Data Type Inference for Logic Programming

João Barbosa, Mário Florido, and Vítor Santos Costa
Faculdade de Ciências, Universidade do Porto, Portugal
{joao.barbosa,amflorid,vscosta}@fc.up.pt

Abstract In this paper we present a new static data type inference algorithm for logic programming. Without the need of declaring types for predicates, our algorithm is able to automatically assign types to predicates which, in most cases, correspond to the data types processed by their intended meaning. The algorithm is also able to infer types given data type definitions similar to data definitions in Haskell and, in this case, the inferred types are more informative in general. We present the type inference algorithm, prove some properties and finally, we evaluate our approach on example programs that deal with different data structures.

Keywords: Logic Programming, Types, Type Inference

1 Introduction

Types are program annotations that provide information about data usage and program execution. Ensuring that all types are correct and consistent may be a daunting task for humans. However, this task can be automatized with the use of a type inference algorithm which assigns types to programs.

Logic programming implementers have been interested in types from early on [Zob87, DZ92, Lu01, FSVY91, YFS92, MO84, LR91, SBG08, HJ92, SCWD08]. Most research approached typing an over-approximation (a superset) of the program semantics [Zob87, DZ92, YFS92, BJ88, FSVY91]: any programs that succeed will necessarily be well-typed. Other researchers followed the experience of functional languages and took a more aggressive approach to typing, where only well-typed programs are acceptable. Over the course of the last years it has become clear that there is a need for a type inference system that can support Prolog well. Next, we report on recent progress on our design, the $YAP^T$ type system. We will introduce the key ideas and then focus on the practical aspects.

Our approach is motivated by the belief that programs (Prolog or otherwise) are about manipulating data structures. In Prolog, data structures are denoted by terms with a common structure, and, being untyped one cannot naturally distinguish between failure and results of type erroneous calls. We believe that to fully use data structures we must be able to discriminate between failure, error, and success [BFC19].

1 This work is partially funded by the portuguese Fundação para a Ciência e a Tecnologia and by LIACC (FCT/UID/CEC/0027/2020).
Our starting point was a three-valued semantics that clearly distinguishes error and falsehood. In [BFC19] we use it to define a type system for logic programming that is semantically sound. In [BFC19] we presented a type system which formally defines the meaning of a well-typed program and showed that this notion of well-typing is sound with respect to a three-valued program semantics. From now on when we say well-typed we mean the notion presented in [BFC19]. Here we present the YAPF type inference algorithm which is able to automatically infer data type definitions.

To use our semantics, we shall assume that typed Prolog programs operate in a context, e.g., suppose a programming context where the well-known append predicate is expected to operate on lists:

```
append([],X,X).
append([X|R],Y,[X|R1]) :- append(R,Y,R1).
```

This information is not achievable when using type inference as a conservative approximation of the success set of the predicate. The following example shows the output of type inference in this case, where \( t_i \) is the type of the \( i \)-th argument of append, “+” means type disjunction and “A” and “B” are type variables:

\[
\begin{align*}
  t_1 &= \{\} + \{A \mid t_1\} \\
  t_2 &= B \\
  t_3 &= B + \{A \mid t_3\}
\end{align*}
\]

Figure 1. (1) Program approximation ; (2) Well-typing

Types \( t_2 \) and \( t_3 \), for the second and third argument of the left-hand side (1), do not filter any possible term, since they have a type variable as a member of the type definition, which can be instantiated with any type. And, in fact, assuming the specific context of using append as list concatenation, some calls to append succeed even if unintended\(^2\), such as \( \text{append}([\],1,1) \). The solution we found for these arguably over-general types is the definition of closed types, that we first presented in [BFSC17], which are types where every occurrence of a type variable is constrained. We also defined a closure operation, from open types into closed types, using only information provided by the set of types themselves. Applying our type inference algorithm with closure to the append predicate yields the types on the right-hand side (2), which are the intended types \( \text{[Na92]} \) for the append predicate.

Our type inference algorithm\(^3\) works for pure Prolog with system predicates for arithmetic. We assume as base types int, float, string, and atom. There is an optional definition of type declarations (like data declarations in Haskell), which, if declared by the programmer, are used by the type inference algorithm to refine

\(^2\) accordingly to a notion of intended meaning first presented in [Na92]

\(^3\) Implementation at https://github.com/JoaoLBarbosa/TypeInferenceAlgorithm
types. We follow a syntax inspired in [SCWD08] to specify type information. One example of such a declaration is the list datatype

\[-\text{type list}(A) = [\] + [A | list(A)].\]

In order to simplify further processing our type inference algorithm first compiles every predicate to a simplified form called kernel Prolog [VR90]. In this representation, each predicate is defined by a single clause \((H : - B)\), where the head \(H\) contains distinct variables as arguments and the body \(B\) is a disjunction (represented by the symbol :) of queries. We assume that there are no common variables between queries, except for the variables that occur in the head of the clause, without loss of generality. In this form the scope of variables is not limited to a single clause, but is extended over the whole predicate and thus type inference is easier to perform. In the rest of the paper we will assume that predicate definitions are always in kernel Prolog.

As simple illustrating examples, using a pre-defined type declaration for lists, and reading ":=" as "has type", our type inference algorithm gives the following results:

| PREDICATE | TYPE |
|-----------|------|
| l(X) :- (X = []; X = [Y|Ys], l(Ys)). | l :: t1 |
| p(X) :- X = a ; X = 3. | p :: t2 |
| q(Y) :- Y = 1.23; Y = 5. | q :: t3 |
| h(Z) :- Z = W, Z = Q, p(W), q(Q). | h :: t4 |

2 Types

Here we define a new class of expressions, which we shall call types. We first define the notion of type term built from an infinite and enumerable set of type variables \(TVar\), a finite set of base types \(TBase\), an infinite and enumerable set of constants \(TCons\), an infinite and enumerable set of function symbols \(TFunc\), and an infinite and enumerable set of type symbols, \(TSym\). Type terms can be:

- a type variable \((\alpha, \beta, \gamma, \cdots \in TVar)\)
- a type constant \((1, [], 'c', \cdots \in TCons)\)
- a base type \((\text{int, float,} \cdots \in TBase)\)
- a function symbol \(f \in TFunc\) associated with an arity \(n\) applied to an \(n\)-tuple of type terms \((f(\text{int,} [], g(X)))\).
– a type symbol $\sigma \in T\text{Sym}$ associated with an arity $n$ ($n \geq 0$) applied to an
$n$-tuple of type variables ($\sigma(X,Y)$).

Type variables, constants and base types are called basic types. A ground
type term is a variable-free type term. A type definition, which introduces a new
type called an algebraic data type, is of the form:

$$\sigma(X_1, \ldots, X_k) = \tau_1 + \ldots + \tau_n,$$

where each $\tau_i$ is a type term and $\sigma$ is the type symbol being defined. Variables
$X_1, \ldots, X_n$, for $n \geq 0$, include the type variables occurring in $\tau_1 + \ldots + \tau_n$, and
are called type parameters. The sum $\tau_1 + \ldots + \tau_n$ is a union type, describing
values that may have one of the types $\tau_1, \ldots, \tau_n$. Note that type definitions may
be recursive. Deterministic type definitions are type definitions where, on the
right-hand side, none of $\tau_i$ is a type symbol and if $\tau_i$ is a type term starting with
a type function symbol $f$, then no other $\tau_j$ starts with $f$.

Example 1. Assuming a base type int for the set of all integers, the type list of
integers is defined by the type definition $\text{list} = [\text{int}] + ([\text{list} | \text{int}])$.

A predicate type is a functional type from a tuple of the type terms defining
the types of its arguments to bool: $\tau_1 \times \ldots \times \tau_n \rightarrow \text{bool}$. A type can be a type
term, an algebraic data type or a predicate type. In [BFC19] we defined a formal
Hindley-Milner semantics for types.

Our type language enables parametric polymorphism through the use of type
schemes. A type scheme is defined as $\forall X_1 \ldots \forall X_n T$, where $T$ is a predicate type
and $X_1, \ldots, X_n$ are distinct type variables. In logic programming, there have
been several authors that have dealt with polymorphism with type schemes
or in a similar way [PR89, BG92, Hen93, Zob87, FSY91, GdW94, YFS92, FD92]. Type schemes have type variables as generic place-holders for ground
type terms. Parametric polymorphism comes form the fact these type variables
can be instantiated with any type.

Example 2. A polymorphic list is defined by the following type definition:

$$\text{list}(X) = [\text{int}] + ([X \mid \text{list}(X)]$$

Notation Throughout the rest of the paper, for the sake of readability, we
will omit the universal quantifiers on type schemes and the type parameters as
explicit arguments of type symbols in inferred types. Thus we will assume that
all free type variables on type definitions of inferred types are type parameters
which are universally quantified.

Most type languages in logic programming use tuple distributive closures of
types. The notion of tuple distributivity was given by Mishra [Mis84]. Throughout
this paper, we restrict our type language to tuple distributive types, where
type definitions are deterministic.

4 Type definitions will use the user friendly Prolog notation for lists instead of the list constructor
Sometimes, the programmer wants to introduce a new type in a program, so that it is recognized when performing type inference. It is also a way of having a more structured and clear program. These declarations act similarly to data declarations in Haskell.

In our algorithm, types can be declared by the programmer in the following way: $\text{type symbol(\text{type vars})} = \text{type term}_1 + \ldots + \text{type term}_n$. One example would be:

$$\text{:- type tree(X) = empty + node(X, tree(X), tree(X)).}$$

In the rest of the paper we will assume that all constants and function symbols that start a summand in a declared type cannot start a summand in a different one, thus there are no overloaded constants nor function symbols. Note that there is a similar restriction on data declarations in functional programming languages.

### 2.1 Closed Types

Closed types were first defined in [BFSC17]. Informally, they are types where every occurrence of a type variable is constrained. If a type is not closed, we say that it is an open type. The restrictions under the definition of closed type can be compressed in the following three principles:

- Types should denote a set of terms which is strictly smaller than the set of all terms
- Every use of a variable in a program should be type constrained
- Types are based on self-contained definitions

The last one is important to create a way to go from open types to closed types. We defined what is an unconstrained type variable as follows:

**Definition 1 (Unconstrained Type Variable).** A type variable $\alpha$ is unconstrained with respect to a set of type definitions $T$, notation $\text{unconstrained}(\alpha; T)$, if and only if it occurs exactly once in the set of all the right-hand sides of type definitions in $T$.

Unconstrained type variables type terms with any type, thus they do not really provide type information. We now define closed algebraic data types, which are types without variables as summands in their definition.

**Definition 2 (Closed Algebraic Data Type).** An algebraic data type $\tau$ is closed, notation $\text{closedDataType}(\tau)$, if and only if it has no type variables as summands in its type definition.

The definition for closed types uses these two previous auxiliary definitions. Closed types correspond to close records or data definitions in functional programming languages. The definition follows:
**Definition 3 (Closed Types).** A type term $\tau$ is closed with respect to a set of type definitions $T$, notation $\text{closed}(\tau, T)$, if and only if the predicate defined as follows holds:

$$
\text{closed}(\tau, T) = \begin{cases} 
\text{closedDataType}(\tau) & \text{if } \tau \text{ is an algebraic data type} \\
\neg\text{unconstrained}(\tau, T) & \text{if } \tau \text{ is a type variable} \\
\text{True} & \text{if } \tau \text{ is basic but not a type variable}
\end{cases}
$$

**Example 3.** We recall the example in the Introduction, of the following types for the $\text{append}$ predicate, where $t_n$ is the type of $n$th predicate argument:

- $t_1 = [] + [A \mid t_1]$
- $t_2 = B$
- $t_3 = B + [A \mid t_3]$

Type $t_3$, for the third argument of $\text{append}$, is open, because $t_3$ has a type variable as a summand, thus it does not filter any possible term, since the type variable can be instantiated with any type. An example of a valid closed type for $\text{append}$ is:

- $t_1 = [] + [A \mid t_1]$
- $t_2 = [] + [A \mid t_2]$
- $t_3 = [] + [A \mid t_3]$

The next step is to transform open types into closed types. Note that some inferred types may be already closed. For the ones that are not, we defined a closure operation, described in detail in [BFSC17]. This closure operation is present in our type inference algorithm as an optional step to perform on the resulting types after solving the type constraints generated by the algorithm.

### 3 Examples

There are some flags in the type inference algorithm that can be turned on or off:

- **basetype** (default: on) - this flag types constants with their base types when turned on and with a constant symbol equal to the constant itself otherwise;
- **list** (default: off) - this flag adds the data type declaration for polymorphic lists to the program when turned on;
- **closure** (default: off) - this flag applies closure to the resulting type definitions after solving the constraints when turned on.

In the following examples $p_i$ is the type symbol for the type of the $i$th argument of predicate $p$ and we assume that all free type variables on type definitions are universally quantified and that the type of arguments of built-in arithmetic predicates is predefined as $\text{int} + \text{float}$.

**Example 4.** Let us consider the predicate $\text{concat}$, which flattens a list of lists, where $\text{app}$ is the $\text{append}$ predicate:
concat(X1, X2) :- X1=[], X2=[];
    X1=[X|Xs], X2=List, concat(Xs,NXs), app(X,NXs, List).

app(A, B, C) :- A=[], B=D, C=D;
    app(E, F, G), E=H, F=I, G=J, A=[K|H], B=I, C=[K|J].

The types inferred with all the flags off correspond to types inferred in previous
type inference algorithms which view types as an approximation of the success
set of the program:

concat :: concat1 x concat2
concat1 = [] + [ t | concat1 ]
concat2 = C + [] + [ B | concat2 ]
t = [] + [ B | t ]

app :: app1 x app2 x app3
app1 = [] + [ A | app1 ]
app2 = B
app3 = B + [ A | app3 ]

Now the types inferred when turning on the closure flag, closure_flag, are:

concat :: concat1 x concat2
concat1 = [] + [ concat2 | concat1 ]
concat2 = [] + [ B | concat2 ]

app :: app1 x app2 x app3
app1 = [] + [ A | app1 ]
app2 = [] + [ A | app2 ]
app3 = [] + [ A | app3 ]

Note that these types are not inferred by any previous type inference algorithm
for logic programming so far, and they are a step forward to the automatic
inference of program use in a specific context, more precisely, a context which
corresponds to how it would be used in a programming language with data type
declarations, such as Curry [Han13] or Haskell.

*Example 5.* Let `rev` be the reverse list predicate, defined using the `append` defini-
tion used in the previous example:

rev(A, B) :- A=[], B=[] ;
    rev(C, D), append(D, E, F), E=[G], A=[G|C], B=F.

The inferred types with all flags off is (the types inferred for append are the
same as the one in the previous example):

rev :: rev1 x rev2
rev1 = [] + [ A | rev1 ]
rev2 = [] + [ t | rev2 ]
t = B + A
If we turn on the `list_flag`, which declares the data type for Prolog lists, the type inference algorithm outputs the same types that would be inferred in Curry or Haskell with pre-defined built-in lists:

\[
\text{rev} :: \text{rev1} \times \text{rev2} \\
\text{rev1} = \text{list(A)} \\
\text{rev2} = \text{list(A)} \\
\text{list(X)} = [\] + [ X | \text{list}(X) ]
\]

We now show an example of the minimum of a tree.

**Example 6.** Let `tree_minimum` be the predicate defined as follows:

\[
\text{tree_min}(A, B) :- A=\text{empty}, B=0 ;  \\
A=\text{node}(C, D, E), \text{tree_min}(D, F), \text{tree_min}(E, G),  \\
Y=[C,F,G], \text{minimum}(Y, X), X=B.  \\
\text{minimum}(A, B) :- A=[I], B=I;  \\
A=[X|Xs], \text{minimum}(Xs, C), X=C;  \\
A=[Y|Ys], \text{minimum}(Ys, D), D=Y, B=D.  \\
\text{tree_min} :: \text{tree_min1} \times \text{tree_min2}  \\
\text{tree_min1} = \text{atom} + \text{node}(\text{tree_min2}, \text{tree_min1}, \text{tree_min1})  \\
\text{tree_min2} = \text{int} + \text{float}  \\
\text{minimum} :: \text{minimum1} \times \text{minimum2}  \\
\text{minimum1} = [ \text{minimum2} | t ]  \\
\text{minimum2} = \text{A} + \text{int} + \text{float}  \\
t2 = [\] + [ \text{minimum2} | t2 ]
\]

The inferred types with all flags off, except for the `basetype_flag`, are:

\[
\text{tree_minimum} :: \text{tree_minimum1} \times \text{tree_minimum2}  \\
\text{tree_minimum1} = \text{tree}(\text{tree_minimum2})  \\
\text{tree_minimum2} = \text{int} + \text{float}  \\
\text{minimum} :: \text{minimum1} \times \text{minimum2}  \\
\text{minimum1} = \text{list}(\text{minimum2})  \\
\text{minimum2} = \text{int} + \text{float}  \\
\text{tree}(X) = \text{empty} + \text{node}(X, \text{tree}(X), \text{tree}(X))  \\
\text{list}(Y) = [\] + [ Y | \text{list}(Y) ]
\]
4 Type Inference

The type inference algorithm is composed of several modules, as described in Figure 2. On a first step, when consulting programs, we apply term expansion to transform programs into the internal format that the rest of the algorithm expects. Secondly, we have the type inference phase itself, where a type constraint solver outputs the inferred types for a given program. There is also a simplification phase performed during inference. After this, we run directly a type pretty printer, or go through closure before printing the types.

Thus the type inference algorithm is composed of four main parts with some auxiliary steps:

- Term expansion
- Constraint generation
- Constraint solving
- Closure (optional)

There is also a simplification step, that is applied at every step. Without closure or type declarations our algorithm follows a standard approach of types as approximations of the program semantics. Using our algorithm to infer well-typings (which filter program behaviour instead of approximating it) is possible either by using explicit type declarations or by using the closure step.

4.1 Stratification

We assume that the input program of our algorithm is stratified. To understand the meaning of stratified programs, let us define the dependency directed graph of a program as the graph that has one node representing each predicate in the program and an edge from q to p for each call from a predicate p to a predicate q.
Definition 4 (Stratified Program). A stratified program $P$ is such that the dependency directed graph of $P$ has no cycles of size more than one.

This means that our type inference algorithm deals with predicates defined by direct recursion but not with mutual recursion. Note that stratified programs are widely used and characterize a large class of programs which is used in several database and knowledge base systems [Ull88]. Despite this, expanding our algorithm to mutually recursive predicates is a step we are working towards.

4.2 Constraints and Constraint Generation

The type inference algorithm begins by generating type constraints from a logic program, that will be solved by a constraint solver in a second stage of the algorithm. There are two different kinds of type constraints: equality constraints and subtyping constraints. The constraint generation step of the algorithm will output two sets of constraints, $Eq$ (a set of equality constraints) and $Ineq$ (a set of subtyping constraints), that need to be solved during type inference. An equality constraint is of the form $\tau_1 = \tau_2$ and a subtyping constraint is of the form $\tau_1 \leq \tau_2$. In the subtyping constraint case, $\tau_i$ can be a type term or a sum of several type terms.

An assumption is a type declaration for a variable, written $X : \tau$, where $X$ is a variable and $\tau$ a type. We define a context $\Gamma$ as a set of assumptions with distinct variables as subjects (alternatively contexts can be defined as functions from variables to types and where domain($\Gamma$) stands for its domain). Contexts are synthetized by the type inference algorithm in two different ways: we can build a new context by linking assumptions in different contexts with disjunction (corresponding to different clauses of the same predicate) or with conjunction (corresponding to different queries). For this we define two auxiliary functions, $\otimes$ (for conjunction) and $\oplus$ (for disjunction), which are used by the constraint generation algorithm. They are defined as follows:

Definition 5. Let $\Gamma_1$ and $\Gamma_2$ be two contexts and $Def$ be a set of type definitions defining the type symbols in $\Gamma_1 \cup \Gamma_2$. Let $V = \text{domain}(\Gamma_1) \cap \text{domain}(\Gamma_2)$.

$\otimes((\Gamma_1 \cup \Gamma_2), Def) = (\Gamma, Def^t)$, such that $Def \subseteq Def^t$,

where $\forall X \in V. \Gamma(X) = \sigma$ and $(\sigma = \Gamma_1(X) + \Gamma_2(X)) \in Def^t$, all $\sigma$s fresh, and

$\forall X. X \in \text{domain}(\Gamma_i), X \notin V. \Gamma(X) = \Gamma_i(X)$, for $i, j = 1, 2$.

Definition 6. Let $\Gamma_1$ and $\Gamma_2$ be two contexts and $Def$ be a set of type definitions defining the type symbols in $\Gamma_1 \cup \Gamma_2$. Let $V = \text{domain}(\Gamma_1) \cap \text{domain}(\Gamma_2)$.

$\oplus((\Gamma_1 \cup \Gamma_2), Def) = (\Gamma, Def^t, Eq)$

where $\forall X \in V. \Gamma(X) = \sigma$, $(\sigma = \alpha) \in Def^t$, and $\{\alpha = \text{def}(\Gamma_i(X)), \alpha = \text{def}(\Gamma_2(X))\} \in Eq$, all $\alpha$s and $\sigma$s fresh, and

$\forall X. X \in \text{domain}(\Gamma_i), X \notin V. \Gamma(X) = \Gamma_i(X)$, for $i, j = 1, 2$.

Let $P$ be a term, an atom, a query, a sequence of queries or a clause. $\text{generate}(P)$ is a function that outputs a tuple of the form $(\tau, \Gamma, Eq, Ineq, Def)$,
where $\tau$ is a type, $\Gamma$ is an context for logic variables, $Eq$ is a set of equality constraints, $Ineq$ is a set of subset constraints, and $Def$ is a set of type definitions.

The function $generate$, which generates the initial type constraints, is defined by cases from the program syntax. Its definition follows:

$$generate(P) =$$

- $generate(X) = (\alpha, \{ X : \sigma \}, \emptyset, \emptyset, \{ \sigma = \alpha \})$, $X$ is a variable, where $\alpha$ is a fresh type variable and $\sigma$ is a fresh type symbol.
- $generate(c) = (\text{basetype}(c), \emptyset, \emptyset, \emptyset, c)$, $c$ is a constant.
- $generate(f(t_1, \ldots, t_n)) = (\text{basetype}(f)(\tau_1, \ldots, \tau_n), \Gamma, Eq, \emptyset, Def)$, where $generate(t_i) = (\tau_i, \Gamma_i, Eq_i, \emptyset, Def_i)$, $(\Gamma, Eq, Def) = \otimes((\Gamma_1 \cup \ldots \cup \Gamma_n), (Def_1 \cup \ldots \cup Def_n))$, and $Eq = Eq_1 \cup \ldots \cup Eq_n \cup Eq$.
- $generate(t_1 = t_2) = (\text{bool}, \Gamma, Eq, \emptyset, Def)$ where $generate(t_i) = (\tau_i, \Gamma_i, Eq_i, \emptyset, Def_i)$ $(\Gamma, Def, Eq) = \otimes((\Gamma_1 \cup \Gamma_2), (Def_1 \cup Def_2))$, and $Eq = Eq_1 \cup Eq_2 \cup \{ \tau_1 = \tau_2 \} \cup Eq$.
- $generate(p(t_1, \ldots, t_n)) = (\text{bool}, \{ X_1 : \sigma_1, \ldots, X_n : \sigma_n \}, Eq, \{ \alpha_1 \leq \tau_1, \ldots, \alpha_n \leq \tau_n \}, Def)$, a non-recursive call, where $generate(p(Y_1, \ldots, Y_n) : \text{~body}) = (\text{bool}, \Gamma, Eq, Ineq, Def)$, $\{ Y_1 : \tau_1, \ldots, Y_n : \tau_n \} \in \Gamma$ $Def = Def \cup \{ \sigma_i = \alpha_i \}$, and $\sigma_i$ and $\alpha_i$ are all fresh.
- $generate(p(t_1, \ldots, t_n)) = (\text{bool}, \{ X_1 : \alpha_1, \ldots, X_n : \alpha_n \}, \emptyset, \{ \alpha_1 \leq \tau_1, \ldots, \alpha_n \leq \tau_n \}, \emptyset)$, a recursive call, where $\tau_1, \ldots, \tau_n$ are the types for the variables in the head of the clause this call occurs in, and $\alpha_i$ are all fresh type variables.
- $generate(c_1, \ldots, c_n) = (\text{bool}, \Gamma, Eq, Ineq_1 \cup \ldots \cup Ineq_n, Def)$, where $generate(c_i) = (\text{bool}, \Gamma_i, Eq_i, Ineq_i, Def_i)$, $(\Gamma, Def, Eq) = \otimes((\Gamma_1 \cup \ldots \cup \Gamma_n), (Def_1 \cup \ldots \cup Def_n))$, and $Eq = Eq_1 \cup \ldots \cup Eq_n \cup Eq$.
- $generate(b_1; \ldots; b_n) = (\text{bool}, \Gamma, Eq_1 \cup \ldots \cup Eq_n, Ineq_1 \cup \ldots \cup Ineq_n, Def)$, where $generate(c_i) = (\text{bool}, \Gamma_i, Eq_i, Ineq_i, Def_i)$, and $(\Gamma, Def) = \oplus((\Gamma_1 \cup \ldots \cup \Gamma_n), (Def_1 \cup \ldots \cup Def_n))$.
- $generate(p(X_1, \ldots, x_n) : \text{~body}) = (\text{bool}, \Gamma, Eq, Ineq, Def)$, where $generate(\text{body}) = (\text{bool}, \Gamma, Eq, Ineq, Def)$.  

**Example 7.** Consider the following predicate:
list(X) :- X = []; X = [Y|YS], list(YS).

the output of applying the generate function to the predicate is:

generate(list(X) :- X = []; X = [Y|YS], list(YS)) =
{bool, {X : σ₁, Y : σ₂, Ys : σ₃}, {α = [], β = [δ | ε], ε = ν}, {ν ≤ σ₁}, {σ₁ = α + β, σ₂ = δ, σ₃ = ε, σ₄ = ν}}
The set {ν ≤ σ₁} comes from the recursive call to the predicate, while α = [] comes from X = [], and β = [δ | ε] comes from X = [Y|YS]. The other constraints come from the application of the ⊗ operation.

4.3 Constraint Solving

Let Eq be a set of equality constraints, Ineq be a set of subtyping constraints, and Def a set of type definitions. Function solve(Eq, Ineq, Def) solves the constraints, outputing a new set of type definitions. Note that the rewriting rules in the following definitions of the solver algorithm are assumed to be ordered.

\[
solve(Eq, Ineq, Def) = FinalDef,
\]

\[
solve(Eq) = S
\]

\[
solve_{ineq}(S(Ineq), S(Def)) = FinalDef
\]

**Definition 7.** A set of equality constraints is in solved form if all constraints are of the form α = τ, where α is a type variable and τ is a type term.

**Solving Equality Constraints** Let Eq be a set of equality constraints. Then solve_eq is a rewriting algorithm that outputs a set of equality constraints in solved form, which can be interpreted as a substitution, that will later be applied to Ineq and Def.

We now present the rewrite rules for solving the type equality constraints. A configuration is either the term fail (representing failure) or a set of equality constraints Eq. The rewriting algorithm consists of the transformation rules on configurations listed below.

\[
solve_{eq}(Eq) =
\]

1. \{t = t\} ∪ Rest → Rest
2. \{α = t\} ∪ Rest → \{α = t\} ∪ Rest[α → t], if α, which is a type variable, occurs somewhere in Rest
3. \{t = α\} ∪ Rest → \{α = t\} ∪ Rest, where α is a type var and t is not a type variable nor a type symbol
4. \{f(t₁, . . . , tₙ) = f(s₁, . . . , sₙ)\} ∪ Rest → \{t₁ = s₁, . . . , tₙ = sₙ\} ∪ Rest
5. otherwise → fail

Note that an occur check would be required in step 2, but from the definition of our constraint generation algorithm, it never happens that α occurs in t.
Example 8. Following example\textsuperscript{7} applying solve\textsubscript{eq} to the set of equality constraints is as follows step-by-step:

\[ \{ \alpha = [], \beta = [\delta | e], \epsilon = v \} \rightarrow_2 \{ \alpha = [], \beta = [\delta | v] \} \]

And the final set only contains constraints in solved form, that can be seen as a substitution to be applied to Ineq and Def.

Solving Subtyping Constraints Let Ineq be a set of inequality constraints and Def be a set of type definitions. solve\textsubscript{ineq} is a rewriting algorithm that outputs a new set of type definitions.

A configuration now is either the term fail (representing failure) or a pair of the form (Ineq, Def), where Ineq and Def are as described above. The rewriting algorithm consists on the transformation rules on configurations listed below. In this algorithm we assume a store of pairs of types that already have been compared by the algorithm.

\[ \text{solve}_{\text{ineq}}(\text{Ineq}, \text{Def}) = \]

1. \((\{t \leq t\} \cup \text{Rest}, \text{Def}) \rightarrow (\text{Rest}, \text{Def})\)
2. \((\{f(t_1, \ldots, t_n) \leq f(s_1, \ldots, s_n)\} \cup \text{Rest}, \text{Def}) \rightarrow (\{t_1 \leq s_1, \ldots, t_n \leq s_n\} \cup \text{Rest}, \text{Def})\)
3. \((\{\alpha \leq t_1, \ldots, \alpha \leq t_n\} \cup \text{Rest}, \text{Def}) \rightarrow (\{\alpha \leq t\} \cup \text{Rest}, S(\text{Def}))\), where \(\alpha\) is a type variable, \(n \geq 2\), and intersect\((t_1, \ldots, t_n) = (t, S)\)
4. \((\{\alpha \leq t\} \cup \text{Rest}, \text{Def}) \rightarrow (\text{Rest}, \text{Def}[\alpha = t])\), where \(\alpha\) is a type variable
5. \((\{t_1 + \ldots + t_n \leq t\} \cup \text{Rest}, \text{Def}) \rightarrow (\{t_1 \leq t, \ldots, t_n \leq t\} \cup \text{Rest}, \text{Def})\)
6. \((\{\sigma \leq t\} \cup \text{Rest}, \text{Def}) \rightarrow (\text{Rest}, \text{Def}), \) if \(\sigma\) and \(t\) have already been compared
7. \((\{\sigma \leq t\} \cup \text{Rest}, \text{Def}) \rightarrow (\{\text{Rhs}_\sigma \leq t\} \cup \text{Rest}, \text{Def})\), where \(\sigma\) is a type symbol, and \(\sigma = \text{Rhs}_\sigma \in \text{Def}\). Also add \((\sigma, t)\) to the store of pairs of types that have been compared.
8. \((\{t_1 \leq \alpha, \ldots, t_n \leq \alpha\} \cup \text{Rest}, \text{Def}) \rightarrow (\text{Rest}, \text{Def}[\alpha = t_1 + \ldots + t_n])\)
9. \((\{t \leq t_1 + \ldots + t_n\} \cup \text{Rest}, \text{Def}) \rightarrow (\{t \leq t_1\} \cup \text{Rest}, \text{Def})\), where \(t_i\) is one of the summands
10. \((\{t \leq \sigma\} \cup \text{Rest}, \text{Def}) \rightarrow (\text{Rest}, \text{Def}), \) if \(\sigma\) and \(t\) have already been compared.
11. \((\{t \leq \sigma\} \cup \text{Rest}, \text{Def}) \rightarrow (\{t \leq \text{Rhs}_\sigma\} \cup \text{Rest}, \text{Def}), \) where \(\sigma\) is a type symbol, and \(\sigma = \text{Rhs}_\sigma \in \text{Def}\). Also add \((\sigma, t)\) to the store of pairs of types that have been compared.
12. \((\emptyset, \text{Def}) \rightarrow \text{Def}\)
13. otherwise \(\rightarrow \) fail

Type intersection, intersect\((t_1, \ldots, t_n)\), is calculated using a version of Zobel's intersection algorithm previously presented in \textsuperscript{Zob87}. 

\[ \]

13
Example 9. Following example \(\ref{example_7}\), applying `solve.ineq` to the set of subtyping constraints is as follows step-by-step:

\[
\left\{ v \leq \sigma_1 \right\}, \left\{ \sigma_1 = [\ ] + [\delta \mid v] \right\} \rightarrow_4 (\emptyset, \left\{ \sigma_1 = [\ ] + [\delta \mid \sigma_1] \right\})
\]

**Arithmetic** To extend the type inference algorithm to include built-in arithmetic predicates (\(\leq/2\), \(</2\), \(\geq/2\), \(>/2\), and \(is/2\)) we add the following rule to the constraint generation algorithm:

- \(\text{generate}(p(t_1, t_2)) = (\text{bool}, \left\{ X_1 : \sigma_1, \ldots, X_n : \sigma_n \right\}, \emptyset, \left\{ \alpha_1 \leq \text{num}, \ldots, \alpha_n \leq \text{num} \right\}, \left\{ \sigma_1 = \alpha_1, \ldots, \sigma_n = \alpha_n \right\})\), where \(X_1, \ldots, X_n\) are the variables occurring in \(t_1\) and \(t_2\); \(\sigma_i\) and \(\alpha_i\) are fresh; and \(\text{num} = \text{int} + \text{float} \).

### 4.4 Decidability and Soundness

The next theorem shows that both the equality constraint and subtyping constraint solvers terminate at every input set of constraints.

**Theorem 1 (Termination).** `solve.eq` and `solve.ineq` always terminate. When `solve.eq` terminates, it either fails or the output is in solved form. When `solve.ineq` terminates, it either fails or the output is a pair with zero subtyping constraints.

Proofs of this theorem follows a usual termination proof approach, where we show that a carefully chosen metric decreases at every step.

We do not have a formal proof of soundness yet, but we believe that proofs of soundness of type inference algorithms for functional programming can be adapted to the logic programming case; in particular, we need the following extra definitions and lemmas:

- a type system which defines the notion of well-typed program: for this we use the type system defined in [BFC19];
- a notion of constraint satisfaction;
- a lemma for the soundness of constraint solving which states that the solver terminates in a set of constraints in solved form which satisfies the initial set of constraints.

Informally, the soundness theorem should state that if one applies the substitution corresponding to the solved form of the generated constraints to the type obtained by the constraint generation function, we get a well-typed program.

### 5 Related Work

Types have been used before in Prolog systems: relevant works on type systems and type inference in logic programming include types used in the logic programming systems CIAO Prolog [SG95] [VB02], SWI and Yap [SCWD08]. CIAO uses types as approximations of the success set, while we use types as
filters to the program semantics. There is an option where the programmer gives
the types for the programs in the form of assertions, which is recommended in
PCPH08. The well-typings given in SBCG08, also have the property that they
never fail, in the sense that every program has a typing, which is not the case in
our algorithm, which will fail for some predicates. The previous system of Yap
only type checked predicate clauses with respect to programmer-supplied type
signatures. Here we define a new type inference algorithm for pure Prolog, which
is able to infer data types.

In several other previous works types approximated the success set of a pre-
dicate ZOB87, DZ92, YFS92, BJ88. This sometimes led to overly broad types,
because the way logic programs are written can be very general and accept more
than what was initially intended. These approaches were different from ours in
the sense that in our work types can filter the success set of a predicate, whenever
the programmer chooses to do so, using the closure operation.

A different approach relied on ideas coming from functional program-
languages MO84, LR91, HL94, SCWD08. Other examples of the influence of
functional languages on types for logic programming are the type systems used
in several functional logic programming languages Han13, SHC96. Along this
line of research, a rather influential type system for logic programs was Mycroft
and O’Keefe type system MO84, which was later reconstructed by Lakshman
and Reddy LR91. This system had types declared for the constants, func-
sion symbols and predicate symbols used in a program. Key differences from our
work are: 1) in the Mycroft-O’Keefe type system, each clause of a predicate must have
the same type. We lift this limitation extending the type language with sums
of types, where the type of a predicate is the sum of the types of its clauses;
2) although we may use type declarations, they are optional and we can use a
closure operation to infer datatype declarations from untyped programs.

Set constraints have also been used by many authors to infer types for logic
programming languages HJ92, GdW94, TTD97, CP98, DMP00, DMP02. Al-
though these approaches differ from ours since they follow the line of conservative
approximations to the success set, we were inspired from several techniques from
this area to define our type constraint solvers.

6 Conclusions and Future Work

In this paper, we present a flexible type inference algorithm for Prolog. Inferred
types are standard semantic approximations by default, but the user may tune
the algorithm, quite easily, to automatically infer types which correspond to the
usual algebraic data types used in the program. Moreover, the algorithm may
also be tuned to use predefined (optional) data type declarations to improve the
output types. Although our algorithm was tested and it was able to locate type
errors in several existing Prolog programs, the next obvious step of this work is
to prove its soundness with respect to a type system which defines the notion of
well-typed program.
References

BFC19. João Barbosa, Mápio Florido, and Vítor Santos Costa. A three-valued semantics for typed logic programming. In Proceedings 35th International Conference on Logic Programming (Technical Communications), ICLP 2019 Technical Communications, Las Cruces, NM, USA, September 20-25, 2019, volume 306 of EPTCS, pages 36–51, 2019.

BFSC17. João Barbosa, Mápio Florido, and Vítor Santos Costa. Closed types for logic programming. In 25th Int. Workshop on Functional and Logic Programming (WFLP 2017), 2017.

BG92. Roberto Barbuti and Roberto Giacobazzi. A bottom-up polymorphic type inference in logic programming. Sci. Comput. Program., 19(3):281–313, 1992.

BJ88. Maurice Bruynooghe and Gerda Janssens. An instance of abstract interpretation integrating type and mode inferencing. In Fifth International Conference and Symposium, Washington, 1988, pages 669–683, 1988.

CP98. Witold Charatonik and Andreas Podelski. Directional type inference for logic programs. In Giorgio Levi, editor, Static Analysis, pages 278–294. Springer Berlin Heidelberg, 1998.

DMP00. Wlodzimierz Drabant, Jan Maluszynski, and Paweł Pietrzak. Locating Type Errors in Untyped CLP Programs, pages 121–150. Springer Berlin Heidelberg, 2000.

DMP02. Wlodzimierz Drabant, Jan Maluszynski, and Paweł Pietrzak. Using parametric set constraints for locating errors in CLP programs. Theory and Practice of Logic Programming, 2(4-5):549–610, 2002.

DZ92. Philip W. Dart and Justin Zobel. A regular type language for logic programs. In Frank Pfenning, editor, Types in Logic Programming, pages 157–187. The MIT Press, 1992.

FD92. Mápio Florido and Luís Damas. Types as theories. In Proc. of post-conference workshop on Proofs and Types, Joint International Conference and Symposium on Logic Programming, 1992.

FSVY91. Thom W. Frühwirth, Ehud Y. Shapiro, Moshe Y. Vardi, and Eyal Yardeni. Logic programs as types for logic programs. In Proc. of the Sixth Annual Symposium on Logic in Computer Science (LICS ’91), Netherlands, 1991, pages 300–309, 1991.

GdW94. John P. Gallagher and D. Andre de Waal. Fast and precise regular approximations of logic programs. In Logic Programming, International Conference on Logic Programming, Italy, 1994, pages 599–613, 1994.

Han13. Michael Hanus. Functional logic programming: From theory to Curry. In Programming Logics - Essays in Memory of Harald Ganzinger, pages 123–168, 2013.

Hen93. Fritz Henglein. Type inference with polymorphic recursion. ACM Trans. Program. Lang. Syst., 15(2):253–289, 1993.

HJ92. Nevin Heintze and Joxan Jaffar. Semantic types for logic programs. In Frank Pfenning, editor, Types in Logic Programming, pages 141–155. The MIT Press, 1992.

HL94. Patricia M. Hill and John W. Lloyd. The Gödel programming language. MIT Press, 1994.
T. L. Lakshman and Uday S. Reddy. Typed Prolog: A semantic reconstruction of the Mycroft-O'Keefe type system. In Logic Programming, Proceedings of the 1991 International Symposium, San Diego, California, USA, 1991.

Lunjin Lu. On Dart-Zobel algorithm for testing regular type inclusion. SIGPLAN Not., 36(9):81–85, 2001.

Prateek Mishra. Towards a theory of types in Prolog. In Proceedings of the 1984 International Symposium on Logic Programming, Atlantic City, New Jersey, USA, February 6-9, 1984, pages 289–298. IEEE-CS, 1984.

Alan Mycroft and Richard A. O'Keefe. A polymorphic type system for Prolog. Artif. Intell., 23(3):295–307, 1984.

Lee Naish. Types and the intended meaning of logic programs. In Frank Pfenning, editor, Types in Logic Programming, pages 189–216. The MIT Press, 1992.

Pawel Pietrzak, Jesus Correas, Germán Puebla, and Manuel V. Hermenegildo. A practical type analysis for verification of modular Prolog programs. In Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, pages 61–70, 01 2008.

Changwoo Pyo and Uday S. Reddy. Inference of polymorphic types for logic programs. In Logic Programming, Proceedings of the North American Conference 1989, USA, 1989. 2 Volumes, pages 1115–1132, 1989.

Tom Schrijvers, Maurice Bruynooghe, and John P. Gallagher. From monomorphic to polymorphic well-typings and beyond. In Logic-Based Program Synthesis and Transformation, 18th International Symposium, LOPSTR 2008, Spain, July 17-18, 2008, pages 152–167, 2008.

Tom Schrijvers, Vitor Santos Costa, Jan Wielemaker, and Bart Demoen. Towards typed Prolog. In Logic Programming, 24th International Conference, ICLP 2008, Udine, Italy, Proceedings, pages 693–697, 2008.

Hüseyin Sağlam and John P. Gallagher. Approximating constraint logic programs using polymorphic types and regular descriptions. In Manuel Hermenegildo and S. Doaitse Swierstra, editors, Programming Languages: Implementations, Logics and Programs, pages 461–462, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

Zoltan Somogyi, Fergus Henderson, and Thomas C. Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. J. Log. Program., 29(1-3):17–64, 1996.

J. M. Talbot, S. Tison, and P. Devienne. Set-based analysis for logic programming and tree automata. In Pascal Van Hentenryck, editor, Static Analysis, pages 127–140. Springer Berlin Heidelberg, 1997.

Jeffrey D. Ullman. Principles of Database and Knowledge-base Systems. Computer Science Press, Inc., 1988.

Claudio Vaucheret and Francisco Bueno. More precise yet efficient type inference for logic programs. In Proceedings of the 9th International Symposium on Static Analysis, SAS '02, pages 102–116. Springer-Verlag, 2002.

Peter Lodewijk Van Roy. Can Logic Programming Execute as Fast as Imperative Programming? PhD thesis, EECS Department, University of California, Berkeley, Nov 1990.

Eyal Yardeni, Thom W. Frühwirth, and Ehud Shapiro. Polymorphically typed logic programs. In Frank Pfenning, editor, Types in Logic Programming, pages 63–90. The MIT Press, 1992.
Zob87. Justin Zobel. Derivation of polymorphic types for Prolog programs. In 
Logic Programming, Proceedings of the Fourth International Conference, 
Melbourne, 1987, 1987.
A Decidability

Lemma 1 (Equality Constraints). \textit{solve}\_eq always terminates and when it does, either it fails or the output is in normal form.

\textbf{Proof.} We will define the following metrics for \textit{solve}\_eq:

\begin{itemize}
  \item NVRS: number of variables on the right-hand side of equalities, that occur somewhere in another equality.
  \item NSE: number of symbols in the equalities.
  \item NENNF: number of equalities not in normal form.
\end{itemize}

We will prove termination by showing that NENNF reduces to zero. Termination of \textit{solve}\_eq is proven by a measure function that maps the constraint set to a tuple (NVRS,NSE,NENNF). The following table shows that each step decreases the tuple w.r.t. the lexicographical order of the tuple.

\begin{center}
\begin{tabular}{cccc}
  NVRS & NSE & NENNF \\
  \hline
  1. & $\geq$ & $>$ \\
  2. & $\geq$ & $>$ \\
  3. & $=$ & $=$ & $>$ \\
  4. & $=$ & $>$ \\
  5. & 0 & 0 & 0
\end{tabular}
\end{center}

Lemma 2 (Subtyping). \textit{solve}\_ineq always terminates.

\textbf{Proof.} We will define the following metrics for \textit{solve}\_ineq:

\begin{itemize}
  \item NPC: number of possible comparisons between types that have not been made yet.
  \item NVRSI: number of variables on the right-hand side of inequalities.
  \item NSI: number of symbols in the inequalities.
  \item NI: number of inequalities.
\end{itemize}

We will prove termination by showing that NI reduces to zero. Termination of \textit{solve}\_ineq is proven by a measure function that maps the constraint set to a tuple (NPC,NVRSI,NSI,NI). The following table shows that each step decreases the tuple w.r.t. the lexicographical order of the tuple.
Theorem 1 (Termination). \texttt{solve\_eq} and \texttt{solve\_ineq} always terminate. When \texttt{solve\_eq} terminates, it either fails or the output is in solved form. When \texttt{solve\_ineq} terminates, it either fails or the output is a pair with zero subtyping constraints.

\textit{Proof.} By lemma 1 and lemma 2, we prove this theorem.