ソフトウェア基礎論　37－5
プログラミング言語　27－5
（1990. 12. 13）

多相型を持つPrologにおける型検査法の最適化

申　東旭

富士通（株）国際情報社会科学研究所

あらまし：　多相型を持つPrologは、Prologに多相型システムを付け加えた言語である。この拡張は、Prologをより明解にさせ、仕様言語として適切なものとするので、広く研究されている。しかしこの多相型システムはゴールが実行される時、必ず型の一貫性を検査しなければならないのでoverheadが大きいという問題がある。これまで、この問題点を克服するためwell-typingという概念に基づく型検査法の最適化が開発されたが、最適化できる条件が厳しいてすから、まだこの方法が適用されるプログラムは多くない。本論文は、これまでの手法よりさらに弱いwell-typingの概念を提案する。これによって、広い範囲のプログラムで型検査法が最適化されるようになった。加えて、本論文はモード情報に関連のある最適化手法を発展させ、しかもモードが使えるならば最適化が促進されることを示す。

An Optimal Type Checking for Polymorphic Typed Prolog[1]

Dongwook SHIN

International Institute for Advanced Study of Social Information Science,
FUJITSU LIMITED

17-25, SHINKAMATA, 1-CHOME,OTA-KU, TOKYO 144, JAPAN

Abstract:　Polymorphic typed Prolog is a language which offers polymorphic type system in Prolog framework. This type extension of Prolog has received widespread attraction, because it is believed to make Prolog clearer and more suitable for specification. However, it may raise inefficiency in resolution procedure in that unification should always deal with the type consistency of terms. To improve the inefficiency, some optimization techniques based on the notion of *well-typing* have been developed. However these require too strong conditions, so that they can be utilized only in a limited number of programs. This paper attempts to establish another weaker notion of well-typing than theirs, so that the resolution procedure could be optimized in a wider range of programs. Furthermore, it develops an optimization technique associated with mode information and shows that optimization can be promoted if modes are available.

[1]The full version of this paper is "Toward Optimal Type Checking for The Polymorphic Type System of Prolog" [Shin 90]. All proofs omitted here are found in the paper.

1 Introduction

From the last decade, type systems for Prolog have received widespread attention, mainly because they are believed to make Prolog clearer and more suitable for specification, and help programmers to write correct programs. The type systems of Prolog can be categorized in two approaches: descriptive and prescriptive one [Jacobs 90]. The descriptive approach [Kanamori 84, Mishra 84, Zobel 87, and Yardeni 87] attempts to infer the type information from the given Prolog programs. The goal here is to find an approximation of the success set of each predicate, which is given as the notion of type, for compile-time type prediction, or program optimization.

On the other hand, the prescriptive approach does not infer the type information. Rather, it aims to investigate if the type is used consistently or not, when the types of each predicate and function are declared. To do this, unification should treat the type consistency of terms in resolution procedure, which causes inefficiency. Mycroft and O'Keefe [Mycroft 84] proposed a polymorphic type system of Prolog in the prescriptive way and suggested a criterion to find out the programs which do not raise type errors at run time. This criterion, called well-typing, is effective because if a program is proved well-typed and free from type errors, the type consistency checking in the resolution procedure can be omitted. Hence, the simple unification in Prolog is enough to prove the goals associated with this program. Dietrich and Hagl [Dietrich 88] added subtype relation into this framework and showed that the notion of well-typing still holds if the mode information is available.

Hanus [Hanus 89a] also developed an optimization technique which sorts out the programs free from type errors at run time. It turned out that his notion of optimization technique, called type general is the same as the well-typing of Mycroft and O'Keefe's.

However the condition for well-typing and type general is too strong, so that only a limited number of type correct programs are optimized by this technique. For instance, the following *append* program is well-typed by Mycroft and O'Keefe's notion:

Example 1.
func []: $\rightarrow$ list(α)
func . : α,list(α) $\rightarrow$ list(α)
pred append: list(α),list(α),list(α)
append([],X,X) $\leftarrow$
append([H|T],X,[H|T1]) $\leftarrow$ append(T,X,T1)

However, if the following clause is added to this program:

append([1],[2],[1,2]) $\leftarrow$

it becomes ill-typed. This kind of specific clause is often added to prove some queries immediately. However, the optimization technique can not be applied to the expanded program, any more. Worse, a programmer may doubt if this program has serious type errors, even though it is type correct in itself and has the same semantics as the original one.

Hence, we need some way to further the optimization in a wider range of programs and at the same time, assure that the type correct clauses in themselves do not raise serious type errors. To do this, this paper, first, introduces the notion of *weakly well-typing* which is weaker than the original notion of well-typing in Mycroft and O'Keefe's [Mycroft 84], and shows that the new notion helps to optimize the various kinds of programs. In contrast to the *strongly well-typing* which assures that a program is type correct with respect to well-typed queries, the *weakly well-typing* only guarantees that a program is type correct for *permissible* queries which is a class of well-typed queries. Secondly, this paper extends the notion of well-typing to the one associated with modes. This notion is useful for programs which are not type correct in general, but do not raise any type error if some of arguments of predicates are assumed to be ground terms. This paper also presents a criterion which finds out the type correct programs with respect to given modes and proves its soundness. This paper is organized as follows. Section 2 presents the basic preliminaries used in this paper. Section 3 describes polymorphic typed Prolog, its syntax and the notion of well-typing. Section 4 presents the notion of weakly well-typing and proves the soundness of the extension. Section 5 introduces the notion of modes and an optimization technique associated with it. Section 6 states concluding remarks and discusses further studies.

2 Preliminaries

This section reviews some notions often used in logic programming and type theory. We assume that the reader is familier with Prolog and recall only the important notions associated with logic programming and type theory. A more detailed definition or explanation is referred to [Lloyd 84] and [Hanus 89a].

This paper assumes several disjoint name sets with respect to polymorphic typed Prolog programs. These sets are a set of variable names, function names, and predicate names, which are denoted by Var, Fun, and $Pred$, respectively. Variable names are denoted by x, y, z,... and often by the names beginning with capital letters like H, T, and so forth. Constants are considered as functions with arity zero. We also assume another disjoint name sets for types, namely, sets of type variable names, $Tvar$ and type constructor names, $Tcons$. $Tvar$ contains type variables used for polymorphic types, whereas $Tcons$ has the names representing specific types. For instance, $Tcons$ may include a type constructor *int* which represents the set of integers. To avoid the confusion with the term

names, types defined below are denoted by the greek letters α, β, γ,...

Based on this assumption, the syntax of type is defined as follows:

1. a type variable $\alpha \in Tvar$ is a type,

2. if τ is a type constructor and σ_i is a type for $1 \leq i \leq m$, then $\tau(\sigma_1, \cdots, \sigma_m)$ is also a type.

For instance, *int* is a type, if $int \in Tcons$, and $list(\alpha)$ is also a type, if $\alpha \in Tvar$ and $list \in Tcons$. Now, let us define the notion of substitution, renaming, and unification.

Definition 1. A *substitution* is a finite set of variable replacements $\theta = \{x_1 \leftarrow t_1, \cdots, x_n \leftarrow t_n\}$, where each x_i is a variable and t_i is a term. The set of variables affected by θ is defined as $Dom(\theta) = \{x \mid \theta(x) \neq x\}$ and the set of variables introduced by θ as $Ran(\theta)$. A term t is said to be *subsumed by* (or an *instance of*) s, if there is a substitution θ such that $t = s\theta$. In this case, we denote the relation by $t \preceq s$. If both $t \preceq s$ and $s \preceq t$ hold, we say t is a *renaming* of s (or s is a renaming of t) and write $t \cong s$. Furthermore, t and s are said to be *unifiable* if there exist a substitution θ such that $t\theta = s\theta$. These relations also hold between two predicates p and q.

The notion of substitution and unification are easily applied to types if we replace variables by type variables and terms by types.

Definition 2. A *type substitution* is a finite set of type variable replacements $\xi = \{\alpha_1 \leftarrow \tau_1, \cdots, \alpha_n \leftarrow \tau_n\}$, where each α_i is a type variable and τ_i is a type. The notion of subsumption, renaming, and unification also hold between types in the same way as terms.

This paper also uses a notation $t{:}\tau$, where t is a term and τ is a type. This notation means that t is typed by τ. Once again, the notion of substitution can be extended to these forms, in such a way that variables are replaced by terms and type variables are replaced by types at the same time.

Definition 3. A *typed substitution* is a pair of a finite set of variable replacements and a set of type variable replacements, $(\theta, \xi) = (\{x_1 \leftarrow t_1, \cdots, x_n \leftarrow t_n\}, \{\alpha_1 \leftarrow \tau_1, \cdots, \alpha_n \leftarrow \tau_n\})$, where each x_i is a variable, t_i is a term, α_i is a type variable, and τ_i is a type. A form $t{:}\tau$ is said to be *subsumed by* (or an *instance of*) $s{:}\sigma$, if there is a *typed substitution* (θ, ξ) such that $t{:}\tau = s{:}\sigma(\theta, \xi)$. In this case, we also denote the relation by $t{:}\tau \preceq s{:}\sigma$. The renaming relation and unification also hold between these forms in the same way.

3 Polymorphic Typed Prolog

Polymorphic typed Prolog presented in this paper supports parametric polymorphism which is attained by allowing type variables in the type declaration. Similar systems are Mycroft and O'Keefe's system [Mycroft 84] and Hanus's [Hanus 89a]. Mycroft and O'Keefe's system offers not only parametric polymorphism, but also overloading, however the semantics of overloading is unclear. On the other hand, Hanus's type system provides only parametric polymorphism and is exactly the same as ours. As Hanus's has its own denotational semantics [Hanus 89a], we can give the same semantics to polymorphic typed Prolog.

3.1 Type Declaration

Polymorphic typed Prolog is a prescriptive type system [Jacob 90], in that the types of all functions and predicates occurring in a program should be declared. Once these types are declared, a programmer need not annotate the types of variables in clauses, because the type inference rule in Section 3.2 (also appeared in [Hanus 89b]) infers the type annotations of the variables.

The type of a function is declared as:

func $f : \tau_1, \cdots, \tau_n \rightarrow \tau$,

where $\tau_1, \cdots, \tau_n$ are arbitrary types and τ is the result type. If $n = 0$, f is called a constant function. Note that a function need not be *type preserving* which requires that the result type τ should contain every type variable appearing in τ_i. In the similar way, the type of a predicate is declared as:

pred $p : \tau_1, \cdots, \tau_n$,

where $\tau_1, \cdots, \tau_n$ are arbitrary types. The **func** and **pred** are reserved words which denote the declarations of a function and a predicate, respectively.

The type variables in a type declaration are universally quantified over all types. And a type of a function or a predicate is called a *generic instance* of a type declaration if it can be obtained from the declaration by replacing each occurrence of one or more type variables by other types [Damas 82].

3.2 Well-typing of Prolog

A typing for a predicate or a clause associates a type with every variable, functor, and predicate. The type annotations need not be provided by the users because most general type annotations can be computed by the typing rule in Table 1 [Hanus 89b]. A *typed variable* has the form $x : \tau$ where $x \in Var$ and τ is an arbitrary type. V is called an *allowed set of typed variables* if V contains only typed variables and $x{:}\tau, x{:}\tau' \in V$ implies $\tau = \tau'$. $H \leftarrow B$ (p, respectively) is called a *typed clause* (*typed predicate*, respectively)

Variable :	$\frac{}{V \vdash x{:}\tau}$	$(x{:}\tau \in V)$
Term :	$\frac{V \vdash t_1{:}\tau_1, \cdots, V \vdash t_n{:}\tau_n}{V \vdash f(t_1{:}\tau_1, \cdots, t_n{:}\tau_n){:}\tau}$	$(f : \tau_1, \cdots, \tau_n \rightarrow \tau$ is a generic instance of a function declaration, $n \geq 1)$
Predicate :	$\frac{V \vdash t_1{:}\tau_1, \cdots, V \vdash t_n{:}\tau_n}{V \vdash p(t_1{:}\tau_1, \cdots, t_n{:}\tau_n)}$	$(p : \tau_1, \cdots, \tau_n$ is a generic instance of a predicate declaration, $n \geq 1)$
Clause :	$\frac{V \vdash L_0, \cdots, V \vdash L_n}{V \vdash L_0 \leftarrow L_1, \cdots, L_n}$	(each L_i has the form $p(\cdots)$, $i = 0, \cdots, n)$

Table 1. Typing rule for polymorphic typed Prolog clauses

if there is an allowed set of typed variables V and $V \vdash H \leftarrow B$ ($V \vdash p$, respectively) is derivable by the typing rule in Table 1. A typed clause (typed predicate, respectively) of a clause C (a predicate p, respectively) is denoted by $\overline{C}$ ($\overline{p}$, respectively).

For instance, applying the typing rule for the first *append* clause in Example 1, we get a typed clause:

$$append([]{:}list(\alpha), X{:}list(\alpha), X{:}list(\alpha)).$$

Note that the variable X can not be annotated by a type variable β, because β is not a generic instance of $list(\alpha)$ which is the declared argument type of *append* predicate.

Now let us define the notion of *well-typing* of a predicate, *strongly well-typing* of a clause and a program. The notion of strongly well-typing of a clause corresponds to that of well-typing in [Mycroft 84]. In this paper, we change the original terminology in order to introduce another weaker notion of well-typing in Section 4.

Definition 4. A predicate p is defined *well-typed* if there is a typed predicate $\overline{p}$ for p.

Definition 5. A clause $p(t_1, \cdots, t_n) \leftarrow B_1, \cdots, B_m$ is defined *strongly well-typed* if there is a typed clause $p(t_1{:}\tau_1, \cdots, t_n{:}\tau_n) \leftarrow \overline{B_1}, \cdots, \overline{B_m}$ such that the type of the predicate p/n is declared as $p(\rho_1, \cdots, \rho_n)$ and $(\tau_1, \cdots, \tau_n) \cong (\rho_1, \cdots, \rho_n)$. A polymorphic typed Prolog program is defined *strongly well-typed* if every clause in the program is *strongly well-typed.*

Theorem 1. (Soundness of strongly well-typing) If a polymorphic typed Prolog program is strongly well-typed and every function is *type preserving*, then an ill-typed resolvent never takes place in the refutation procedure, if the first query is well-typed [Mycroft 84].

The same proof is found in [Hanus 89a]. This property is often rephrased as "if a program is well-typed, it never invokes any run-time type error for well-typed queries".

However, as mentioned in Section 1, the condition of strongly well-typing is too severe so that it often tells that a program is not well-typed, even though it is type correct in itself. For instance, the following *append* clause,

append([1],[2],[1,2])

is not well-typed because its most general typed clause is,

$append([1{:}int]{:}list(int), [2{:}int]{:}list(int), [1{:}int, 2{:}int]{:}list(int))$,

but $append(list(int), list(int), list(int))$ is not a renaming of the type declaration $append(list(\alpha), list(\alpha), list(\alpha))$.

4 The Extended Notion of Well-typing

This section presents the notion of *weakly well-typing* and shows that it helps to promote the optimization in a wider range of programs. At first, let us introduce the notion of *most general typing* of a predicate p/n, which is the most general one among the typed predicates of p/n.

Definition 6. The *most general typing(MGT)* of a predicate $p(t_1, \cdots, t_n)$, is a *typed predicate* $p(t_1{:}\tau_1, \cdots, t_n{:}\tau_n)$ such that if there is another typed predicate $p(t_1{:}\tau'_1, \cdots, t_n{:}\tau'_n)$, then $(\tau'_1, \cdots, \tau'_n) \preceq (\tau_1, \cdots, \tau_n)$ holds. The MGT of a predicate p is denoted by $\lceil \overline{p} \rceil$ and the type of a term t with respect to the MGT of p is denoted by $\lceil \overline{p} \rceil | t$.

For example, the MGT of the predicate $append([], X, X)$ is:

$$append([]{:}list(\alpha), X{:}list(\alpha), X{:}list(\alpha)),$$

and $\lceil \overline{append([], X, X)} \rceil | X$ is $list(\alpha)$.

Using the definition of the MGT, we can extend the notion of well-typing as follows.

Definition 7. Suppose that the type of the predicate p/n is declared as $p(\rho_1, \cdots, \rho_n)$. A clause $p(t_1, \cdots, t_n) \leftarrow B_1, \cdots, B_m$ is defined *strongly well-typed* if there is a typed clause $p(t_1{:}\tau_1, \cdots, t_n{:}\tau_n) \leftarrow \overline{B_1}, \cdots, \overline{B_m}$ such that $(\tau_1, \cdots, \tau_n) \cong (\rho_1, \cdots, \rho_n)$ and every B_i is a *strongly well-typed* predicate. A predicate p/n is defined *strongly well-typed* if each clause whose head is p/n is strongly well-typed. Furthermore, a polymorphic typed Prolog program P is defined *strongly well-typed* if every predicate in P is strongly well-typed.

The notion of *strongly well-typing* in Definition 7 exactly corresponds to that in Definition 5, if a program is strongly well-typed. Now, let us present another weaker notion of well-typing.

Definition 8. A clause $C \equiv p(t_1,\cdots,t_n) \leftarrow B_1,\cdots,B_m$ is defined *weakly well-typed*, if it belongs to one of the following types, where $p(\rho_1,\cdots,\rho_n)$ is the type declaration of p/n;

type (1): $m=0, p(t_1,\cdots,t_n)$ is *ground*, and its *MGT* $p(t_1{:}\tau_1,\cdots,t_n{:}\tau_n)$ satisfies $(\tau_1,\cdots,\tau_n) \prec (\rho_1,\cdots,\rho_n)$,

type (2): $m=0$, and the *MGT* of the head predicate, $p(t_1{:}\tau_1,\cdots,t_n{:}\tau_n)$ satisfies $(\tau_1,\cdots,\tau_n) \prec (\rho_1,\cdots,\rho_n)$, furthermore, if $\tau_i \prec \rho_i$, the type variables in ρ_i do not appear in ρ_j, where $1 \leq i \neq j \leq n$,

type (3): $m>0$ and there is a typed clause for C, $p(t_1{:}\tau_1,\cdots,t_n{:}\tau_n) \leftarrow \overline{B_1},\cdots,\overline{B_m}$ satisfying $(\tau_1,\cdots,\tau_n) \prec (\rho_1,\cdots,\rho_n)$. Furthermore, if $\tau_i \prec \rho_i$, the type variables in ρ_i do not appear in ρ_j, where $1 \leq i \neq j \leq n$, and for each variable x in C, $\lceil \overline{p(t_1,\cdots,t_n)} \rceil | x \cong \lceil \overline{B_1} \rceil | x \cong \cdots \cong \lceil \overline{B_m} \rceil | x$,

type (4): $m>0$ and there is a typed clause for C, $p(t_1:\tau_1,\cdots,t_n:\tau_n) \leftarrow \overline{B_1},\cdots,\overline{B_m}$ satisfying $(\tau_1,\cdots,\tau_n) \cong (\rho_1,\cdots,\rho_n)$ and at least one of B_i is not *strongly well-typed*. Furthermore, for each variable x in C, $\lceil p(t_1,\cdots,t_n) \rceil | x \cong \lceil \overline{B_1} \rceil | x \cong \cdots \cong \lceil \overline{B_m} \rceil | x$.

Definition 9. A predicate p/n is defined *weakly well-typed*, if at least one of the clauses whose heads are p/n is *weakly well-typed* and other clauses are *strongly well-typed*. A polymorphic typed Prolog program P is defined *weakly well typed*, if each predicate in P is *weakly well-typed* or *strongly well-typed*.

By the Definition 9, a weakly well-typed program P is also strongly well-typed, if it has only strongly well-typed predicates. Now let us explain the weakly well-typed clauses taking examples. First, for instance, the unit clause $append([1],[2],[1,2])$ is weakly well-typed because its *MGT* is:

$$append([1{:}int]{:}list(int),[2{:}int]{:}list(int),[1{:}int,2{:}int]{:}list(int)),$$

and $append(list(int),list(int),list(int))$ is an instance of the declaratrion $append(list(\alpha),list(\alpha),list(\alpha))$. The *append* program becomes weakly well-typed after the ground unit clause is added, while the original program is strongly well-typed.

For explaining the second and third cases, let us take an example of "likes" relation [Clocksin 84].

Example 2.
func john: → man
func alfred: → man
func edward: → man
func wine: → alcohol
func beef: → food
pred likes: α,β
likes(john, X) ← likes(X,wine),
likes(X,beef)
likes(john, wine) ←
likes(alfred, wine) ←
likes(alfred, beef) ←
likes(edward, X) ←
likes(cat, mouse) ← .

The unit clause $likes(edward,X)$ belongs to the second type, because a typed clause $likes(edward{:}man,X:\gamma)$ satisfies the relation $(man,\gamma) \preceq (\alpha,\beta)$ and there is no type variable which appears both in α and β. The clause,

likes(john, X) ← likes(X,wine), likes(X,beef)

belongs the the third type. Hence, this program is *weakly well-typed* and we can omit the type consistency checking in proving *permissible queries* defined below. A clause of the final case is defined *weakly well-typed* because even though it is very much similar to a *strongly well-typed* clause, it loses the property of the *strongly well-typing* owing to the weakly well-typed predicates in the body part.

Definition 10. A conjunction of predicates A_1, $\cdots$, A_k is defined *permissible* if it is well-typed and for a variable x appearing in a weakly well-typed predicate A_i, it holds that $\lceil \overline{A_i} \rceil | x \preceq \lceil \overline{A_j} \rceil | x$, for $1 \leq i \neq j \leq k$.

For instance, a conjunction of predicate,

append(X,Y,Z), member(U,Z)

is *permissible*, because $\lceil \overline{append(X,Y,Z)} \rceil | Z \preceq \lceil \overline{member(U,Z)} \rceil | Z$, with respect to the following *member* program.

Example 3.
pred member: γ,list(γ)
member(X,[X|Y]).
member(X,[Y|Z]) ← member(X,Z)

Note that a well-typed query with respect to a strongly well-typed program is also permissible because there is no weakly well-typed predicate. Before proving the soundness of weakly well-typing, we need a definition and some lemmas.

Definition 11. A typed term $t{:}\tau$ is said to have *the most general type with respect to* (w.r.t.) ρ if $\tau \preceq \rho$ and for another typed term $t{:}\tau'$ such that $\tau' \preceq \rho$, then $\tau' \preceq \tau$.

Lemma 1. Let $p(\rho_1,\cdots,\rho_n)$ be the type declaration of p/n and $p(t_1{:}\tau_1,\cdots,t_n{:}\tau_n)$ be the *MGT* of

$p(t_1, \cdots, t_n)$ satisfying $(\tau_1, \cdots, \tau_n) \preceq (\rho_1, \cdots, \rho_n)$. If there is i such that $\tau_i \prec \rho_i$ and the type variables in ρ_i do not appear in ρ_j for $i \neq j$, then $t_i{:}\tau_i$ has the most general type w.r.t. ρ_i.

Lemma 2. If a typed term $t{:}\tau$ has the most general type with respect to ρ, then for a typed term $s{:}\sigma$ satisfying $\sigma \preceq \rho$, t is unifiable with s iff $t{:}\tau$ is unifiable with $s{:}\sigma$.

Theorem 2. Suppose that a well-typed query $A_1 \equiv p(s_1, \cdots, s_n)$ is unified with a *strongly well-typed* clause $C \equiv A \leftarrow B_1, \cdots, B_m$ with the mgu θ = mgu(A_1, A). Then for each variable $x \in Dom(\theta)$, $\lceil \overline{A_1} \rceil | x \cong \lceil \overline{A_1\theta} \rceil | x\theta$ if every function is type-preserving.

Lemma 3. Suppose that the type of the predicate p/n is declared as $p(\rho_1, \cdots, \rho_n)$. If $p(\tau_1, \cdots, \tau_n)$ and $p(\sigma_1, \cdots, \sigma_n)$ are generic instances of the type declaration, then the following property holds: $\tau_1, \cdots, \tau_n$ are unifiable with $\sigma_1, \cdots, \sigma_n$, respectively iff $p(\tau_1, \cdots, \tau_n)$ is unifiable with $p(\sigma_1, \cdots, \sigma_n)$.

In the following lemma, $t[y]$ denotes that a term t contains a variable y.

Lemma 4. Suppose that every function is type preserving. Let $A_1, \cdots, A_k$ be well-typed and x be a variable appearing in it. Supposing that $\lceil \overline{A_i} \rceil | x \preceq \lceil \overline{A_j} \rceil | x$, for a substitution $\theta = \{x \leftarrow t[y]\}$, it holds that $\lceil \overline{A_i\theta} \rceil | y \preceq \lceil \overline{A_j\theta} \rceil | y$, if $(A_1, \cdots, A_k)\theta$ is well-typed.

Now, let us prove the soundness of the extended notion of well-typing, so called, *permissible queries do not go wrong for a weakly well-typed program.*

Theorem 3. (Soundness of weakly well-typing) If a program is weakly well-typed and every function is type preserving, a *permissible* query does not spawn an ill-typed resolvent in the refutation procedure.

5 Well-typing with Mode Information

This section presents a method which furthers optimization if mode information is available in polymorphic typed Prolog. Section 4 develops an optimization technique based on the notion of weakly well-typing. However, there are still some programs which can not be optimized by this method because they are not always type correct for every permissible query. This section attempts to optimize the programs which are partially type correct on the assumption that some arguments of predicates are *ground terms.* For instance, let us consider the following program.

Example 4.
func []: $\rightarrow$ list(α)
func . : α,list(α) $\rightarrow$ list(α)
pred flatten: α,list(β)
pred atom: α
flatten([],[]) $\leftarrow$
flatten([H|T],O) $\leftarrow$ flatten(H,Y),
flatten(T,Z),
append(Y,Z,O)
flatten(X,[X]) $\leftarrow$ atom(X)
Example 1 is appended here.

This program is not strongly well-typed, nor weakly well-typed, owing to the second *flatten/2* clause. That is, the clause is not *strongly well-typed* since the MGT of the first argument is $list(\gamma)$ which is less than the type declaration α. Moreover, it is not *weakly well-typed* because $\lceil \overline{flatten([H \mid T], O)} \rceil | T \equiv list(\alpha)$ is not a renaming of $\lceil \overline{flatten(T, O)} \rceil | T \equiv \alpha$. A type error ocurrs if the first argument of $flatten/2$ is given as a variable X, unified with the second $flatten/2$ clause, and the variable T in the second predicate $flatten(T, Z)$ of the body part is instantiated to an element which is not of list type.

However, if the first argument of $flatten/2$ predicate is always given as a *ground term*, this type error can be avoided and the program does not invoke any run-time type error. The following definition and theorem prove this property. Before proving it, let us define the modes of a program.

Definition 12. A set of modes for a program P is a set of functions $M = \{ m_{p/n} \mid p \in P\}$ such that $m_{p/n} : \{1, \cdots, n\} \rightarrow \{I, O\}$ for every predicate $p/n \in P$. A query $A_1, \cdots, A_m$ is said to be *mode consistent with respect to* M if for each predicate $A_i \equiv p(t_1, \cdots, t_n)$, the following condition holds:

if $m_{p/n}(i) = I$, then t_i is *ground.*

For simplicity, a mode of a predicate is often denoted in the argument positions of the predicate. For instance, if a mode of the *append* is defined as:

$m_{append/3}(1) = I$, $m_{append/3}(2) = I$, $m_{append/3}(3) = O$,

it is also represented as $append(I, I, O)$.

Definition 13. Suppose that the type of the predicate p/n is declared as $p(\rho_1, \cdots, \rho_n)$. A clause $p(t_1, \cdots, t_n) \leftarrow B_1, \cdots, B_m$ is defined *strongly well-typed with respect to (w.r.t.) a set of modes* M if there is a typed clause $p(t_1{:}\tau_1, \cdots, t_n{:}\tau_n) \leftarrow \overline{B_1}, \cdots, \overline{B_m}$ such that $(\tau_1, \cdots, \tau_n) \preceq (\rho_1, \cdots, \rho_n)$ and the following conditions are satisfied:

1. if there is B_j such that $m_{B_j}(l) = I$, then for each variable $x \in Var$(the l-th argument of B_j), there exists t_k such that $x \in Var(t_k)$ and $m_{p/n}(k) = I$,

2. at least one of the following conditions should be satisfied:

   (a) $(\tau_1, \cdots, \tau_n) \cong (\rho_1, \cdots, \rho_n)$,

   (b) $m_{p/n}(i) = I$, and ρ_i contains all type variables occurring in the declared type $\rho_1, \cdots, \rho_n$,

   (c) for each i satisfying $\tau_i \prec \rho_i$, $m_{p/n}(i) = I$.

Condition 1 assures that the body predicates are *mode consistent* if the head predicate is *mode consistent*. Condition 2 includes the cases that the type information can be omitted in the unification of a subgoal and the head part. Condition 2-(b) was already found out in [Hanus 89b] and proved to be useful for higher-order programming.

Definition 14. A predicate p/n is defined *strongly well-typed w.r.t. a set of modes M*, if each clause whose head is p/n is strongly well-typed w.r.t. M. Furthermore, a polymorphic typed Prolog program P is defined *strongly well-typed* w.r.t. a set of modes M if every predicate in P is strongly well-typed w.r.t. M.

For instance, the predicate $flatten/2$ is strongly well-typed w.r.t. the mode

$$\{flatten(I, O),\ append(O, O, O),\ atom(O)\}.$$

Theorem 4. (Soundness of strongly well-typing w.r.t. modes) If a program is strongly well-typed w.r.t. a set of modes M, and every function is type preserving, then any *mode consistent query w.r.t. M* does not spawn an ill-typed query in the refutation procedure.

The notion of well-typing associated with modes is useful for programs which are partially type correct. If modes are given in these programs, the criterion in Definition 13 is able to judge if they are type correct w.r.t. the modes or not. Even though modes are not specified in these programs, we can find out the modes which guarantee the well-typedness of them.

However, one problem here is that the condition 1 in Definition 13 is rather strong, because the groundness conditions of the predicates in the body part should always be satisfied by the unification of the head predicate. Some of these conditions may be satisfied in the resolution steps of the former predicates in the body part. Hence, if this criterion is mixed with a method which could predict if the result of the refutation of a goal is ground or not, we are able to make this condition weaker.

6 Conclusions

Polymorphic typed Prolog is a language which offers polymorphic type system in Prolog framework. This type extension could make Prolog as a specification language suitable for programming in-the-large, whereas it may raise inefficiency problem associated with type consistency checking in unification. This inefficiency can be reduced by optimizing the cases that the type checking is redundant. Mycroft and O'Keefe developed a criterion, called well-typing, to detect the programs in which the type checking is redundant in the whole refutation procedure of a well-typed query. Hanus also presented an optimization technique which avoids the run-time type checking if a program satisfies his type general condition. However these conditions are too strong, so that only a limited number of programs can be optimized by these techniques.

This paper, first, attempts to introduce another weaker notion of well-typing, called *weakly well-typing*, and show that the new notion promotes the optimization in a wider range of programs. *Strongly well-typing* assures that a program is type correct with respect to any well-typed queries, whereas *weakly well-typing* only guarantees that a program is type correct for *permissible* queries. Secondly, this paper extends the notion of well-typing to the one associated with modes. This notion is useful for programs which are not type correct, but are free from type errors if some arguments of predicates are given as ground terms. Hence, optimization is possible for these programs on the mode assumption of some predicates. This paper also develops a criterion which finds out the type correct programs with respect to given modes.

Further works remain to be done. As mentioned in Section 5, the condition 1 of Definition 13 is rather strong. One of the further works is to study an abstract interpretation method which is able to predict if the result of the refutation of a goal is ground or not, for relieving this condition.

This paper is only concerned with the prescriptive approach. Sometimes, users may want to omit some type declarations which are, they feel, unnecessary. In this sense, the prescriptive system is too restrictive to satisfy this kind of user's desire. To meet this requirement, this paper may be extended to include a type inference system to find out the undeclared types.

Acknowledgements

We are indebted to IIAS-SIS, Fujitsu for supporting us to do this work. We are also grateful to Dr. Jiro Tanaka for many useful discussions.

References

[Clocksin 84] W.F. Clocksin and C.S. Mellish, Programming in Prolog, Second Edition, Springer-Verlag, 1984.

[Damas 82] L. Damas and R. Milner, Principle type-schemes for functional programs, Proceedings of the 9th POPL, 1982, pp. 207-212.

[Dietrich 88] R. Dietrich and F. Hagl, A Polymorphic Type System with Subtypes for Prolog, Proceedings of the Second European Symposium on Programming, 1988, pp. 79-93.

[Hanus 89a] M. Hanus, Horn Clause Programs with Polymorphic Types: Semantics and Resolution, Proceedings of TAPSOFT'89, Lecture Note in Computer Science 352, pp. 225-240.

[Hanus 89b] M. Hanus, Polymorphic Higher-Order Programming in Prolog, Proceedings of the Sixth International Conference, 1989, pp. 382-397.

[Jacobs 90] D. Jacobs, Type Declarations as Subtype Constraint in Logic Programming, Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation, 1990, pp. 165-173.

[Kanamori 84] T. Kanamori and K. Horiuchi, Type Inference in Prolog and Its Applications, ICOT TR-095, 1984.

[Lloyd 84] J.W. Lloyd, Foundation of Logic Programming, Springer-Verlag, 1984.

[Milner 78] R. Milner, A Theory of Type Polymorphism in Programming, in: Journal of Computer and System Science, Vol. 17, 1978, pp. 348-375.

[Mishra 84] P. Mishra, Toward a Theory of Types in Prolog in: Proceedings of IEEE International Symposium on Logic Programming, 1984, pp. 289-298.

[Shin 90] D.W. Shin, Toward Optimal Type Checking for The Polymorphic Type System of Prolog, submitted for pubilcation.

[Mycroft 84] A. Mycroft and R.A. O'Keefe, A Polymorphic Type System for Prolog, in:Artificial Intelligence, Vol. 23, 1984, pp. 295-307.

[Yardeni 87] E. Yardeni and E. Shapiro, A Type System for Logic Programs, Concurrent Prolog: collected papers, 1987, pp. 211-244.

[Zobel 87] J. Zobel, Derivation of Polymorphic Types for Prolog Programs, Proc. of Fourth International Conference on Logic Programming, 1987, pp. 817-838.