**ECRC/TR/91/LP-61**
11 July 1991

# Polymorphically Typed Logic Programs

Eyal Yardeni, Thom Fruehwirth, Ehud Shapiro

**Abstract**

We introduce polymorphically typed logic programs, an integration of a polymorphic type system with logic programs. The first-order notion of predicates is extended to parametric predicates, which are parameterized by types. The type system accommodates both subtypes and parametric polymorphism. It unifies features of value-based and name-based approaches. The denotation of a typed logic program is given by its type completion, a transformation that incorporates explicit type conditions into a parametric logic program. The result of the transformation is a restricted form of a HiLog program. We give fixpoint semantics to our language (actually to full HiLog). We define a notion of well-typing, which relates type declarations for predicates in the program to an approximation of the denotation of the type completed program. We present a type-checking algorithm for verifying that a program is indeed well-typed. Finally we discuss some extensions to the type system.

**European Computer-Industry**
**Research Centre GmbH**
Arabellastr. 17
D-8000 München 81
Germany

A short version of this paper was published at the 8th Intl Conf on Logic Programming, MIT Press, June 1990
The address of the first and third author is Department of Applied Mathematics
and Computer Science, The Weizmann Institute of Science, Rehovot 76100, Israel
Work of the second author was done while visiting the Weizmann Institute of Science
and while at the Department of Computer Science, Technical University of Vienna, Austria

# 1 Introduction

Types in logic programming have been used for approximating untyped programs by *type inference* [BJ88, Fru89b, HJ90, Mis84, PR89, Zob87] as well as for verifying well-formedness of typed programs by *type checking* [Bru82, DH88, Fru90, Han89, Nai87, MO83, Smo88, YS87a]. In the type inference approach, a type is inferred that is a set of atoms that covers (hence approximates) the success set of the logic program. In the type checking approach, types are named syntactic objects defined by the user to restrict the denotation of a predicate. This paper is about type checking.

Type systems for logic programming languages are usually considered to be a tool put on top of the language. The type language and the typed language are only loosely coupled. In the approach taken in our paper, we suggest a tight integration of polymorphic types and logic programs into a single language, polymorphically typed logic programs. This approach makes use of experiences with the type systems of [YS87a, YS89] and [Fru89a, Fru90]. Its main contributions are:

- A truly polymorphic type language that supports subtypes as well as types as parameters.
- Polymorphic predicates which are parameterized by types (including type variables).
- Types that can be freely incorporated into the program as type conditions.
- Head-only type completion as a way to define semantics for polymorphically typed logic programs in terms of the semantics of logic programs.
- An intuitive notion of well-typing based on an approximation of the denotation of the program.
- Algorithms for type checking polymorphically typed logic programs.
- As a by-product, fixpoint semantics for the language HiLog [CKW89].

The idea of types in logic programming traces back as early as 1982, when [Bru82] suggested to add 'useful redundancy' to logic programs in the form of mode and type declarations. In their classical paper, [MO83] adopted the outlook of [Mil78] that 'well-typed programs do not go wrong', where well-typing is a well-formedness condition stated through inference rules. They proved for their ML-style type system, that if the program and the initial goal are well-typed, then variables at each resolution step can only be instantiated to terms which are allowed by their types. Unfortunately, this property does not hold for extensions of Mycroft's and O'Keefe's type checker like subtypes [DH88, Fru89a, Fru90] or type declarations for higher-order predicates [Han89]. Smolka [Smo88] developed a typed predicate logic with subtyping relation. Based on it he defines syntax and semantics of relational programs computing on polymorphically order-sorted types.

More recently, [YS87a, YS89] introduced the intuitive idea that a program is well-typed if its type declarations approximate the denotation of the program. It has been shown in [YS87a] that abstraction of the fixpoint semantics provides a useful and intuitive characterization of types and well-typing in logic programs. This choice of the notion of well-typing is the main factor that makes this work different from others that were mentioned. In contrast to the work above about subtyping, types are identified with the sets that they represent. A type is a subset of another type if and only if its denotation is a subset of the other's one. Hence subtyping comes for free and is completely unrestricted. However, as the type language lacks parametric polymorphism, type dependencies between different arguments cannot be expressed and the definition of types with similar structure turns out to be cumbersome.

This and other recent work on type checking for logic programs has either shown or acknowledged the need for flexible type languages that support *inclusion polymorphism* (the possibility to define subtypes) as well as *parametric polymorphism* (the possibility to define types with parameters that take types as arguments). In other words, *full* polymorphism in the type language is essential for the practicality of a typed language. This paper brings full polymorphism to logic programs themselves, not only to their types. Parametric predicates enable a programming style that can take full advantage of type information. Calling a parametric predicate with specific types creates an instance of the predicate with the appropriate type conditions enforced on its arguments.

From the beginning the developments regarding types in logic programming to a great extent parallel those of types in functional programming (e.g. [MO83] was influenced by [Mil78]). Bearing this in mind, parametric predicates can be seen as analogous to polymorphic functions in typed lambda calculus [GLT89, Rey85], where beta reduction is replaced by resolution.

The move from non-parameterized programs to parameterized programs complicates the syntax, the semantics, the well-typing definition, and the type checking algorithms. We show that the syntax and the semantics are decidable while admitting a powerful type language. Well-typing a predicate definition guarantees that in all contexts the predicate will be well-typed, i.e. all its ground type instances, over any vocabulary, are well-typed. The type checking algorithm combines an extension of the one in [YS87a] and a novel one that checks inclusion of parameterized types.

A typed logic program includes *type definitions* for function symbols and *type declarations* for predicates. The syntax of logic programs is extended to a simple instance of HiLog [CKW89] in order to accommodate parametric predicates. We stay in first-order logic with this move, because HiLog provides a second-order syntax while retaining a first order-semantics [CKW89].

For example, the polymorphic version of the predicate `append(L1,L2,L3)` that is true if the concatenation of the lists `L1` and `L2` yield the list `L3` can be defined as:

*List*($\tau$) ::= [] ; [ $\tau$ |*List*($\tau$)].

*procedure append*($\tau$)(*List*($\tau$),*List*($\tau$),*List*($\tau$)).

`append`($\tau$)`([],L,L).`

`append`($\tau$)`([X|L1],L2,[X|L3])` $\leftarrow$ `append`($\tau$)`(L1,L2,L3).`

The type definition defines a recursive, polymorphic type *List*($\tau$) to be either the empty list `[]` or the cons-structure `[_|_]`, where the first argument is a term of type $\tau$ and the second argument is a term of type *List*($\tau$). The type of the elements of the list is left open as a type variable $\tau$. The type declaration declares that the predicate `append`($\tau$) takes lists of type $\tau$ in its three arguments, where the type of the elements, $\tau$, is left unspecified. Note that the type variable $\tau$ is added to the predicate as *type argument.* This expresses that the predicate is polymorphic in the type of the elements of the lists it deals with. In other words, `append`($\tau$) can concatenate two lists as long as their elements have a common type $\tau$. For example, given the following type definitions for natural and positive numbers:

*Nat* ::= *0* ; *s*(*Nat*).

*Pos* ::= *s*(*Nat*).

we can call the predicate as

$\leftarrow$ `append(`*Nat*`)([s(0),s(s(0))],[0],L).`

and as

$\leftarrow$ `append(`*Pos*`)([s(0),s(s(0))],[0],L).`

The first goal will succeed as expected as *Nat* is a common type for all the elements of the list, while the second goal is ill-typed and should fail, as the only element of the list `[0]` is not of type *Pos*.

The semantics of a typed logic program is defined in terms of the semantics of its *type completion*, a transformation from typed programs to untyped programs. Transformation of many-sorted and order-sorted logic languages to first-order languages is a well-known way to define semantics for the sorted language in terms of the unsorted language [Llo87]. A standard kind of type completion has been used in [YS87a, XW88, Nai87]. It turned out that this standard type completion has drawbacks, as it well-types predicates with non-intuitive type declarations. In [YS89] this problem was overcome by suggesting a modified type completion which just restricts *head-only variables* (i.e., variables that appear only in the head of a clause) to their types, while standard type completion restricts all arguments of the head. In this paper, we substantially extend the idea of *head-only type completion* to polymorphically typed logic programs.

For example, the head-only type completion of the program for `append(`$\tau$`)` is:

```
list(τ)([]).

list(τ)([X|L]) ← τ(X),list(τ)(L).

append(τ)([],L,L) ← list(τ)(L).

append(τ)([X|L1],L2,[X|L3]) ← τ(X),append(τ)(L1,L2,L3).
```

Type definitions have been translated into logic clauses and the type declaration has been incorporated in the predicate via explicit *type conditions.* In our proposal, the run-time type checks performed by the type conditions are not seen as annoyance, but as integral part of the typed logic programming language. Note that exactly the head-only variables are the reason why predicates may be under-specified and these are the ones for which type conditions are added. For example, the first clause of the usual untyped `append` program does not make any statement about the types of the second and the third arguments. Hence a goal like ← `append([],a,a)` succeeds — a fact not necessarily intended by the programmer. In a well-typed program, however, such a goal will fail, because `a` is not of type *List(*$\tau$*)* for any $\tau$.

The rest of the paper is organized as follows: In the next section we define the syntax of typed logic programs. In section 3 we give the semantics of typed logic programs in terms of their type completion. Section 4 defines well-typedness of programs, describes an algorithm for type checking and gives a justification for type completion. Section 5 shortly discusses some extensions of the type language. Section 6 concludes the paper. Finally, the appendix presents the detailed proofs of the two theorems in the paper.

# 2 Syntax

In this section we define the syntax of type definitions and type declarations as well as of parametric and typed logic programs. Related terminology is introduced.

We assume a first order language $\mathcal{L}$ with a fixed set of (data) variables, function symbols (including constants) and predicates. We also assume a type language $\mathcal{T}$ with a fixed set of *type variables*[1] and *type constructors* (including *type constants*). We use small Greek letters e.g. $\tau_1, \tau_2, \ldots$ for type variables and slanted strings beginning with capital letters e.g. *Int, Nat, List*$/1, \ldots$ for type constructors.

As usual, a *term* is either a variable or a function symbol of arity $n$ applied to $n$ terms ($n \geq 0$).

Analogously, a *type* is either a type variable or a type constructor of arity $n$ applied to $n$ types ($n \geq 0$). A *type constant* is a type constructor of arity 0. A type is *ground* if it does not contain type variables. An example of a ground type is *BTree(List(Nat))* and a non-ground type is *BTree(*$\tau$*)*.

We extend predicates to type parametric predicates by adding *type arguments*, i.e. arguments that can take types.

**Definition**. A *type parametric predicate* of arity $n$ is a first order logic predicate $p$ of arity $n$ parameterized by zero, one or more types $T_i$ written as:

$\mathtt{p(T_1,T_2,\ldots,T_m)}$ $\mathtt{(m \geq 0)}$

Predicates with a different number of type arguments are considered to be different.

[1] Also called type parameters in the literature

## 2.1 Type Definition

Type definitions and type declarations are given in the BNF-style syntax of [YS87a] which we extend with type arguments to express parametric polymorphism.

**Definition**. A *type definition* for a type constructor $c$ of arity $n$ is of the form

$c(\tau_1,\ldots,\tau_n) ::= f_1(T_1^1,\ldots,T_{n_1}^1);\ldots;f_k(T_1^k,\ldots,T_{n_k}^k)\ (n, n_j \geq 0,\ k \geq 1)$

where the left hand side (l.h.s.) is a type constructor $c$ applied to $n$ type variables $\tau_i$. The right hand side (r.h.s.) is a union of $k$ function symbols $f_j$ applied to types as arguments.

The type definition states that the type constructor $c$ is polymorphic in $n$ parameters and that its denotation contains terms of the form $f_j(t_1^j,\ldots,t_{n_j}^j)$ provided their arguments are of the type specified.

**Example**. Given the following type definitions:

$Evenlist(\tau_1,\tau_2) ::= []\ ;\ [\tau_1 \mid Oddlist(\tau_1,\tau_2)].$

$Oddlist(\tau_1,\tau_2) ::= [\tau_2 \mid Evenlist(\tau_1,\tau_2)].$

$BTree(\tau) ::= empty\ ;\ tree(\tau, BTree(\tau), BTree(\tau)).$

where the type $Evenlist(\tau_1,\tau_2)$ includes all lists of the form $[t_1,\ldots,t_{2m}]$, such that $t_{2i}$ is of type $\tau_2$ and $t_{2i+1}$ is of type $\tau_1$ for $(0 \geq i \geq 2m)$. The type $BTree(\tau)$ is the type of all binary trees whose nodes can take terms of type $\tau$.

In the rest of the paper, we assume *well-formed* type definitions, which fulfill the following requirements:

1. For each type constructor, there is exactly one type definition.
2. A type variable that occurs on the r.h.s. of a type definition also occurs on its l.h.s.
3. The type variables on the l.h.s. of a type definition are different from each other.
4. The function symbols on the r.h.s. of a type definition are different from each other.
5. The type constructors in a program can be assigned a well-formed ranking (as defined below).

**Explanation**. Requirement 2 means that there are no local type variables. This requirement is called type transparency in [HT90] and type preserving in [Han89]. All type variables are assumed to be universally quantified over all possible types. We do not think that the advantages of local type variables would justify the resulting complications for both the theory of the typed language and the implementation of the algorithms.

For the same reasons we restrict the left-hand sides to be most general, i.e. (requirement 3) to be type constructors applied to different type variables (and not arbitrary types).

Not requiring different function symbols on the r.h.s. of a type definition (requirement 4) would increase the complexity of the algorithms involving types and would not add to the readability of such type definitions, although it might be more precise. However, we do allow overloading, i.e. a function symbol can occur in more than one type definition.

We require the type constructors appearing in a program to have a ranking. The ranking condition (requirement 5) slightly limits the use of recursion in type definitions so that a decidable class of types is obtained.

**Definition**. Given a program $P$, a *ranking* $r$ is a function from the type constructors in $P$ to positive integers. It is well-formed if the rank of each type constructor appearing on the r.h.s. of the type definition is non-increasing whenever it is applied to type variables only and decreasing otherwise, i.e. it is well-formed if for every type constructor $c$ in the program whose type definition contains a type constructor $c_i$ on the r.h.s. the following holds:

- $r(c) \geq r(c_i)$, if $c_i$ is applied to type variables,
- $r(c) > r(c_i)$, otherwise.

**Example**. Below we have that $r(List) = 1$, $r(Matrix) = 2$ and $r(Crazy) = 3$.

*Matrix*($\tau$) ::= [] ; [*List*($\tau$) | *List*(*List*($\tau$)].

*Crazy*($\tau$) ::= $foo$(*List*(*Matrix*(*List*($\tau$)))).

However, a type like

*Cf*($\tau$) ::= [] ; [$\tau$ | *Cf*(*T*($\tau$))].

is not well-formed, because the ranking of *Cf* on the r.h.s. must be less than the ranking of *Cf* on the l.h.s., which is a contradiction. As the denotation of types can be represented by automata, equivalence of such types would correspond to the open problem of the equivalence of deterministic push-down automata [Sol78].

## 2.2 Type Declaration

Type declarations relate predicates to the type of their arguments.

**Definition**. A *type declaration* for a predicate $\mathtt{p}(\tau_1, \ldots, \tau_m)$ of arity $n$ has the form

*procedure* $p(\tau_1,\ldots,\tau_m)(T_1,\ldots,T_n)$. $(n, m \geq 0)$

where $\tau_i$ are type variables and $T_j$ are types. The type declaration states that the predicate $\mathtt{p}(\tau_1, \ldots, \tau_m)$ is polymorphic in $m$ parameters and that its $j$-th argument has type $T_j$. Again, we only consider well-formed type declarations, where the $\tau_i$'s are different and where any type variable that occurs in $T_i$ is one of $\tau_1,\tau_2,\ldots,\tau_m$. Also, for each predicate, there is exactly one type declaration.

**Example**. A predicate `tree_to_list`($\tau$) that flattens a binary tree into a list could be declared as:

*procedure tree_to_list*($\tau$)(*BTree*($\tau$),*List*($\tau$)).

Note that a type declaration can be viewed as a kind of type definition for the type *procedure*. This interpretation unifies typing for functions and predicates and therefore enables the straightforward typing of meta-programs [Fru90].

## 2.3 Typed Logic Programs

The syntax of parametric and typed logic programs is an extension of first-order logic programs in that type arguments are added to predicates and in that types (including type variables) may be used as unary predicates. This extension can be easily accommodated in HiLog [CKW89] as a proper subset of it.

The alphabet of a HiLog language contains disjoint sets of logical symbols and variables. Unlike other logic languages, HiLog does not distinguish between predicates and function symbols.

**Definition**. A *HiLog term* is either a logical symbol, a variable or $t(t_1, \ldots, t_n)$, where $t, t_1, \ldots, t_n$ are HiLog terms. An *atomic HiLog formula* (*atom*) is a HiLog term. A *HiLog program* is a finite set of HiLog clauses, which are built from HiLog atoms in the usual way [Llo87].

**Example**. The following is a notorious HiLog term (atom):

```
Z(b,f(Y,c)(d,X))(r,X(s,t),t(t))
```

Within the HiLog language, we can now define parametric and typed logic programs.

**Definition**. A *parametric logic program* is a (finite) set of parametric clauses of the form

$\mathtt{H} \leftarrow \mathtt{B}_1, \mathtt{B}_2, \ldots, \mathtt{B}_n. \quad (n \geq 0)$

where the head atom H is of the form

$\mathtt{p}(\tau_1, \tau_2, \ldots, \tau_k)(\mathtt{t}_1, \mathtt{t}_2, \ldots, \mathtt{t}_r) \quad (k, r \geq 0)$

where $\tau_i$ are different type variables and $\mathtt{t}_j$ are terms. And where $\mathtt{B}_i$ are the body atoms of the form

$\mathtt{q}(\mathtt{T}_1, \mathtt{T}_2, \ldots, \mathtt{T}_l)(\mathtt{t}'_1, \mathtt{t}'_2, \ldots, \mathtt{t}'_m) \quad (l, m \geq 0)$ or $\mathtt{T}(\mathtt{t}')$

where T and $\mathtt{T}_i$ are types whose type variables are contained in $\{\tau_1, \ldots, \tau_k\}$ and $\mathtt{t}'$ and $\mathtt{t}'_j$ are terms.

**Definition**. A *typed predicate definition for* $p$ is the set that consists of a type declaration for $p$ and of clauses whose head atom has the predicate $p$. There are no local type variables in the clauses.

We assume that type variables are universally quantified over types and that their scope is a predicate definition. Data variables are treated as usual: They are universally quantified over terms and their scope is a program clause.

**Example**. The predicate `tree_to_list` can be defined as

*procedure tree_to_list($\tau$)(BTree($\tau$),List($\tau$)).*

```
tree_to_list(τ)(empty,[]).

tree_to_list(τ)(tree(X,LT,RT),[X|L3]) ←

        tree_to_list(τ)(LT,L1),

        tree_to_list(τ)(RT,L2),

        append(τ)(L1,L2,L3).
```

Here is an example that uses a specific instance of the predicate:

```
sum_tree(T,N) ← tree_to_list(Nat)(T,L), sum_list(L,N).
```

**Definition**. A *typed logic program* is a union of typed predicate definitions and type definitions for each type constructor occurring in the program.

# 3 Semantics

In this section we define semantics of typed logic programs in terms of their head-only type completion, which transforms a typed logic program into a parametric logic program. As parametric logic programs are instances of HiLog programs, we give fixpoint semantics for HiLog programs. We employ the well-known notion of types based on the concept of tuple-distributivity [Mis84]. A tuple-distributive abstraction of the fixpoint operator [YS87a] allows us to relate types to the denotation of a parametric logic program.

## 3.1 Fixpoint Semantics of HiLog

By defining fixpoint semantics of HiLog programs we can give a denotation to type completed logic programs, as they form a sublanguage of HiLog. The fixpoint semantics is almost identical to the fixpoint semantics for first-order logic programs, but the original work on HiLog [CKW89] gave only proof-theoretic semantics for the language.

The *Herbrand universe*, $\mathcal{H}_L$, of a HiLog language, $L$ is the set of all ground HiLog terms that can be constructed from the logical symbols of $L$. Since terms are also atoms and vice versa, the *Herbrand base*, $\mathcal{B}_L$, is identical to $\mathcal{H}_L$.

It follows from Chen *et. al.* [CKW89] that a ground atom $G$ is in the *minimal Herbrand model* of a HiLog program $P$ if and only if there exists a refutation of $P \cup \{G\}$.

**Definition**. We define an operator $T_P$, which is syntactically identical to van Emden and Kowalski's [vEK76], except that it operates on another domain:

$T_P : 2^{\mathcal{B}_L} \rightarrow 2^{\mathcal{B}_L}$

$T_P(I) = \{A \in \mathcal{B}_L \mid A \leftarrow B_1, B_2, \ldots, B_n \ll_L C,\ C \in P, \text{ and } B_1, B_2, \ldots, B_n \in I\}.$

where $A \ll_L B$ means that $A$ is a ground instance of $B$ over $\mathcal{B}_L$.

Following Lloyd [Llo87] it can easily be shown that the least fixpoint of $T_P$ is identical to the minimal Herbrand model (denotation) of the program.

### 3.2 Head-Only Type Completion

Head-only type completion translates type definitions into program clauses and adds type conditions, which are derived from type declarations, to the body of existing clauses. This idea was proposed in [YS89] for a non-parametric type system. In subsection 4.1 we will define well-typing of a typed program as a condition on the type completed program.

Type definitions are translated into unary HiLog predicates, a representation suggested in [Fru89a].

**Definition**. The *type completion of a type definition* of the form

$T ::= f_1(T_1^1, \ldots, T_{n_1}^1); \ldots; f_k(T_1^k, \ldots, T_{n_k}^k)$

is a set of parametric clauses defining the *type predicate* $T$:

For each $f_i(T_1^i, \ldots, T_{n_i}^i)$ a *type clause* of the form

$\mathtt{T}(f_i(\mathtt{X}_1, \ldots, \mathtt{X}_{n_i})) \leftarrow \mathtt{T}_1^i(\mathtt{X}_1), \ldots, \mathtt{T}_{n_i}^i(\mathtt{X}_{n_i}).$

is constructed, where $X_i$ are different data variables.

**Definition**. The *head-only type completion* of a typed program is a parametric program, called head-only type completed program, which is defined as follows:

- Each type definition is translated into a type predicate as defined above.
- Each type declaration is removed, as it is incorporated into the program clauses.
- Each program clause $(\mathtt{H} \leftarrow \mathtt{B})$ is transformed into a clause $\mathtt{H} \leftarrow \mathtt{T}_1(\mathtt{X}_1), \mathtt{T}_2(\mathtt{X}_2), \ldots, \mathtt{T}_n(\mathtt{X}_n), \mathtt{B}.$ $(n \geq 0)$. A type condition $\mathtt{T}_i(\mathtt{X}_i)$ is added for each head-only occurrence of a data variable $\mathtt{X}_i$, where $\mathtt{T}_i$ is the type of $\mathtt{X}_i$ according to the type declaration

We use the algorithm for induced types (specified in subsection 4.2) to find out the right $\mathtt{T}_i$. If there is no induced type for a variable, then no type condition is added for that variable[2].

[2] The clause is ill-typed in this case, as we will see in subsection 4.2

The run-time effect of type conditions is that of type checking the instances of head-only variables, restricting the denotation of the transformed predicates. Together with a definition of well-typing this guarantees that ill-typed goals can never succeed as they either will not unify with any clause head or will not pass the type conditions.

**Example**. The head-only type completion of the program for `tree_to_list` is:

BTree($\tau$)(empty).

BTree($\tau$)(tree(X,Y,Z)) $\leftarrow$ $\tau$(X),Btree($\tau$)(Y),BTree($\tau$)(Z).

list($\tau$)([]).

list($\tau$)([X|L]) $\leftarrow$ $\tau$(X),list($\tau$)(L).

tree_to_list($\tau$)(empty,[]).

tree_to_list($\tau$)(tree(X,LT,RT),[X|L3]) $\leftarrow$

$\tau$(X),

tree_to_list($\tau$)(LT,L1),

tree_to_list($\tau$)(RT,L2),

append($\tau$)(L1,L2,L3).

append($\tau$)([],L,L) $\leftarrow$ list($\tau$)(L).

append($\tau$)([X|L1],L2,[X|L3]) $\leftarrow$ $\tau$(X),append($\tau$)(L1,L2,L3).

**Remark**. Type completed logic programs can be easily translated into first-order logic programs by applying the following transformation:

- Each type atom[3] of the form `T(t)`, where `T` is a type and `t` is a term, is replaced by `type(T,t)`, where `type` is a new binary predicate.
- Each other atom of the form p($\tau_1$,$\tau_2$,...,$\tau_k$)(t$_1$,t$_2$,...,t$_r$) is replaced by pred(p($\tau_1$,$\tau_2$,...,$\tau_k$),t$_1$,t$_2$,...,t$_r$), where *pred* is a set of new predicate symbols of arity $r+1$ ($k, r \geq 0$).

## 3.3 Regular Types

One way to approximate the denotation of a program uses the notion of tuple-distributivity [Mis84]. The tuple distributive closure of a set is an approximation of the initial set where argument dependencies are eliminated.

**Definition**. (Adapted from [HJ90]). Let S be a set of ground HiLog terms. The *tuple distributive closure* (or cartesian closure) of S, written $\alpha(S)$, is recursively defined as:

$$\alpha(S) := \{c : c \text{ is a constant, } c \in S\} \cup \{f(t_1,\ldots,t_n) : t_i \in \alpha(f^{-1}_{n,i}(S)), 1 \leq i \leq n\}.$$

where $f$ ranges over the HiLog function symbols in $S$ and $f^{-1}_{n,i}(S)$ is defined as $\{t_i : f(t_1,\ldots,t_i,\ldots,t_n) \in S\}$. The recursive application of $\alpha$ takes care of nested terms.

A set $S$ is *tuple distributive* if $S = \alpha(S)$.

**Example**. Let $S$ be {f(a,b),f(c,d)} then $\alpha(S)$ = {f(a,b),f(a,d),f(c,b),f(c,d)}.

[3] I.e. atoms in type predicates and atoms appearing as type conditions in program clauses

For the practical purpose of type checking, we are interested in algorithms for intersection, tuple-distributive union and equivalence of (the sets denoted by) types, which do not exist for tuple-distributive languages[4]. Therefore we restrict our attention to a well-known subclass of tuple-distributive sets, the so-called regular sets. Algorithms for the intersection, union and equivalence of regular languages [AHU74] are known.

**Definition**. A set of terms is *regular* iff it is the denotation of a type constant in some set of type definitions.

Note that the type definition of a type constant is in standard BNF-syntax. Their type completion are regular unary predicate logic programs [Yar87b], they can also be represented by finite automata [AHU74] (suggested by [Mis84]) as well as deterministic root-to-frontier tree automata [Tha73]. Note that regular sets are closed under intersection and tuple-distributive union (but not under union).

**Definition**. Given a program $P$. The *denotation of a ground type $T$ in $P$* is defined by
$[\![T]\!]_P = \{t \mid T(t)$ is true in the type completion of the type definitions in $P\}$.

**Theorem 1**. The denotation of a ground type is regular.

**Proof Outline**. Given a type completed program $P$, we show with the help of the ranking condition that only a finite number of type ground instances of type definitions is needed to compute the denotation of any ground type. By mapping each ground type occurring in these instances to new type constants, we arrive at a finite set of type definitions. ∎

**Lemma**. Given a program $P$ and a ground type $T$, then the set of types occurring in any derivation of $T$ is finite.

**Proof Outline**. (Full proof in the appendix). The idea is to construct a finite set of types $S_T$, that includes $T$, such that every derivation step applied to any element in the set yields only elements in the set, i.e. the set is closed under derivation. ∎

The importance of the theorem lies in the fact that it supports the basic idea for tackling general types in type checking. The idea is to look at all their ground instances, so that we get rid of type variables.

### 3.4 Abstraction of $T_P$

.

We recall the approximation of $T_P$ introduced in [YS87a], which relates types to the denotation of typed logic programs.

**Definition**. Let $P$ be a program and $S$ a set of atoms. Then:

$$T_P^\alpha(S) \stackrel{def}{=} \alpha(T_P(S))$$

We say that $T_P^\alpha(S)$ is *inferred by $P$ relative to $S$*.

**Lemma**. $T_P^\alpha$ is monotonic and continuous over the set of tuple distributive elements in $2^{B_L}$.

**Proof**. Analogous to the proof for $T_P$ in standard logic programming theory [Llo87]. ∎

The least fixpoint of a program $T_P^\alpha$ is $T_P^\alpha \uparrow \omega$, and is denoted by $[\![P]\!]^\alpha$.

**Example**. Let $P$ be the program:

```
p(a,b).              q(e) ← p(a,d).

p(c,d).
```

Then

[4]For example, the set of all prime numbers represented by natural numbers is tuple-distributive

$[\![P]\!] = \{p(a,b), p(c,d)\}$.

$\alpha([\![P]\!]) = \{p(a,b), p(a,d), p(c,b), p(c,d)\} = \alpha(T_P \uparrow 1) = T_P^{\alpha} \uparrow 1$

$[\![P]\!]^{\alpha} = \{p(a,b), p(a,d), p(c,b), p(c,d), q(e)\} = T_P^{\alpha} \uparrow 2$

Note that $[\![P]\!] subseteq\ \alpha([\![P]\!])\ subseteq\ [\![P]\!]^{\alpha}$ holds for every program $P$.

# 4 Well-Typing and Type Checking

We extend the notion of well-typing of [YS89], which relates the type declarations of a typed program to its fixpoint approximation, to parametric predicates. Type checking determines whether a program is well-typed by its type declarations. We suggest an algorithm for type checking of typed logic programs. Based on these results we give a justification for head-only type completion.

## 4.1 Well-Typing

A type completed program is well-typed with respect to its type definitions and declarations if the fixpoint approximation of each predicate yields exactly the types it is declared for.

**Definition**. An atom is *type ground* if it does not contain type variables. A predicate is *type ground* if all its type arguments are ground types.

**Example**. The atom `append(`$Nat$`)([1,2],[X,4],L)` is type ground and so is the predicate `append(`$Nat$`)`, while the atom `append(`$List(\tau)$`)([[a],[b]],[[c]],[[a],[b],[c]])` is not.

**Notation**. Let $S$ be a subset of the Herbrand base and $p(T_1, \ldots, T_k)$ be a type ground predicate of arity $n$. Then $S/_{p(T_1,\ldots,T_k)} = \{p(T_1, \ldots, T_k)(t_1, \ldots, t_n) \mid p(T_1, \ldots, T_k)(t_1, \ldots, t_n) \in S\}$, where the $t_i$'s are terms and the $T_j$'s are ground types.

**Definition**. The *denotation of a type declaration procedure* $p(\tau_1, \ldots, \tau_k)(T_1, \ldots, T_n)$ in a program $P$ is the set

$\{p(U_1, \ldots, U_k)(t_1, \ldots, t_n) \mid t_j \in [\![T_j[\tau_1 \to U_1, \ldots, \tau_k \to U_k]]\!]_P,\ U_1, \ldots, U_k$ are ground types$\}$.

**Notation**. Let $S_P$ be the denotation of all the type declarations and ground types in a typed program $P$.

**Notation**. Let $T$ be a ground type. Then $def(T)$ is the set of type definitions needed to define $T$.

We first define well-typing for type ground predicates and then for arbitrary parametric predicates by considering all their type ground instances.

**Definition**. Let $P$ be a typed logic program and $P_H$ its head-only type completion. A type ground predicate $p(T_1, \ldots, T_m)$ $(m \geq 0)$ in $P$ is *well-typed by* some set of ground atoms $S$, if:

1. $T^{\alpha}_{P_H}(S)/_{p(T_1,\ldots,T_m)} = S/_{p(T_1,\ldots,T_m)}$.
2. For every clause $C$ of the predicate $p(\tau_1, \ldots, \tau_m)$, $T_{\{C\}}(S)/_{p(T_1,\ldots,T_m)} \neq \emptyset$.

We introduce condition 2 for practical purposes. This non-emptiness condition requires each clause to contribute at least one atom to the approximation. Clauses violating this condition are called *useless* [YS87a]. The type-checker should report on useless clauses, as they usually indicate a programming error.

We would like the composition of well-typed programs to yield a well-typed program. This is important for large programs, which are usually structured by some module system, as it allows each module to be type checked separately. In other words, this means that our definition of well-typing should be independent of the programs vocabulary and types. This is reflected in considering arbitrary types in the following definition.

**Input**: A type ground predicate $p(T_1,\ldots,T_k)$ and a regular set $S$

**For** each clause $C$ in the definition of $p(T_1,\ldots,T_k)$ **do**
  *Type* each body atom computing the induced types.
  **If** failed **then** *fail*.
  **For** each variable $X$ that appears in $C$ **do**
    *Intersect* all the induced types for $X$ getting the type of $X$
  **od**
  *Construct* the inferred type $T_C$ denoting $T^{\alpha}_{\{C\}}(S)$.
  **If** $T_C = \phi$ **then** *fail*.
**od**
Find the tuple-distributive closure of the union of the $T_C$'s.
**If** the result is equal to $S/_{p(T_1,\ldots,T_k)}$ **then** *succeed* **else** *fail*.

Figure 4.1: A type checking algorithm for type ground predicates

**Definition**. A predicate $p(\tau_1,\ldots,\tau_k)$ $(k \geq 0)$ in a typed program $P$ is *well-typed* if for every non-empty ground types $T_1,\ldots,T_k$ (not necessarily in the program's vocabulary), $p(T_1,\ldots,T_k)$ is well-typed by $S_{P\cup\{def(T_1),\ldots,def(T_k)\}}$. A typed logic program $P$ is *well-typed* if and only if each of its predicates is well-typed.

**Definition**. A ground atom is called *proper* if all its type arguments are types. Given a set $S$ of ground atoms we define the function $pro : 2^{\mathcal{B}_L} \rightarrow 2^{\mathcal{B}_L}$ to be

$pro(S) = \{A \in S \mid \text{A is proper}\}$

**Lemma**. Given a well-typed typed logic program $P$ and its head-only type completion $P_H$. Then $S_P = pro(T^{\alpha}_{P_H}(S_P))$.

**Proof**. Obvious, as a logic program is a union of predicate definitions. ∎

In subsection 4.4 we show that whenever the program is well-typed under head-only type completion, it is well-typed under standard type completion, as the two different type completions produce equivalent programs in that case.

## 4.2 Type Checking Type Ground Predicates

We check whether a type ground predicate $p(T_1,\ldots,T_k)$ in a typed program $P$ is well-typed by a regular set $S$ according to the algorithm in figure 4.1, which is basically the one considered in [YS87a]. The differences between the algorithm presented in [YS87a] and the one presented here are that the latter checks well-typing predicate-wise and type checks type ground predicates while the former type checks all the predicates in the entire program simultaneously and is restricted to first order predicates.

**A Type Checking Algorithm**. Let $P$ be a head-only type completed program and $S$ a regular set. We check whether a type ground predicate, $p$, is well-typed by $S$ as follows: For each clause $C$ of $p$ we compute $T^{\alpha}_{\{C\}}(S)$, the maximal set of atoms that can be inferred relative to $S$. The set $T^{\alpha}_{\{C\}}(S)$ will be represented as the denotation of the so-called *inferred type of* $C$, written $T_C$. Then we find the tuple-distributive union of all inferred types (provided their denotation is non-empty) and check if it equals $S$.

Next we show how to type an atom and then how to obtain the inferred type $T_C$.

*Typing* an atom means to find out if its arguments are in the types prescribed by the declaration for the corresponding predicate and to find the so-called *induced* types for the variables in the atom. Induced types are the maximal types such that all ground instances of the atom (where each variable is substituted to a ground term of its induced type) are in the denotation of its type declaration. The algorithm is based on the idea that given a term and its expected type we can find out the types of its subexpressions, as a function symbol with a given type together with its type definition determines the types of its arguments.

**Subalgorithm for Induced Types**. The algorithm takes an atom and the type declarations and type definitions in a program $P$ and returns the induced types for the variables occurring in the atom or fails if the atom cannot be typed.

- If $\mathtt{p(T_1,\ldots,T_k)(t_1,\ldots,t_n)}$ is an atom and there is a type declaration *procedure* $p(\tau_1,\ldots,\tau_k)(U_1,\ldots,U_n)$, then recursively type each $\mathtt{t}_i$ with its induced type $U_i\,[\tau_1 \mapsto \mathtt{T}_1, \ldots, \tau_k \mapsto \mathtt{T}_k]$.
- If $\mathtt{f(t_1,t_2,\ldots,t_n)}$ is a term with type $t(T_1,\ldots,T_k)$ and there is a type definition $t(\tau_1,\ldots,\tau_k)$ $::= f_1(T_1^1,\ldots,T_{n_1}^1);\ldots;f(U_1,\ldots,U_n);\ldots;f_m(T_1^m,\ldots,T_{n_m}^m)$ then recursively type each $\mathtt{t}_i$ with its induced type $U_i\,[\tau_1 \mapsto \mathtt{T}_1, \ldots, \tau_k \mapsto \mathtt{T}_k]$.
- If $X$ is a variable with type $T$, then $T$ is the induced type for $X$.
- Otherwise fail.

**Remark**. Remember that type variables are regarded to be universally quantified. Since no non-variable term can be of all possible types, typing a non-variable term with a type variable will fail.

By typing each body atom in $C$ we compute the induced type of each occurrence of each variable. Since occurrences of the same variable in the body mean that they are to be instantiated to the same terms, we have to intersect the induced types to get the maximal type of the variable (call it *the body type of the variable*).

Let the inferred type $T_C$ of a clause $C$ be the inferred type of its head constructed according to the following algorithm.

**Subalgorithm for inferred type**. To construct the inferred type $T$ of a term (or atom) $f(t_1,\ldots,t_n)$ define $T ::= f(T_1,\ldots,T_n)$ and for each $i$, $i \leq 1 \leq n$ do:

- If $t_i$ is a variable, then let $T_i$ be the body type of the variable.
- If $t_i$ is not a variable, then let $T_i$ be a new type and recursively construct the inferred type $T_i$ of $t_i$.

Note that this construction parallels the function 'build' in [PR89].

**Claim**. Let $S$ be a regular set. Then $p(T_1,\ldots,T_k)$ is well-typed by $S$ iff the type checking algorithm succeeds.

**Proof**. Analogous to the proof in [YS89] ∎

## 4.3 Type Checking Parametric Predicates

Now we introduce a type checking algorithm for parametric predicates.

**Theorem 2**. A parametric predicate $p(\tau_1,\ldots,\tau_k)$ in a typed program $P$ is well-typed iff

1. $p(T_1,\ldots,T_k)$ is well-typed in the program $P \cup \{def(T_1),\ldots,def(T_k)\}$, where $T_1 ::= c_1,\ldots,T_k ::= c_k$, and $c_1,\ldots,c_k$ are different constants not in the program's vocabulary.
2. For every clause $C$ in the head-only type completion of the procedure of $p(\tau_1,\ldots,\tau_k)$, we have that $T^{\alpha}_{\{C\}}(S_{P\cup\{def(U_1),\ldots,def(U_k)\}})/_{p(U_1,\ldots,U_k)} \subseteq S_{P\cup\{def(U_1),\ldots,def(U_k)\}}/_{p(U_1,\ldots,U_k)}$ for any set of ground types $\{U_1,\ldots,U_k\}$.

**Proof Outline**. (Full proof is presented in the appendix).

$(\Longrightarrow)$ Trivial from the definition of well-typing.

($\Longleftarrow$) In the appendix we prove that condition 1 implies that $T_P^{\alpha}(S_{P\cup\{def(U_1),\ldots,def(U_k)\}})/_{p(U_1,\ldots,U_k)} \supseteq S_{P\cup\{def(U_1),\ldots,def(U_k)\}}/_{p(U_1,\ldots,U_k)}$ for any set of ground types $\{U_1,\ldots,U_k\}$. Together with condition 2 the theorem follows. ∎

Next we present a simplification of the decision algorithm for the second condition. It is enough to check that the induced type of each variable occurrence in the head contains the body type of the variable. Since the body type of a variable is the intersection of the induced types of its occurrences in the body, we suggest a polymorphic intersection algorithm. This algorithm is a straightforward extension of the standard algorithms for intersection of regular sets in order to deal with type variables. Whenever a type variable is encountered, the algorithm stops and returns the intersection as a partial result.

**Algorithm for Intersection.**

Let $F ::= f_1(F_1^1,\ldots,F_{n_1}^1);\ldots;f_m(F_1^m,\ldots,F_{n_m}^m);r_1(R_1^1,\ldots,R_{l_1}^1);\ldots;r_k(R_1^k,\ldots,R_{l_k}^k)$.

and $G ::= g_1(G_1^1,\ldots,G_{j_1}^1);\ldots;g_i(G_1^i,\ldots,G_{j_i}^i);r_1(S_1^1,\ldots,S_{l_1}^1);\ldots;r_k(S_1^k,\ldots,S_{l_k}^k)$.

be type definitions generalized by intersections with type variables, where all function symbols are different. Each argument on the right-hand side is of the form $\tau_1\ldots\cap\ldots\tau_r\cap T_1\ldots\cap\ldots T_q$. If the right-hand side is empty, we call it an *empty type*, whose denotation is the empty set.

The intersection of $F$ and $G$ denoted by $F\cap G$ is defined by the rules generated by:

$F\cap G ::= RHS$

where $RHS' = r_1(R_1^1\cap S_1^1,\ldots,R_{l_1}^1\cap S_{l_1}^1);\ldots;r_k(R_1^k\cap S_1^k,\ldots,R_{l_k}^k\cap S_{l_k}^k)$

and RHS is the same as RHS' after removing expressions $r_u(R_1^u\cap S_1^u,\ldots,R_{l_u}^u\cap S_{l_u}^u)$ that include empty types.

**Algorithm for Condition 2.** Let $T$ be the induced type for a variable in a clause head and let $R$ be the body type of that variable. Note that the inclusion has to hold for all instances of the type variables, as they are all-quantified over types. So it is sufficient to find a single counterexample of a term with a particular type substitution.

In general, the type definitions supporting the body types $R$ include intersections with type parameters as introduced in the intersection algorithm, while the definition supporting the declared types $T$ are of the usual syntax.

In the following, let the definitions of $F$ and $G$ be of the form:

$F ::= f_1(F_1^1,\ldots,F_{n_1}^1);\ldots;f_m(F_1^m,\ldots,F_{n_m}^m);r_1(R_1^1,\ldots,R_{l_1}^1);\ldots;r_k(R_1^k,\ldots,R_{l_k}^k)$.

$G ::= g_1(G_1^1,\ldots,G_{j_1}^1);\ldots;g_i(G_1^i,\ldots,G_{j_i}^i);r_1(T_1^1,\ldots,T_{l_1}^1);\ldots;r_k(T_1^k,\ldots,T_{l_k}^k)$.

The algorithm succeeds if the inclusion holds. The algorithm is defined over a set of inclusion constraints $I$ as follows:
initialize $q := 0$ and $I_0 := \{R\subseteq T\}$.
repeat

let $I := I_q$

for each element $(\tau_1\ldots\cap\ldots\tau_r\cap T_1\ldots\cap\ldots T_q\subseteq G')\in I_q$ do

if $G'$ is a type variable $\tau$ then **case 1**

if $\tau$ is one of the $\tau_1,\ldots,\tau_r$ then *do nothing* **case 1.1**

else *fail* **case 1.2**

else **case 2**

if $q = 0$ then *fail* **case 2.1**

assume $F\theta = T_1 \ldots \cap \ldots T_q$ and $G\sigma = G'$

if $m > 0$ then *fail* **case 2.2**

else let $I := I \cup \{R_1^1\theta \subseteq T_1^1\sigma, \ldots, R_{l_1}^1\theta \subseteq T_{l_1}^1\sigma, \ldots, R_1^k\theta \subseteq T_1^k\sigma, \ldots, R_{l_k}^k\theta \subseteq T_{l_k}^k\sigma\}$ **case 2.3**

od

let $I_{q+1} := I$ and $q := q + 1$

until $Iq + 1 = I_q$ (up to renaming of variables).
*succeed.*

**Observations**

1. The algorithm terminates because the number of different inclusions is bounded. The proof is similar to the proof that the number of different types occurring in any derivation of a ground type is bounded.

2. Case 1.1 is a tautology.

3. In case 1.2, usually, a ground type substitution for the inclusion that makes it a invalid can be constructed. Let $c$ be a new constant, define $C := c$ and substitute $\tau$ by $C$. Now $C$ does not intersect with any term in the program. Hence each term derived from the l.h.s. is not covered by the r.h.s., i.e. each term in the l.h.s. is a counterexample.

   Note however, that $\tau$ can occur on the l.h.s. inside some $f_i$ or $r_i$ or as one of the $\tau_i$ in the inclusion from which the current inclusion was derived. In the first case we might have to add some values to the type $C$ for $\tau$, because the intersection with $C$ would produce the empty type otherwise. Then the l.h.s. might reduce to the empty set making the counterexample invalid. In the second case, we essentially have to do the same to ensure that a term that contains the counterexample term as a subterm can be constructed. Now if these values added to $C$ cover all possible counterexamples of terms, then the inclusion still holds.

   We take the l.h.s of the inclusion and substitute $\tau$ to $C$ and all other $\tau_1, \ldots, \tau_r$ to a type *All*, which includes all terms in the program vocabulary. Clearly, all counterexamples must be in the denotation of the resulting l.h.s. Now, if we can find for each term $t$ in the l.h.s. a type of the form $C \cap F'$ in the l.h.s. of the inclusions, the current inclusion was derived from or which are derived from the current inclusion, such that $[\![F']\!]_P = t$, then we have no valid counterexample, as we have to add every counterexample to $C$ to avoid empty types and the empty l.h.s.

   But note that this invalidation of the counterexamples can only happen if the l.h.s. denotes a finite (sufficiently small) set and if $\tau$ occurs in intersections with types that denote single values. Overall, it is extremely unlikely in practice, that all counterexamples get invalidated. We only found very contrived examples, which shouldn't be in a good program anyway. Therefore the inclusion algorithm does not verify that there is indeed a valid counterexample, although this would be possible at additional cost.

   **Example**. Let *A ::= a*, *AB ::= a;b*, *C(τ) ::= f(AB,τ)*, *D(τ)∩FA ::= f(τ∩A,A)*, and *D(τ)∩FAB ::= f(τ∩AB,A)* be type definitions, and assume an inclusion $D(\tau) \cap FA \subseteq C(\tau)$. After one iteration we get $A \subseteq \tau$, but $A$ is not a valid counterexample as $\tau \cap A$ implies that for all type substitutions for $\tau$, that do not include $A$, the empty type results, which makes the l.h.s. empty. But note that in $D(\tau) \cap FAB \subseteq C(\tau)$ we can construct a valid counterexample by substituting $\tau$ to $B$ where $B ::= b$ with $\{f(b,a)\} \not\subseteq \{f(a,b), f(b,b)\}$.

4. In case 2.1 we can give the type $C$ as above to all the type variables to show invalidation of the inclusion.

5. Clearly, the inclusion doesn't hold for case 2.2.

6. It is easy to see that in case 2.3 the initial inclusion is a logical consequence of the new inclusions produced. In other words, if the above initial inclusion does not hold, then there exists a type substitution such that one of the derived inclusions does not hold.

## 4.4 A Justification for Head-Only Type Completion

Using the standard way of transforming many-sorted and order-sorted logic to first-order logic [Llo87], programs with typings can be transformed to untyped programs. This transformation has been used in [YS87a, XW88, Nai87] and has been called relativization in [Coh89, Obe62]. We will refer to this transformation as *standard type completion* as opposed to the *head-only type completion* we introduced in subsection 3.2.

**Definition**. The *standard type completion of a type declaration* of the form

*procedure* $p(\tau_1,\ldots,\tau_k)(T_1,T_2,\ldots,T_n)$

is a clause of the form

$\texttt{procedure(p(}\tau_1,\ldots,\tau_k\texttt{)(X}_1\texttt{,}\ldots\texttt{,X}_n\texttt{))} \leftarrow \texttt{T}_1\texttt{(X}_1\texttt{),}\ldots\texttt{,T}_n\texttt{(X}_n\texttt{).}$

where $X_i$ are different data variables and `procedure` is a new predicate of arity 1.

**Definition**. The *standard type completion* of a typed program is a parametric program defined as follows:

- Each type definition is translated into type predicates as in the head-only type completion.
- Each type declaration is translated as defined above.
- Each program clause $\texttt{(H} \leftarrow \texttt{B)}$ is transformed into a clause $\texttt{(H} \leftarrow \texttt{procedure(H),B)}$.

In the program resulting from this transformation, it is verified by the added type condition $procedure(H)$ that each argument of the head is in the type declared for it. Recall that in subsection 4.1 we defined well-typing of a typed program as a condition on some kind of type completion of the program. Practical experience with the type system described in [YS87a] showed that requiring well-typing under the standard type completion is too weak, as it well-types programs one would like to consider ill-typed.

**Example**. The program

*Foo*$(\tau)$ ::= [] ;$[\tau]$.

*procedure append*$(\tau)$(*Foo*$(\tau)$,*Foo*$(\tau)$,*Foo*$(\tau)$).

$\texttt{append(}\tau\texttt{)([],L,L).}$

$\texttt{append(}\tau\texttt{)([X|L1],L2,[X|L3])} \leftarrow \texttt{append(}\tau\texttt{)(L1,L2,L3).}$

is well-typed under the standard type completion, but ill-typed under our head-only type completion. In the first case, the inferred type of the first clause is $\texttt{append(}\tau\texttt{)([],[];[}\tau\texttt{],[];[}\tau\texttt{])}$, and of the second clause it is $\texttt{append(}\tau\texttt{)([}\tau\texttt{],[];[}\tau\texttt{],[}\tau\texttt{])}$. The tuple distributive union gives $\texttt{append(}\tau\texttt{)([];[}\tau\texttt{],[];[}\tau\texttt{],[];[}\tau\texttt{])}$ which is the declared type. In the second case, we get a bigger inferred type for the second clause, $\texttt{append(}\tau\texttt{)([}\tau\,|\texttt{[];[}\tau\texttt{]],[];[}\tau\texttt{], [}\tau\,|\texttt{[];[}\tau\texttt{]]])}$, as only head-only variables are restricted by the head-only type completion. This results in a type that is bigger than the declared type, hence the program is ill-typed under head-only type completion.

Next we show that for a program that is well-typed under head-only type completion, these two kinds of type completion are equivalent.

**Claim**. Given a typed logic program $P$. Let $P_H$ be its head-only type completion and $P_S$ be its standard type completion. Then $pro([\![P_H]\!]) = pro([\![P_S]\!])$ provided $P_H$ is well-typed by $S$.

**Proof**. By definition of tuple distributive closure, $pro(T_{P_H}(S)) \subseteq S$ as $pro(T^{\alpha}_{P_H}(S)) = S$ according to lemma of subsection 4.1. Further we have that $\forall I.\ T_P(I) \cap S = T_{P_S}(I) \subseteq T_{P_H}(I) \subseteq T_P(I)$. This implies

$pro([\![P_S]\!]) \subseteq pro([\![P_H]\!]) \subseteq S$. By induction we prove that $pro(T_{P_S} \uparrow n) \supseteq pro(T_{P_H} \uparrow n)$, which proves the claim.

For $n = 0$, the claim trivially holds.

Assume true for $n$ and prove for $n + 1$.

$pro(T_{P_S} \uparrow n+1) = pro(T_{P_S}(T_{P_S} \uparrow n)) = pro(T_{P_S}(pro(T_{P_S} \uparrow n))) \supseteq pro(T_{P_S}(pro(T_{P_H} \uparrow n))) = pro(T_{P_S}(T_{P_H} \uparrow n)) = pro(T_P(T_{P_H} \uparrow n) \cap S) \supseteq pro(T_{P_H}(T_{P_H} \uparrow n) \cap S) = pro(T_{P_H} \uparrow n+1) \cap S = pro(T_{P_H} \uparrow n+1)$. ∎

# 5 Extensions of the Type Language

We discuss some extensions of the type system, namely basic types, the universal type and non-canonical type definitions.

In a practical type systems, *basic types* like *Integer* and *Constant*[5] are necessary to describe types of built-in predicates correctly. Basic types partition the terms in the Herbrand universe into equivalence classes. Hence the intersection of two different basic types is always empty. We assume that membership in a basic type is determined by the outermost function symbol. Under these assumptions, the algorithms for intersection, tuple-distributive union and equivalence can be easily extended to handle basic types.

With the help of the *universal type* named *All*, we can well-type predicates with their most general types[6]. Any term in the Herbrand Universe of the language is of type *All*. If a term is of type *All*, then the induced types for all occurrences of variables in the term is *All* as well. For practical reasons, the union of all basic types should be equivalent to *All*.

For example, we can well-type a type argument free `append` by:

*procedure append(List(All),All,All).*

But note that the following type declaration is ill-typed

*procedure append(All,All,All).*

as the first argument of `append` cannot take *all* terms, but only terms with the function symbols `[]` and `[_|_]`.

Another extension is to allow *non-canonical* type definitions and type declarations to gain flexibility. *Subtypes* can be utilized for more natural type definitions:

*Parent ::= Father ; Mother.*

Moreover, this extension allows for more general recursive types without introducing decidability problems:

$Evenlist(\tau_1, \tau_2) ::= [] \; ; [\tau_1, \tau_2 \,|Evenlist(\tau_1, \tau_2)]$.

Non-canonical type definitions can be transformed to canonical ones by introducing auxiliary types and unfolding subtypes. In particular, non-canonical recursive types can be transformed to mutual recursive types:

$Evenlist(\tau_1, \tau_2) ::= [] \; ; [\tau_1 \,|Oddlist(\tau_1, \tau_2)]$.

$Oddlist(\tau_1, \tau_2) ::= [\tau_2 \,|Evenlist(\tau_1, \tau_2)]$.

[5] Excluding numbers
[6] These are the *structural types* which are inferred in [PR89]

# 6 Conclusion

We suggested an integration of polymorphic type systems and logic programming languages into polymorphically typed logic programming. The typed logic programming language extends Horn clauses to parametric Horn clauses by parameterizing predicates with types. The resulting language is seen as a simple instance of HiLog. The denotation of a typed logic program is given by its head-only type completion, in which head-only variables are restricted to their declared types. We extended the notion of well-typedness of [YS89] to type parametric logic programs. We proposed algorithms for type checking type ground and parametric predicates.

Future work will tackle type inference within this framework and investigate a richer type language. In addition, global analysis to eliminate redundant type conditions is required to increase the practicality of the approach.


# Acknowledgements

We would like to thank Mark Miller for his ideas, Moshe Vardi, Frank Pfenning and Marc Dennecker for useful discussions, as well as Kim Marriot and anonymous referees for helpful comments.


# Appendix

Here we present the proofs for theorem 1 and theorem 2 of the paper.

## Theorem 1

For theorem 1, we have to prove the following lemma.

**Lemma**. Given a program $P$ and a ground type $T$, then the set of types occurring in any derivation of $T$ is finite.

**Proof**. The idea is to construct a finite set of types $S_T$, that includes $T$, such that every derivation step applied to any element in the set yields only elements in the set, i.e. the set is closed under derivation.

In the following we will ignore the arguments in the derivations of type atoms, i.e. we will only look at the predicates.

There exists a minimal ranking function $r$, s.t. for every type constructor $c$ in the program, whose definition contains a type constructor $c_i$ on the r.h.s., the following holds:

1. $r(c) \geq r(c_i)$, if $c_i$ is applied to type variables only,
2. $r(c) > r(c_i)$, otherwise;
3. $max(range(r))$ is minimal.

Note that all type constructors on the r.h.s. except those applied directly to type variables have a rank strictly less than the rank of the constructor defined. Because of the minimality condition, the range of $r$ includes all numbers from 1 to $max(range(r))$.

**Definition**. The *depth* of a type $T$ is defined as follows:

- $depth(T) = 0$, if $T$ is a type variable,
- $depth(T) = 1$, if $T$ is a type constant,

- $depth(c(T_1,...T_n)) = max(depth(T_i)) + 1$.

Let $d = max(depth(T_i))$, where $T_i$ are the types occurring on the r.h.s. of the type definitions in $P$. Let $k = depth(T)$ and $R = max(range(r))$. Let $S_T$ be the set of all ground types up to depth $R*d+k$ built from the type constructors in $P$, such that all type constructors appearing between depth $(g-1)*d+k+1$ and depth $g*d+k$ are at most of rank $(R-g)$, where $1 \leq g \leq R$. Note that if some type belongs to $S_T$, then also all its subterms belong to $S_T$.

Take an element of $S_T$ of the form $c(T_1,...T_n)$. Let $g$ be the number such that $(g-1)*d+k+1 \leq depth(c(T_1,...T_n)) \leq g*d+k$. Do one derivation step for $c(T_1,...T_n)$ using a type of depth $m$ from the r.h.s of the type definition for $c$ resulting in a type which has a depth of at most $m + depth(c(T_1,...T_n)) - 1$. By definition of $d$, $0 \leq m \leq d$. The resulting type is built of type constructors from the definition of $c$ and of the $T_i$'s. This implies that all the constructors at depth greater than $g*d+k$ are of rank less than the rank of $c$, i.e. $(R-g)$, and that all other constructors of depth less or equal to $g*d+k$ are either of rank less or equal $(R-g)$ or belong to one of the $T_i$'s. Also note that in the new type, the $T_i$'s (and their constructors) will occur at the same depth as in $c(T_1,...T_n)$. ■

## Theorem 2

**Theorem 2**. A parametric predicate $p(\tau_1, \ldots, \tau_k)$ in a typed program $P$ is well-typed iff

1. $p(T_1, \ldots, T_k)$ is well-typed in the program $P \cup \{def(T_1), \ldots, def(T_k)\}$, where $T_1 ::= c_1, \ldots, T_k ::= c_k$, and $c_1, \ldots, c_k$ are different constants not in the program's vocabulary.
2. For every clause $C$ in the head-only type completion of the procedure of $p(\tau_1, \ldots, \tau_k)$, we have that $T^{\alpha}_{\{C\}}(S_{P \cup \{def(U_1), \ldots, def(U_k)\}})/_{p(U_1,\ldots,U_k)} \subseteq S_{P \cup \{def(U_1),\ldots,def(U_k)\}}/_{p(U_1,\ldots,U_k)}$ for any set of ground types $\{U_1, \ldots, U_k\}$.

**Proof**

$(\Longrightarrow)$ Trivial from the definition of well-typing.

$(\Longleftarrow)$ We prove that condition 1 implies that $S_{P \cup \{def(U_1),\ldots,def(U_k)\}}/_{p(U_1,\ldots,U_k)} \subseteq T^{\alpha}_{P}(S_{P \cup \{def(U_1),\ldots,def(U_k)\}})/_{p(U_1,\ldots,U_k)}$ for any set of ground types $\{U_1, \ldots, U_k\}$. Together with condition 2 the theorem follows.

W.O.L.G. $c_1 \ldots, c_k$ do not appear in $\{def(U_1), \ldots, def(U_k)\}$

Let $P_U = P \cup \{def(U_1), \ldots, def(U_k)\}$ and $P_T = P \cup \{def(T_1), \ldots, def(T_k)\}$.

**Lemma 2**. Let $R$ be a type whose type variables are in the set $\{\tau_1, \ldots, \tau_k\}$. Then

$\alpha\left(\bigcup\{[\![R\eta]\!]_{P_T}\rho \mid u_j \in [\![U_j]\!]_{P_U},\ 1 \leq j \leq k\}\right) = [\![R\zeta]\!]_{P_U}$.
where $\rho$, $\eta$ and $\zeta$ are shorthand for the substitutions $[c_1 \mapsto u_1, \ldots, c_k \mapsto u_k]$, $[\tau_1 \mapsto T_1, \ldots, \tau_k \mapsto T_k]$ and $[\tau_1 \mapsto U_1, \ldots, \tau_k \mapsto U_k]$ respectively.

**Proof**. For a set of terms $S$, denote by $S^d$ the set of all terms in $S$ up to depth $d$. We prove the following claim (which entails lemma 2) by induction on $d$:

**Claim**. For any depth $d$ and for any type $R$ whose variables are in $\{\tau_1, \ldots, \tau_k\}$,

$\left(\alpha\left(\bigcup\{[\![R\eta]\!]_{P_T}\rho \mid u_j \in [\![U_j]\!]_{P_U},\ 1 \leq j \leq k\}\right)\right)^d = [\![R\zeta]\!]^d_{P_U}$.

**Proof**. If $R = \tau_i$ $1 \leq i \leq k$ then

$\alpha\left(\bigcup\{[\![R\eta]\!]_{P_T}\rho \mid u_j \in [\![U_j]\!]_{P_U},\ 1 \leq j \leq k\}\right) =$

$$\alpha\left(\bigcup\{[\![T_i]\!]_{P_T}\rho \mid u_j \in [\![U_j]\!]_{P_U},\ 1 \le j \le k\}\right) =$$

$$\alpha\left(\bigcup\{\{c_i\}\rho \mid u_j \in [\![U_j]\!]_{P_U},\ 1 \le j \le k\}\right) =$$

$$\alpha\left(\bigcup\{\{u_i\} \mid u_j \in [\![U_j]\!]_{P_U},\ 1 \le j \le k\}\right) =$$

$$\alpha([\![U_i]\!]_{P_U}) =$$

$$[\![U_i]\!]_{P_U} =$$

$$[\![R\zeta]\!]_{P_U}.$$

Assume that $R = c(R_1,\ldots,R_n)$, $n \ge 0$ and that the type definition for $c$ is of the form:

$c(\tau_1,\ldots,\tau_n) ::= f_1(R^1_1,\ldots,R^1_{n_1})\ ;\ \ldots\ ;\ f_l(R^l_1,\ldots,R^l_{n_l})\ ;\ a_1\ ;\ \ldots\ ;\ a_m.$

where $n_i \ge 1,\ 1 \le i \le l$ and $a_1,\ldots,a_m$ are constants.

Assume $d = 1$

$$\left(\alpha\left(\bigcup\{[\![R\eta]\!]_{P_T}\rho \mid u_j \in [\![U_j]\!]_{P_U},\ 1 \le j \le k\}\right)\right)^1 =$$

$$\{a_1,\ldots,a_m\} =$$

$$[\![R\zeta]\!]^1_{P_U}.$$

Assume the claim holds for $d$ and prove for $d+1$.

$$\left(\alpha\left(\bigcup\{[\![R\eta]\!]_{P_T}\rho \mid u_j \in [\![U_j]\!]_{P_U},\ 1 \le j \le k\}\right)\right)^{d+1} =$$

$$\left(\alpha\left(\bigcup_{1\le i\le l}\left\{f_i(t^i_1,\ldots,t^i_{n_i})\rho \;\middle|\; \begin{array}{l} t^i_o \in [\![R^i_o\eta]\!]_{P_T},\ 1 \le o \le n_i, \\ u_j \in [\![U_j]\!]_{P_U},\ 1 \le j \le k \end{array}\right\}\right)\right)^{d+1} \cup\ \{a_1,\ldots,a_m\} =$$

$$\left(\alpha\left(\bigcup_{1\le i\le l}\left\{f_i(t^i_1,\ldots,t^i_{n_i}) \;\middle|\; \begin{array}{l} t^i_o \in [\![R^i_o\eta]\!]_{P_T}\rho,\ 1 \le o \le n_i, \\ u_j \in [\![U_j]\!]_{P_U},\ 1 \le j \le k \end{array}\right\}\right)\right)^{d+1} \cup\ \{a_1,\ldots,a_m\} =$$

$$\left(\bigcup_{1\le i\le l}\left\{f_i(t^i_1,\ldots,t^i_{n_i}) \;\middle|\; t^i_o \in \alpha\left(\cup\left\{[\![R^i_o\eta]\!]_{P_T}\rho \;\middle|\; \begin{array}{l} u_j \in [\![U_j]\!]_{P_U}, \\ 1 \le j \le k \end{array}\right\}\right),\ 1 \le o \le n_i,\right\}\right)^{d+1} \cup \{a_1,\ldots,a_m\} =$$

$$\bigcup_{1\le i\le l}\left\{f_i(t^i_1,\ldots,t^i_{n_i}) \;\middle|\; \begin{array}{l} t^i_o \in \left(\alpha\left([\![R^i_o\eta]\!]_{P_T}\rho\right)\right)^d, 1 \le o \le n_i, \\ u_j \in [\![U_j]\!]_{P_U},\ 1 \le j \le k \end{array}\right\} \cup\ \{a_1,\ldots,a_m\} =$$

$$\bigcup_{1\le i\le l}\{f_i(t^i_1,\ldots,t^i_{n_i}) \mid t^i_o \in [\![R^i_o\zeta]\!]^d_{P_U},\ 1 \le o \le n_i\}\ \cup\ \{a_1,\ldots,a_m\} =$$

$$[\![R\zeta]\!]^{d+1}_{P_U}$$

Back to the theorem:

Let $\rho$ be a shorthand for $[c_1 \mapsto u_1, \ldots, c_k \mapsto u_k]$ and $C$ a shorthand for $H \leftarrow B$. The letter $\sigma$ denotes the substitution that replaces the type $T_i$ by $U_i$, $1 \leq i \leq k$. The symbol $B$ stands for a set of body atoms. In the following formulas, $\phi$ and $\theta$ are ordinary ground substitutions, i.e. their domains contain only variables.

1. $S_{P_U}/_{p(U_1,\ldots,U_k)} =$

2. $\alpha\left(\bigcup_{\substack{u_j \in [\![U_j]\!]_{P_U}\\ 1\leq j\leq k}} \left(S_{P_T}/_{p(T_1,\ldots,T_k)}\right)\sigma\rho\right) =$

3. $\alpha\left(\bigcup_{\substack{u_j \in [\![U_j]\!]_{P_U}\\ 1\leq j\leq k}} \left(T^{\alpha}_{P_T}(S_{P_T})/_{p(T_1,\ldots,T_k)}\right)\sigma\rho\right) =$

4. $\alpha\left(\bigcup_{\substack{u_j \in [\![U_j]\!]_{P_U}\\ 1\leq j\leq k}} \alpha\left(\bigcup_{C\in P} \left(T_{\{C\}}(S_{P_T})/_{p(T_1,\ldots,T_k)}\right)\right)\sigma\rho\right) =$

5. $\alpha\left(\bigcup_{C\in P}\bigcup_{\substack{u_j \in [\![U_j]\!]_{P_U}\\ 1\leq j\leq k}} \left(T_{\{C\}}(S_{P_T})/_{p(T_1,\ldots,T_k)}\right)\sigma\rho\right) =$

6. $\alpha\left(\bigcup_{C\in P}\bigcup_{\substack{u_j \in [\![U_j]\!]_{P_U}\\ 1\leq j\leq k}} \left(\{H\theta \,|\, B\theta \subseteq S_{P_T}\}/_{p(T_1,\ldots,T_k)}\right)\sigma\rho\right) \subseteq$

7. $\alpha\left(\bigcup_{C\in P}\bigcup_{\substack{u_j \in [\![U_j]\!]_{P_U}\\ 1\leq j\leq k}} \{H\theta\sigma\rho \,|\, B\theta \subseteq S_{P_T}\}/_{p(U_1,\ldots,U_k)}\right) \subseteq$

8. $\alpha\left(\bigcup_{C\in P}\bigcup_{\substack{u_j \in [\![U_j]\!]_{P_U}\\ 1\leq j\leq k}} \{H\theta\sigma\rho \,|\, B\theta\sigma\rho \subseteq S_{P_T}\sigma\rho\}/_{p(U_1,\ldots,U_k)}\right) \subseteq$

9. $\alpha\left(\bigcup_{C\in P}\bigcup_{\substack{u_j \in [\![U_j]\!]_{P_U}\\ 1\leq j\leq k}} \{H\theta\sigma\rho \,|\, B\theta\sigma\rho \subseteq S_{P_U}\}/_{p(U_1,\ldots,U_k)}\right) \subseteq$

10. $\alpha\left(\bigcup_{C\in P} \{H\phi \,|\, B\phi \subseteq S_{P_U}\}/_{p(U_1,\ldots,U_k)}\right) =$

11. $T^{\alpha}_{P}(S_{P_U})/_{p(U_1,\ldots,U_k)}$. ∎

# Bibliography


[AHU74] A. V. Aho, J. D. Hopcroft, and J. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.

[BJ88] M. Bruynooghe and G. Janssens, An instance of abstract interpretation integrating type and mode inferencing, *Proc. 5th International Conf. and symp. on Logic Programming*, Seattle, Washington, August 1988, pp. 669–683.

[Bru82] M. Bruynooghe, Adding redundancy to obtain more reliable and more readable Prolog programs, *Proc. First Intl. Logic Prog. Conf.*, 1982, pp. 129–138.

[CKW89] W. Chen, M. Kifer, and D. S. Warren, HiLog: a first-order semantics for higher-order logic programming constructs, *Proc. of the North American Conference on Logic Programming*, Cleveland, Ohio, October 1989, MIT Press, pp. 1090–1114.

[Coh89] A. G. Cohn, Taxonomic reasoning with many-sorted logics, *Artificial Intelligence Review* **3**:3, 1989, pp. 89–128.

[DH88] R. Dietrich and F. Hagl, A polymorphic type system with subtypes for Prolog, *Proc. 2nd European Symposium on Programming*, Springer LNCS, March 1988, pp. 79–93.

[Fru89a] T. Fruehwirth, A polymorphic type checking system for Prolog in HiLog, *6th Israel Conference on Artificial Intelligence and Computer Vision*, Israel, December 1989.

[Fru89b] T. W. Fruehwirth, Type inference by program transformation and partial evaluation, *Meta-Programming in Logic Programming*, MIT Press, 1989.

[Fru90] T. Fruehwirth, Using meta-interpreters for polymorphic type checking, *Meta90 Proc. Workshop on Meta-Programming in Logic*, K.U. Leuven, April 1990.

[GLT89] J. Y. Girard, Y. Lafont, and D. Taylor, *Proofs and Types*, Volume 7 of *Cambridge Tracts in Theoretical Computer Science*, Cambridge University Press, 1989.

[Han89] M. Hanus, Polymorphic higher-order programming in Prolog., *Proc. 6th International Conference on Logic Programming*, Lisbon, 1989, pp. 382–397.

[HJ90] N. C. Heintze and J. Jaffar, A finite presentation theorem for approximating logic programs, *Proc. 17th ACM Symp. on Principles of Programming Languages*, January 1990, pp. 197–209.

[HT90] P. M. Hill and R. W. Topor, *A Semantics for Typed Logic Programs*, Technical Report TR-90-11, University of Bristol, May 1990.

[Llo87] J. W. Lloyd, *Foundations of Logic Programming*, Springer-Verlag, 1987.

[Mil78] R. Milner, A theory of type polymorphism in programming, *Journal of Computer and System Sciences* **17**:3, 1978, pp. 348–375.

[Mis84] P. Mishra, Towards a theory of types in Prolog, *International Symposium on Logic Programming*, IEEE, 1984, pp. 289–298.

[MO83] A. Mycroft and R. A. O'Keefe, A polymorphic type system for Prolog, *Logic Programming Workshop*, 1983, pp. 107–121.

[Nai87] L. Naish, Specification = program + types, *Proc. 7th Conf. on Foundations of Software Technology and Theoretical Computer Science*, 1987, pp. 326–339.

[Obe62] A. Oberschlep, Untersuchungen zur mehrsortigen Quantorenlogik, *Mathematische Annalen*, 1962, pp. 297–333.

[PR89] C. Pyo and U. S. Reddy, Inference of polymorphic types for logic programs, *Proceeding of the North American Conference on Logic Programming*, Cleveland, Ohio, October 1989, pp. 1115–1132.

[Rey85] J. C. Reynolds, Three approaches to type structure, *Mathematical Foundations of Software Development (TAPSOFT Proc., Vol. 1)*, Springer LNCS, March 1985, pp. 97–138.

[Smo88] G. Smolka, Logic programming with polymorphically order-sorted types, *Proc. First International Workshop on Algebraic and Logic Programming*, Lec. Notes in Comp. Sci., Springer-Verlag, Gaussig, Germany, 1988, pp. 53–70.

[Sol78] M. Solomon, Type definitions with parameters, *5th ACM Symp. on Principles of Programming Languages*, 1978, pp. 31–38.

[Tha73] J.W. Thatcher, Tree automata: an informal survey, *Currents in the Theory of Computing* (Alfred V.Aho, ed.), chapter 4, pp. 143–172, Prentice-Hall, 1973.

[vEK76] M. H. van Emden and R. Kowalski, The semantics of predicate logic as a programming language, *JACM* **23**:4, October 1976, pp. 33–742.

[XW88] J. Xu and D. S. Warren, A type inference system for Prolog, *Proc. 5th International Conf. and symp. on Logic Programming*, Seattle, Washington, August 1988, pp. 604–619.

[YS87a] E. Yardeni and E. Shapiro, A type system for logic programs, *Concurrent Prolog* (E. Shapiro, ed.), Chapter 28, MIT Press, 1987. Also Weizmann Institute, TR CS87-05.

[Yar87b] E. Yardeni, *A type system for logic programs*, Master's thesis, Weizmann Institute of Science, 1987.

[YS89] E. Yardeni and E. Shapiro, *A type system for logic programs*, TR CS89-03, The Weizmann Institute of Science, 1989. To appear in the Journal of Logic Programming.

[Zob87] J. Zobel, Derivation of polymorphic types for Prolog programs, *Proc. 4th International Conference on Logic Programming*, Melbourne, Australia, May 1987, pp. 817–838.