\exists p,q \in \{L,R\}^*)$
$[\mathit{label}^*(t,p) \approx_k \mathit{label}^*(t,q) \ \wedge \ y = \mathit{label}^*(u,p) \ \wedge \ z = \mathit{label}^*(u,q)]$

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

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

for all $y, z \in X$.

We define the substitution ν by setting, for all $x_i \in X$, $\nu(x_i) = x_k$ where $k = \mathit{min}\{j \mid x_j \equiv x_i\}$.[10] The effect of ν 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:*

(A1) $(\forall \text{ odd } i \in \{1,\ldots,2n\})$
$[\ \mathrm{tree}(t_i) \subseteq \mathrm{tree}(t_{i+1})\ ]$,

(B1) $(\forall i,j \in \{1,\ldots,2n\})(\forall p,q \in \{L,R\}^*)$
$[\ \mathrm{label}^*(t_i,p) \equiv \mathrm{label}^*(t_j,q)$
$\implies \mathrm{subtree}(t_i,p) = \mathrm{subtree}(t_j,q)\ ]$,

then $\nu(\Gamma)$ satisfies conditions (A) and (B) of Lemma 1, i.e., ν is a solution of Γ.

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

3.4 The Measure Δ

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 $z \notin \mathit{var}(\Gamma)$ then $\delta(\Gamma,z) = \emptyset$, and if $z \in \mathit{var}(\Gamma)$ then:

$\delta(\Gamma,z) =$

$\{\mathit{subterm}(t_i,p) \mid (t_i,t_{i+1}) \in \Gamma,\ p \in \mathit{tree}(t_i),\ z = \mathit{label}(t_{i+1},p)\}$

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

$$\Delta(\Gamma,z) = |\mathit{interior}(\bigcup\{\ \mathit{tree}(u) \mid u \in \delta(\Gamma,z)\ \})| \ .$$

This definition is meaningful because the union of binary trees

[9] With the sole exception of substitution ν in Subection 3.3. We do not need to know that ν also satisfies (3) and (4).

[10] In general ν is a substitution with infinite support. Definitions and proofs are not affected.

(i.e., their least upper bound in the poset of binary trees partially ordered under $\subseteq$) is again a binary tree. To cover the case when $\delta(\Gamma, z) = \emptyset$ in the definition of $\Delta(\Gamma, z)$, we pose $interior(\emptyset) = \emptyset$. 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* Γ *satisfies* (A1) *and* (B1) *of Lemma 3.*

Proof: Straightforward. ∎

3.5 A Procedure to Decrease the Value of Δ

The procedure $\mathcal{A}$ expands a given instance Γ of SUP by applying *expand* repeatedly.

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

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

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

(1) terminates,

(2) applies the operation expand *at most* $|\mathrm{var}(\Gamma)|$ *times,*

(3) returns as output an instance Γ'' *such that* $|\mathrm{var}(\Gamma'')| \leq 2 \cdot |\mathrm{var}(\Gamma)|$, *and*

(4) returns as output an instance Γ'' *such that* $\Delta(\Gamma'') = \Delta(\Gamma) - 1$. ∎

3.6 The Decision Procedure

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

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

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

$$|\Gamma| = \sum\{\ |t| \mid t \in \{t_1, t_2, \ldots, t_{2n}\}\ \} .$$

Lemma 6 *If instance* Γ *of* SUP *has a solution, there is a computation of* $\mathcal{B}$ *on input* Γ *which terminates and applies operation* 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. ∎

Theorem F is an immediate consequence of Lemma 6.

References

[1] Clément, D., Despeyroux, J., Despeyroux, T., Kahn, G., "A simple applicative language: Mini-ML", *Proc. ACM Conference on Lisp and Functional Programming*, 1986.

[2] Giannini, P., "Type-checking and type deduction techniques for polymorphic programming languages", Technical Report, Dept of Computer Science, CMU, Dec 1985.

[3] Giannini, P., Ronchi Della Rocca, S., "Characterization of typings in polymorphic type discipline", *Proc of IEEE 3-rd LICS*, July 1988.

[4] Girard, J.-Y., *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieure*, Doctoral thesis, Université Paris VII, 1972.

[5] Girard, J.-Y., "Une extension de l'interprétation de Gödel a l'analyse et son application à l'élimination des coupures dans l'analyse et la théorie des types", in *Proc. 2nd Scandinavian Logic Symposium*, ed. Fenstad, North-Holland, 1971.

[6] Gordon, M.J., Milner, R. and Wadsworth, C.P., *Edinburgh LCF*, LNCS 78, Springer-Verlag, 1979.

[7] Henglein, F., "Type inference and semi-unification", *Proc. ACM Symp. LISP and Functional Programming*, July 1988.

[8] Jategaonkar, L.A., Mitchell, J.C., "ML with extended pattern matching and subtypes", *Proc. ACM Symp. on LISP and Functional Programming*, July 1988.

[9] Kanellakis, P., Mitchell, J.C., "Polymorphic unification and ML typing", *Proc. 16-th Symp. on Principles of Prog. Lang.*, pp 105-115, Jan 1989.

[10] Kapur, D., Musser, D., Narendran, P., and Stillman, J., "Semi-unification", *Proc. of 8-th Conference on Foundations of Software Technology and Theoretical Computer Science*, Pune, India, Dec. 1988.

[11] Kfoury, A.J., Tiuryn, J. and Urzyczyn, P., "A proper extension of ML with an effective type-assignment," *Proc. 15-th ACM Symp. Principles of Programming Languages*, 1988.

[12] Kfoury, A.J., Tiuryn, J. and Urzyczyn, P., "On the computational power of universally polymorphic recursion", *Proc. of IEEE 3-rd LICS* 1988.

[13] Leiss, H., "On type inference for object-oriented programming languages", *Proc. Logik in der Informatik*, Karlsruhe, Springer-Verlag, Dec 1987.

[14] Leivant, D., "Polymorphic type inference", *Proc. of 10-th POPL*, ACM, 1983.

[15] McCracken, N., "The typechecking of programs with implicit type structure", in *Semantics of Data Types*, ed. by Kahn, McQueen and Plotkin, Springer-Verlag LNCS 173, 1984.

[16] Milner, R., "A theory of type polymorphism in programming", *J. of Computer and System Sciences*, Vol. 17, pp 348-375, 1978.

[17] Milner, R., "The standard ML core language", *Polymorphism*, II (2), October 1985.

[18] Mycroft, A., "Polymorphic type schemes and recursive definition," *Int'l Symp. on Programming*, ed. M. Paul and B. Robinet, LNCS Vol 167, pp 217-228, Springer-Verlag, 1984.

[19] Pudlák, P., "On a unification problem related to Kreisel's conjecture", *Commentationes Mathematicae Universitatis Carolinae*, Prague, Tchechoslovakia, **29**, no. 3, pp 551-556, 1988.

[20] Purdom, P.W., "Detecting looping simplifications", in *Proc. 2-nd Conference on Rewriting Techniques and Applications (RTA)*, Bordeaux, France, LNCS Vol 250, pp 54-62, Springer-Verlag, 1987.

[21] Reynolds, J., "Towards a theory of type structure", *Proc. Colloque sur la Programmation*, pp 408-425, Springer-Verlag LNCS 19, 1974.

[22] Statman, R., "Number theoretic functions computed by polymorphic programs", *Proc. of IEEE FOCS*, pp 279-282, 1981.

[23] Statman, R., Private communication, August 1988.

[24] Turner, D.A., "Miranda: a non-strict functional language with polymorphic types", *IFIP Int'l Conf. on Functional Programming and Computer Architecture*, LNCS 201, Springer-Verlag, 1985.