Data Type Inference for Logic Programming

João Barbosa(✉), Mário Florido, and Vítor Santos Costa

Faculdade de Ciências, Universidade do Porto, 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 for 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 it is decidable and sound with respect to a type system, 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 as 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 [MO84,LR91]. Over the course of the last few years it has become clear that there is a need for a type inference system that can support Prolog well [SCWD08]. Next, we report on recent progress on our design, the YAP^T type system[1]. 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

[1] This work is partially funded by the portuguese Fundação para a Ciência e a Tecnologia and by LIACC (FCT/UID/CEC/0027/2020).

E. De Angelis and W. Vanhoof (Eds.): LOPSTR 2021, LNCS 13290, pp. 16–37, 2022.
https://doi.org/10.1007/978-3-030-98869-2_2

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. Thus our starting point was a three-valued semantics that clearly distinguishes type errors from falsehood [BFC19].

There, we first define the YAP^T *type system* that relates programs with their types. This defines the notion of *well-typed* program as a program which is related to a type by the relation defined by the type system. Here we present the YAP^T *type inference* algorithm which is able to automatically infer data type definitions. Finally we show that our type inference algorithm is sound with respect to the type system, in the sense that the inferred type for a program makes the program *well-typed*.

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 figure shows the output of type inference in this case, where `ti` is the type of the *i-th* argument of `append`, "+" means type disjunction and "A" and "B" are type variables (Fig. 1):

```
t1 = [] + [A | t1]          t1 = [] + [A | t1]
t2 = B                      t2 = [] + [A | t2]
t3 = B + [A | t3]           t3 = [] + [A | t3]
```

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

Types `t2` and `t3`, 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 `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 syntax of the programs 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* [Nai92] for the `append` predicate.

[2] Accordingly to a notion of *intended meaning* first presented in [Nai92].

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 types. We follow a syntax inspired in [SCWD08] to specify type information. One example of such a declaration is the list datatype

```
:- type list(A) = [] + [A | list(A)].
```

In order to simplify further processing our type system and type inference algorithm assume that predicates are in a simplified form called *kernel Prolog* [VR90]. In this representation, each predicate in the program 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. The variables in the head of the clause occur in every query in the body of that clause. We assume that there are no other 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 definition and thus type inference is easier to perform. In [VR90] a compilation from full Prolog to kernel Prolog is defined. Thus, in the rest of the paper, we will assume that predicate definitions are always in kernel Prolog.

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 set of type variables $TVar$, a finite set of base types $TBase$, an infinite set of constants $TCons$, an infinite set of function symbols $TFunc$, and an infinite set of type symbols, $TSymb$. *Type terms* can be:

- a type variable ($\alpha, \beta, \gamma, \cdots \in TVar$)
- a constant ($1, [\,], \text{'}c\text{'}, \cdots \in TCons$)
- a base type ($int, float, \cdots \in TBase$)
- a function symbol $f \in TFunc$ associated with an arity n applied to an n-tuple of type terms ($f(int, [\,], g(X))$)
- a type symbol $\sigma \in TSymb$ associated with an arity n ($n \geq 0$) applied to an n-tuple of type terms ($\sigma(X, int)$).

Type variables, constants and base types are called *basic types*. A *ground type term* is a type variable-free type term. Type symbols can be defined in a *type definition*. Type definitions are of the form:

$$\sigma(\alpha_1, \ldots, \alpha_k) = \tau_1 + \ldots + \tau_n,$$

[3] Implementation at https://github.com/JoaoLBarbosa/TypeInferenceAlgorithm.

where each τ_i is a type term and σ is the type symbol being defined. In general these definitions are polymorphic, which means that type variables $\alpha_1, \ldots, \alpha_k$, for $k \geq 0$, include the type variables occurring in $\tau_1 + \ldots + \tau_n$, and are called *type parameters*. If we instantiate one of those type variables, we can replace it in the parameters and everywhere it appears on the right-hand side of the definition. The sum $\tau_1 + \ldots + \tau_n$ is a *union type*, describing values that may have one of the types $\tau_1, \ldots, \tau_n$. The '+' is an idempotent, commutative, and associative operation. Throughout the paper, to condense notation, we will use the symbol $\bar{\tau}$ to denote union types. We will also use the notation $\tau \in \bar{\tau}$ to denote that τ is a summand in the union type $\bar{\tau}$.

Note that type definitions may be recursive. *A deterministic type definitions* is a type definition where, on the right-hand side, none of τ_i start with a type symbol and if τ_i is a type term starting with a function symbol f, then no other τ_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 $list = [\,] + [int \mid list]$[4].

Let $\overrightarrow{\tau}$ stand for a tuple of types $\tau_1 \times \cdots \times \tau_n$. A functional type is a type of the form $\overrightarrow{\tau_1} \rightarrow \tau_2$. A *predicate type* is a functional type from a tuple of the type terms defining the types of its arguments to bool, i.e. $\tau_1 \times \ldots \times \tau_n \rightarrow bool$. A *type* can be a *type term*, an *union type*, or a *predicate type*.

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, FSVY91, GdW94, YFS92, FD92, Han89]. Type schemes have type variables as generic place-holders for ground type terms. Parametric polymorphism comes from the fact these type variables can be instantiated with any type.

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

$$list(X) = [\,] + [X \mid list(X)]$$

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 definitions to be deterministic. The types described this way are tuple distributive.

[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:- `type` $type\_symbol(type\_vars) = type\_term_1 + \ldots + type\_term_n$. One example would be:

```
:- 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 Semantics

In [BFC19] we defined a formal semantics for types. Here we just give the main intuitive ideas behind it:

- The semantics of base types and constant types are predefined sets containing logic terms, for instance, the base type *int* is the set of all integers and the semantics of *bool* is the set of the values $true$ and $false$;
- Tuples of types, $(\tau_1, \ldots, \tau_n)$, are sets of tuples of terms such that the semantics of each term belongs to the semantics of the type in the corresponding position;
- $f(\tau_1, \ldots, \tau_n)$ is the set of all terms with main functor f and arity n applied to the set of tuples belonging to the semantics of $(\tau_1, \ldots, \tau_n)$;
- The semantics of union types is the disjoint union of the semantics of its summands;
- The semantics of type symbols is the set of all terms that can be derived from its definition;
- The semantics of functional types, such as predicate types, is the set of functions that when given terms belonging to the semantics of the input types, output terms belonging to the semantics of output types. For instance the semantics of $int \times float \rightarrow bool$, contains all functions that, given a pair with an integer and a floating point number, output a boolean.
- The semantics of parametric polymorphic types is the intersection of the semantics of its instances (this idea was first used by Damas [Dam84] to define the semantics of type schemes).

In [BFC19] we defined a type system and proved that it is sound with respect to this semantics of types. Here we define a type inference algorithm and prove that it is sound with respect to the type system, thus, using these two results, we can conclude that the type inference algorithm is also sound with respect to the semantics.

2.2 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 α is* unconstrained *with respect to a set of type definitions T, notation* unconstrained(α, T)*, if and only if it occurs exactly once as a summand 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 type definition*, which are type definitions without type variables as summands in their definition.

Definition 2 (Closed Type Definitions). *A type definition $\sigma = \bar{\tau}$ is* closed*, notation* closedTypeDef(σ)*, if and only if there are no type variables as summands in $\bar{\tau}$.*

The definition for closed types uses these two previous auxiliary definitions. Closed types correspond to closed records or data definitions in functional programming languages. The definition follows:

Definition 3 (Closed Types). *A type definition $\sigma = \bar{\tau}$ is* closed *with respect to a set of type definitions T, notation* closed(σ, T)*, if and only if the predicate defined as follows holds:*

$$closed(\sigma, T) = \begin{cases} \neg unconstrained(\alpha, T) & if\ \bar{\tau} = \alpha\ and\ \alpha\ is\ a\ type\ variable \\ closedTypeDef(\sigma) & otherwise \end{cases}$$

Example 3. We recall the example in the Introduction, of the following types for the `append` predicate, where t_n is the type of *nth* predicate argument:

```
t1 = [] + [A | t1]
t2 = B
t3 = B + [A | t3]
```

Type `t3`, for the third argument of `append`, is open, because `t3` 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 `append` is:

```
t1 = [] + [A | t1]
t2 = [] + [A | t2]
t3 = [] + [A | t3]
```

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 an optional post-processing step on our algorithm.

To close types, we calculate what we call the proper variable domain of every type variable that occurs as a summand in a type definition. The proper variable domain corresponds to the sum of the proper domains of each type that shares a type term with the open type we are tying to close. The proper domain of a type is the sum of all summands that are not type variables in its definition. We then replace the type variable with its proper variable domain.

We have tested the closure algorithm on several examples and for the examples we tried, the results seem very promising.

3 Examples

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

- *basetype* (default: *on*) - when this flag is turned on, we assume that each constant is typed with a base type, when it is turned off, we type each constant with a constant type corresponding to itself;
- *list* (default: *off*) - this flag adds the data type declaration for polymorphic lists to the program when turned on;
- *closure* (default: *off*) - when this flag is turned on, the closure operation is applied as a post-processing step on the algorithm.

In the following examples `pi` is the type symbol for the type of the *ith* 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 $int + float$.

Example 4. Let us consider the predicate *concat*, which flattens a list of lists, where *app* is the *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 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 towards the automatic inference of types for programs used 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* definition used in the previous example:

```
rev(A, B) :- A=[], B=[] ;
        rev(C, D), app(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:

```
rev :: rev1 x rev2
rev1 = list(A)
rev2 = list(A)

list(X) = [] + [ X | list(X) ]
```

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

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

```
tree_min(A,B) :- A=empty, B=0 ;
        A=node(C,D,E), tree_min(D,F), tree_min(E,G),
        Y=[C,F,G], minimum(Y,X), X=B.

minimum(A,B) :- A=[I], B=I;
        A=[X|Xs], minimum(Xs,C), X=<C, B=C ;
        A=[Y|Ys], minimum(Ys,D), D=<Y, B=D.
```

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

```
tree_min :: tree_min1 x tree_min2
tree_min1 = atom + node(tree_min2, tree_min1, tree_min1)
tree_min2 = A + int + float

minimum :: minimum1 x minimum2
minimum1 = [ minimum2 | t ]
minimum2 = A + int + float
t2 = [] + [ minimum2 | t2 ]
```

If we now add a predefined declaration of a tree data type and turn on the *list_flag*, the algorithm outputs:

```
tree_minimum :: tree_minimum1 x tree_minimum2
tree_minimum1 = tree(tree_minimum2)
tree_minimum2 = int + float

minimum :: minimum1 x minimum2
minimum1 = list(minimum2)
minimum2 = int + float

tree(X) = empty + node(X, tree(X), tree(X))
list(Y) = [] + [ Y | list(Y) ]
```

4 Type System

Here we define the notion of *well-typed* program using a set of rules assigning types to terms, atoms, queries, sequences of queries, and clauses. This is generally called a *type system* and ours follows the definition in [BFC19] with some differences in the notation for recursive types: here we explicitly add a set of (possibly recursive) type definitions instead of the fix-point notation for types used in the paper mentioned above. These small differences do not alter the soundness of the type system.

We first write the following subtyping relation from [BFC19].

Definition 4 (Subtyping). *Let ϕ be a substitution of types for type variables. Let $\sqsubseteq$ denote the subtyping relation as a partial order (reflexive, anti-symmetric and transitive) defined as follows:*

- $\tau \sqsubseteq \tau\prime$ *if* $\exists\phi.\phi(\tau\prime) = \tau$ *(Instance)*
- $\tau \sqsubseteq \bar{\tau}$ *iff* $\tau \in \bar{\tau}$ *(Subset)*
- $f(\tau_1, \ldots, \tau_n) \sqsubseteq f(\tau\prime_1, \ldots, \tau\prime_n)$ *iff* $\tau_1 \leq \tau\prime_1, \ldots \tau_n \leq \tau\prime_n$ *(Complex term construction/destruction)*
- $\delta_1 \sqsubseteq \delta_2$ *iff, assuming* $\delta_1 \sqsubseteq \delta_2$*, we get* $\bar{\tau}_1 \sqsubseteq \bar{\tau}_2$*, where* $\delta_1 = \bar{\tau}_1$ *and* $\delta_2 = \bar{\tau}_2$ *are the type definitions for* δ_1 *and* δ_2 *(Recursive Type Unfolding)*
- $\tau \sqsubseteq \delta$ *iff* $\tau \sqsubseteq \bar{\tau}$ *and* $\delta = \bar{\tau}$ *is the type definition for* δ *(Right Unfolding)*
- $\delta \sqsubseteq \tau$ *iff* $(\tau) \sqsubseteq \tau$ *and* $\delta = \bar{\tau}$ *is the type definition for* δ *(Left Unfolding)*
- $\tau \sqsubseteq \tau_1 + \tau_2$ *iff* $\tau \sqsubseteq \tau_1$ *or* $\tau_2 \sqsubseteq \tau$ *(Addition)*
- *if* $\tau\prime \sqsubseteq \tau$*, then* $\tau \rightarrow bool \sqsubseteq \tau\prime \rightarrow bool$ *(Contravariance)*

Subtyping of functional types is contravariant in the argument type, meaning that the order of subtyping is reversed. This is standard in functional languages and guarantees that when a function type is a subtype of another it is safe to use a function of one type in a context that expects a function of a different type. It is safe to substitute a function f for a function g if f accepts a more general type of argument than g. For example, predicates of type $int + float \rightarrow bool$ can be used wherever an $int \rightarrow bool$ was expected.

Let us now give some auxiliary definitions: an *assumption* is a type declaration for a variable, written $X : \tau$, where X is a variable and τ a type. We define a *context* Γ as a set of assumptions with distinct variables as subjects (alternatively *contexts* can be defined as functions from variables to types, where $domain(\Gamma)$ stands for its domain). Since Γ can be seen as a function, we use $\Gamma(X) = \tau$ to denote $(X : \tau) \in \Gamma$. A set of type definitions, Δ, is a set of type definitions of the form $\sigma = \bar{\tau}$, where each definition has a different type symbol on the left-hand side. It can also be defined as a function from σ to $\bar{\tau}$. We will therefore use the notation $\Delta(\sigma) = \bar{\tau}$ to denote $(\sigma = \bar{\tau}) \in \Delta$.

Our type system is defined in Fig. 2 and statically relates *well-typed* programs with their types by defining a relation $\Gamma, \Delta \vdash_P p : \tau$, where Γ is a context, Δ a set of type definitions, p is a term, an atom, a query, a sequence of queries, or a clause, and τ is a type. This relation should be read as expression p has type τ, given the context Γ and type definitions Δ, in a program P. We will write $\Gamma \cup \{X : \tau\}$ to represent the context that contains all assumptions in Γ and the additional assumption $X : \tau$ (note that because each variable is unique as a subject of an assumption in a context, in $\Gamma \cup \{X : \tau\}$, Γ does not contain assumptions with X as subject). We will write a sequence of variables $X_1, \ldots, X_n$ as $\overrightarrow{X}$, and a sequence of types as $\overrightarrow{\tau}$. We assume that clauses are normalized and, therefore, every call to a predicate in the body of a clause contains only variables.

Note that we have a different rule for recursive clauses and non-recursive clauses. Whenever we have a recursive clause, its type is derived assuming every recursive call has the same type as the head of the clause. This corresponds to the monomorphic restriction described in [Hen93], where the authors prove that

$$VAR \qquad \Gamma \cup \{X : \sigma\}, \Delta \cup \{\sigma = \tau_1 + \ldots + \tau_n\} \vdash_P X : \tau_i$$

$$CST \qquad \Gamma, \Delta \vdash_P c : \tau, \text{ where } basetype(c) = \tau$$

$$CPL^{(a)} \quad \frac{\Gamma, \Delta \vdash_P t_1 : \tau_1 \quad \ldots \quad \Gamma, \Delta \vdash_P t_n : \tau_n}{\Gamma, \Delta \vdash_P f(t_1, \ldots, t_n) : \tau}$$

$$UNF \quad \frac{\Gamma, \Delta \vdash_P t_1 : \tau \quad \Gamma, \Delta \vdash_P t_2 : \tau}{\Gamma, \Delta \vdash_P t_1 = t_2 : bool}$$

$$CLL^{(b)} \quad \frac{\Gamma \cup \{\overline{Y} : \overrightarrow{\bar{\sigma}\prime}\}, \Delta \vdash_P (p(Y_1, \ldots, Y_n) : -body.) : bool}{\Gamma \cup \{\overline{X} : \bar{\sigma}\}, \Delta \cup \{\bar{\sigma} = \bar{\tau}\} \vdash_P p(X_1, \ldots, X_n) : bool}, \text{ where } \forall i. \sigma_i \sqsubseteq \sigma_i\prime$$

$$CON \quad \frac{\Gamma, \Delta \vdash_P g_1 : bool \quad \Gamma, \Delta \vdash_P g_n : bool}{\Gamma, \Delta \vdash_P g_1, \ldots, g_n : bool}$$

$$CLS^{(c)} \quad \frac{\Gamma, \Delta \vdash_P b_1 : bool \quad \ldots \quad \Gamma, \Delta \vdash_P b_m : bool}{\Gamma, \Delta \vdash_P (p(\overrightarrow{X}) : -b_1; \ldots; b_m.) : bool}$$

$$RCLS^{(d)} \quad \frac{\Gamma \cup \{\overline{X} : \overrightarrow{\tau}, \overrightarrow{\overline{Y_1}} : \overline{\tau}, \ldots, \overrightarrow{\overline{Y_{k_n}}} : \overline{\tau}\}, \overrightarrow{\Delta} \vdash_P p(\overline{X}) : \overrightarrow{-b_1; \ldots; b_{m+n}.} : bool}{\begin{array}{l}\Gamma \cup \{\overline{X} : \overrightarrow{\tau}, \overrightarrow{\overline{Y_1}} : \overline{\tau}, \ldots, \overrightarrow{\overline{Y_{k_n}}} : \overline{\tau}\}, \overrightarrow{\Delta} \vdash_P \\ \quad (p(\overline{X}) : -b_1; \ldots; b_m; \\ \qquad b_{m+1}, p(\overline{Y}_{11}\overrightarrow{)}, \ldots, p(\overline{Y}_{1k_1}\overrightarrow{)}; \\ \qquad\qquad \vdots \\ \qquad b_{m+n}, p(\overline{Y}_{n1}\overrightarrow{)}, \ldots, p(\overline{Y}_{nk_n}\overrightarrow{)}.) : bool\end{array}}$$

(a) Where $basetype(f) = \tau_1\prime \times \ldots \times \tau_n\prime \to \tau$, and $\tau_i \sqsubseteq \tau_i\prime$.
(b) Where the clause defining predicate p is in P.
(c) This rule is for non-recursive predicates only.
(d) This rule is for recursive predicates. Note that all variables in recursive calls in a certain sequence of goals have the same type as the variables in the head in that clause.

Fig. 2. Type System

if we allow polymorphic recursion, i.e. recursion with different instances of the same polymorphic type, then inference is not decidable.

Also note that the type for a predicate call is a subtype of the type for the clause defining it. This captures the fact that we can call a polymorphic predicate with a type that is an instance of the general type scheme, or if the input type of the predicate is a union type, we can call it with only some of the summands of that union.

5 Type Inference

We have seen how to define the notion of *well-typed* program using a set of rules which assign types to programs. Here we will present a type inference algorithm which, given an untyped logic program, is able to calculate a type which makes the program well-typed.

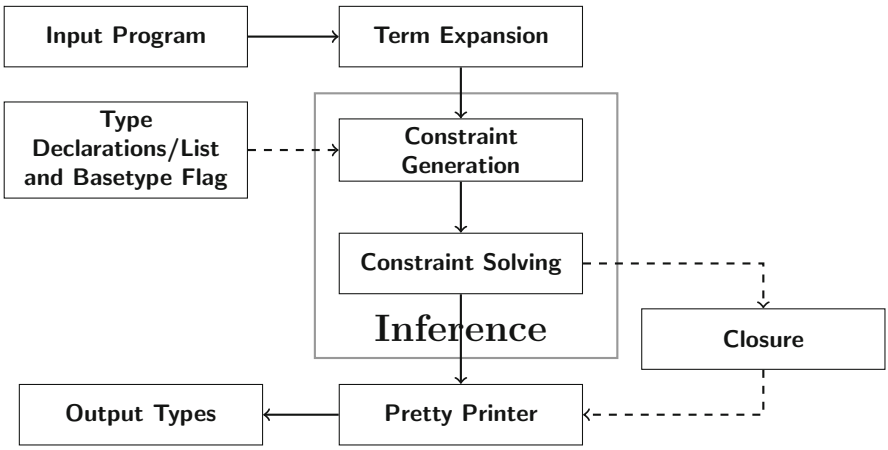

Fig. 3. Type Inference Algorithm Flowchart

Our type inference algorithm is composed of several modules, as described in Fig. 3. 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 constraint generation is performed, and a type constraint solver outputs the inferred types for a given program. There is also a simplification step that is performed during inference, to assure that the type definitions are always deterministic and simplified. After this, we either directly run 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).

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. Using one of the latter approaches, instead of the standard one, yields better results as can be seen in the example Sect. 3.

5.1 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 5 (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].

5.2 Constraints and Constraint Generation

The type inference algorithm begins by generating type constraints from a logic program which are 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. An equality constraint is of the form $\tau_1 = \tau_2$ and a subtyping constraint is of the form $\bar{\tau_1} \leq \bar{\tau_2}$. Ultimately we want to determine if a set of constraints C can be instantiated affirmatively using some substitution S, that substitutes types for type variables. For this we need to consider a notion of *constraint satisfaction* $S \models C$, in a first order theory with equality [Mah88] and the extra axioms in Definition 4 for subtyping.

Definition 6 (Constraint satisfaction). *Let* $\equiv$ *mean syntactic type equality and* $\sqsubseteq$ *the subtyping relation defined in Definition 4.* $S \models C$ *is defined as follows:*

1. $S \models \tau_1 = \tau_2$ *if and only if* $S(\tau_1) \equiv S(\tau_2)$;
2. $S \models \bar{\tau_1} \leq \bar{\tau_2}$ *if and only if* $S(\bar{\tau_1}) \sqsubseteq S(\bar{\tau_2})$;
3. $S \models C$ *if and only if* $S \models c$ *for each constraint* $c \in C$.

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.

Let us first present two auxiliary functions to combine contexts. Contexts can be obtained from the disjunction, or conjunction, of other contexts. For this we define two auxiliary functions, $\oplus$ and $\otimes$, to define the result of disjunction, or conjunction, respectively, of context. These definitions are used by the constraint generation algorithm. They are defined as follows:

Definition 7. *Let* Γ_i *be contexts, and* Δ_i *be disjoint sets of type definitions defining the type symbols in* Γ_i*, respectively. Let* V *be the set of variables that occur in more than one context.*
$\oplus\big((\Gamma_1, \ldots, \Gamma_n), (\Delta_1, \ldots, \Delta_n)\big) = (\Gamma, \Delta)$*, where:*

$\Gamma(X) = \sigma\prime$, where $\sigma\prime$ is a fresh type symbol, for all $X \in V$, and $\Gamma(X) = \Gamma_i(X)$, for all $X \notin V \wedge X \in domain(\Gamma_i)$;
$\Delta(\sigma) = \Gamma_{i_1}(X) + \ldots + \Gamma_{i_k}(X)$, for all type symbols $\sigma \notin \Delta_1 \cup \cdots \cup \Delta_n$, and $\Delta(\sigma) = \Delta_i(\sigma)$, otherwise.

Definition 8. *Let Γ_i be contexts, and Δ_i be disjoint sets of type definitions defining the type symbols in Γ_i, respectively. Let V be the set of variables that occur in more than one context.*
$\otimes\big((\Gamma_1, \ldots, \Gamma_n), (\Delta_1, \ldots, \Delta_n)\big) = (\Gamma, \Delta, Eq)$, where:
$\Gamma(X) = \sigma\prime$, where $\sigma\prime$ is a fresh type symbol, for all $X \in V$, and $\Gamma(X) = \Gamma_i(X)$, for all $X \notin V \wedge X \in domain(\Gamma_i)$;
$\Delta(\sigma) = \alpha$, where α is a fresh type variable, for all type symbols $\sigma \notin \Delta_1 \cup \cdots \cup \Delta_n$, and $\Delta(\sigma) = \Delta_i(\sigma)$, otherwise;
$Eq = \{\alpha = \Delta_i(\Gamma_i(X)), \ldots, \alpha = \Delta_j(\Gamma_j(X))\}$, for all fresh α, such that $(\sigma\prime = \alpha) \in \Delta$, $\Gamma(X) = \sigma\prime$, and $X \in domain(\Gamma_i) \wedge \cdots \wedge X \in domain(\Gamma_j)$.

Let P be a term, an atom, a query, a sequence of queries, or a clause. $generate(P)$ is a function that outputs a tuple of the form $(\tau, \Gamma, Eq, Ineq, \Delta)$, where τ is a type, Γ is an context for variables, Eq is a set of equality constraints, $Ineq$ is a set of subtyping constraints, and Δ is a set of type definitions. The function *generate*, which generates the initial type constraints, is defined case by case from the program syntax. Its definition follows:

$generate(P) =$

- $generate(X) = (\alpha, \{X : \sigma\}, \emptyset, \emptyset, \{\sigma = \alpha\})$, X is a variable,
  where α is a fresh type variable and σ is a fresh type symbol.
- $generate(c) = (basetype(c), \emptyset, \emptyset, \emptyset, \emptyset)$, c is a constant.
- $generate(f(t_1, \ldots, t_n)) = (basetype(f)(\tau_1, \ldots, \tau_n), \Gamma, Eq, \emptyset, \Delta)$, f is a function symbol,
  where $generate(t_i) = (\tau_i, \Gamma_i, Eq_i, \emptyset, \Delta_i)$,
  $(\Gamma, \Delta, Eq\prime) = \otimes\big((\Gamma_1, \ldots, \Gamma_n), (\Delta_1, \ldots, \Delta_n)\big)$, and
  $Eq = Eq_1 \cup \ldots \cup Eq_n \cup Eq\prime$.
- $generate(t_1 = t_2) = (bool, \Gamma, Eq, \emptyset, \Delta)$
  where $generate(ti) = (\tau_i, \Gamma_i, Eq_i, \emptyset, \Delta_i)$,
  $(\Gamma, \Delta, Eq\prime) = \otimes\big((\Gamma_1, \Gamma_2), (\Delta_1, \Delta_2)\big)$, and
  $Eq = Eq_1 \cup Eq_2 \cup \{\tau_1 = \tau_2\} \cup Eq\prime$.
- $generate(p(X_1, \ldots, X_n)) = (bool, (\{X_1 : \sigma_1, \ldots, X_n : \sigma_n\}, \emptyset, \{\sigma_1 \leq \tau_1, \ldots, \sigma_n \leq \tau_n\}, \Delta\prime)$, p is a predicate symbol,
  where $generate(p(Y_1, \ldots, Y_n) : -body) = (bool, \Gamma, Eq, Ineq, \Delta)$,
  $\{Y_1 : \tau_1, \ldots Y_n : \tau_n\} \in \Gamma$
  $\Delta\prime = \Delta \cup \{\sigma_i = \alpha_i\}$, and σ_i and α_i are all fresh.
- $generate(c_1, \ldots, c_n) = (bool, \Gamma, Eq, Ineq_1 \cup \ldots \cup Ineq_n, \Delta)$, a query,
  where $generate(c_i) = (bool, \Gamma_i, Eq_i, Ineq_i, \Delta_i)$,
  $(\Gamma, \Delta, Eq\prime) = \otimes\big((\Gamma_1, \ldots, \Gamma_n), (\Delta_1, \ldots, \Delta_n)\big)$, and
  $Eq = Eq_1 \cup \ldots \cup Eq_n \cup Eq\prime$.

- $generate(b_1; \ldots; b_n) = (bool, \Gamma, Eq_1 \cup \ldots \cup Eq_n, Ineq_1 \cup \ldots \cup Ineq_n, \Delta)$, where $generate(c_i) = (bool, \Gamma_i, Eq_i, Ineq_i, \Delta_i)$, and $(\Gamma, \Delta) = \oplus\big((\Gamma_1, \ldots, \Gamma_n), (\Delta_1, \ldots, \Delta_n)\big)$.
- $generate(p(X_1, \ldots, X_n) : -body.) = (bool, \Gamma, Eq, Ineq, \Delta)$, a non-recursive clause,
  where $generate(body) = (bool, \Gamma, Eq, Ineq, \Delta)$.
- $generate(p(X_1, \ldots, X_n) : -body) = (bool, \Gamma, Eq, Ineq\prime, \Delta)$, a recursive clause, where $generate(p(X_1, \ldots, X_n) : -body\prime) = (bool, \Gamma, Eq, Ineq, \Delta)$, such that $body\prime$ is $body$ after removing all recursive calls,
  and $Ineq\prime = Ineq \cup \{\overrightarrow{\sigma_1} \leq \overrightarrow{\tau}, \ldots, \overrightarrow{\sigma_k} \leq \overrightarrow{\tau}, \overrightarrow{\tau} \leq \overrightarrow{\sigma_1}, \ldots, \overrightarrow{\tau} \leq \overrightarrow{\sigma_k}\}$, such that τ are the types for the variables in the head of the clause in Γ and σ_i are the types for the variables in each recursive call.

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:
$generate(list(X) : -X = [\,]; X = [Y|Ys], list(Ys)) = \{bool, \{X : \sigma_1, Y : \sigma_2, Ys : \sigma_3\}, \{\alpha = [\,], \beta = [\delta \mid \epsilon]\}, \{\sigma_3 \leq \sigma_1, \sigma_1 \leq \sigma_3\}, \{\sigma_1 = \alpha + \beta, \sigma_2 = \delta, \sigma_3 = \epsilon\}\}$.

The set $\{\sigma_3 \leq \sigma_1, \sigma_1 \leq \sigma_3\}$ comes from the recursive call to the predicate, while $\alpha = [\,]$ comes from $X = [\,]$, and $\beta = [\delta \mid \epsilon]$ comes from $X = [Y|Ys]$. The definition $\sigma_1 = \alpha + \beta$ comes from the application of the $\oplus$ operation.

5.3 Constraint Solving

Let Eq be a set of equality constraints, $Ineq$ be a set of subtyping constraints, and Δ a set of type definitions. Function $solve(Eq, Ineq, \Delta)$ is a rewriting algorithm that solves the constraints, outputting a pair of a substitution and a new set of type definitions. Note that the rewriting rules in the following definitions of the solver algorithm are assumed to be ordered.

Definition 9. *A set of equality constraints is in* solved form *if:*

- *all constraints are of the form $\alpha_i = \tau_i$;*
- *there are no two constraints with the same α_i on the left hand side;*
- *no type variables on the left-hand side of the equations occurs on the right-hand side of equations.*

A set of equality constraints in normal form can be interpreted as a substitution, where each constraint $\alpha_i = \tau_i$ corresponds to a substitution for the type variable α_i, $[\alpha_i \mapsto \tau_i]$.

A *configuration* is either the term *fail* (representing failure), a pair of a substitution and a set of type definitions (representing the end of the algorithm), or a triple of a set of equality constraints Eq, a set of subtyping constraints $Ineq$, and a set of type definitions Δ. The following rewriting algorithm consists of the transformation rules on configurations.

$solve(Eq, Ineq, \Delta) =$

1. $(\{\tau = \tau\} \cup Eq, Ineq, \Delta) \rightarrow (Eq, Ineq, \Delta)$
2. $(\{\alpha = \tau\} \cup Eq, Ineq, \Delta) \rightarrow (\{\alpha = \tau\} \cup Eq[\alpha \mapsto \tau], Ineq[\alpha \mapsto \tau], \Delta[\alpha \mapsto \tau])$, if type variable α occurs in Eq, $Ineq$, or Δ
3. $(\{\tau = \alpha\} \cup Eq, Ineq, \Delta) \rightarrow (\{\alpha = \tau\} \cup Eq, Ineq, \Delta)$, where α is a type variable and τ is not a type variable
4. $(\{f(\tau_1, \ldots, \tau_n) = f(\tau\prime_1, \ldots, \tau\prime_n)\} \cup Eq, Ineq, \Delta \rightarrow (\{\tau_1 = \tau\prime_1, \ldots, \tau_n = \tau\prime_n\} \cup Eq, Ineq, \Delta$
5. $(\{f(\tau_1, \ldots, \tau_n) = g(\tau\prime_1, \ldots, \tau\prime_m)\} \cup Eq, Ineq, \Delta) \rightarrow fail$
6. $(Eq, \{\tau \leq \tau\} \cup Ineq, \Delta) \rightarrow (Eq, Ineq, \Delta)$
7. $(Eq, \{f(\tau_1, \ldots, \tau_n) \leq f(\tau\prime_1, \ldots, \tau\prime_n)\} \cup Ineq, \Delta) \rightarrow (Eq, \{\tau_1 \leq \tau\prime_1, \ldots \tau_n \leq \tau\prime_n\} \cup Ineq, \Delta)$
8. $(Eq, \{\alpha \leq \tau_1, \ldots, \alpha \leq \tau_n\} \cup Ineq, \Delta) \rightarrow (Eq \cup Eq\prime, \{\alpha \leq \tau\} \cup Ineq, \Delta\prime)$, where α is a type variable, $n \geq 2$, and $intersect(\tau_1, \ldots, \tau_n, \Delta, I) = (\tau, Eq\prime, \Delta\prime)$
9. $(Eq, \{\alpha \leq \tau\} \cup Ineq, \Delta) \rightarrow (Eq \cup \{\alpha = \tau\}, Ineq, \Delta)$, where α is a type variable and no other constraints exist with α on the left-hand side
10. $(Eq, \{\tau_1 + \ldots + \tau_n \leq \tau\} \cup Ineq, \Delta) \rightarrow (Eq, \{\tau_1 \leq \tau, \ldots, \tau_n \leq \tau\} \cup Ineq, \Delta)$
11. $(Eq, \{\sigma \leq \tau\} \cup Ineq, \Delta) \rightarrow (Eq, Ineq, \Delta)$, if (σ,τ) are on the store of pairs of types that have already been compared
12. $(Eq, \{\sigma \leq \tau\} \cup Ineq, \Delta) \rightarrow (Eq, \{Rhs_\sigma \leq \tau\} \cup Ineq, \Delta)$, where σ is a type symbol, and $\sigma = Rhs_\sigma \in \Delta$. Also add (σ, τ) to the store of pairs of types that have been compared
13. $(Eq, \{\tau_1 \leq \alpha, \ldots \tau_n \leq \alpha\} \cup Ineq, \Delta) \rightarrow (Eq \cup \{\alpha = \tau_1 + \ldots + \tau_n\}, Ineq, \Delta)$
14. $(Eq, \{\tau \leq \tau_1 + \ldots + \tau_n\} \cup Ineq, \Delta) \rightarrow (Eq, \{\tau \leq \tau_i\} \cup Ineq, \Delta)$, where τ_i is one of the summands
15. $(Eq, \{\tau \leq \sigma\} \cup Ineq, \Delta) \rightarrow (Eq, Ineq, \Delta)$, if (σ,τ) are on the store of pairs of types that have already been compared
16. $(Eq, \{\tau \leq \sigma\} \cup Ineq, \Delta) \rightarrow (Eq, \{\tau \leq Rhs_\sigma\} \cup Ineq, \Delta)$, where σ is a type symbol, and $\sigma = Rhs_\sigma \in \Delta$. Also add (σ, τ) to the store of pairs of types that have been compared
17. $(Eq, \emptyset, \Delta) \rightarrow (Eq, \Delta\prime)$
18. otherwise $\rightarrow$ fail.

Note that an occur check is required in steps 2, 9, and 13. This rewriting algorithm is based on the one described in [Mah88] for equality constraints, and an original one for the subtyping constraints. We will now show an example of the execution of the algorithm on the output of the constraint generation algorithm, showed in Example 7.

Example 8. Following Example 7, applying *solve* to the tuple $(Eq, Ineq, \Delta)$, corresponding to $(\{X : \sigma_1, Y : \sigma_2, Ys : \sigma_3\}, \{\alpha = [\,], \beta = [\delta \mid \epsilon]\}, \{\sigma_3 \leq \sigma_1, \sigma_1 \leq \sigma_3\}, \{\sigma_1 = \alpha + \beta, \sigma_2 = \delta, \sigma_3 = \epsilon\})$:
$(\{\alpha = [\,], \beta = [\delta \mid \epsilon]\}, \{\sigma_3 \leq \sigma_1, \sigma_1 \leq \sigma_3\}, \{\sigma_1 = \alpha + \beta, \sigma_2 = \delta, \sigma_3 = \epsilon\}) \rightarrow_2$
$(\{\alpha = [\,], \beta = [\delta \mid \epsilon]\}, \{\sigma_3 \leq \sigma_1, \sigma_1 \leq \sigma_3\}, \{\sigma_1 = [\,] + \beta, \sigma_2 = \delta, \sigma_3 = \epsilon\}) \rightarrow_2$

$(\{\alpha = [\,], \beta = [\delta \mid \epsilon]\}, \{\sigma_3 \leq \sigma_1, \sigma_1 \leq \sigma_3\}, \{\sigma_1 = [\,] + [\delta \mid \epsilon], \sigma_2 = \delta, \sigma_3 = \epsilon\}) \rightarrow_{12}$
$(\{\alpha = [\,], \beta = [\delta \mid \epsilon]\}, \{\epsilon \leq \sigma_1, \sigma_1 \leq \sigma_3\}, \{\sigma_1 = [\,] + [\delta \mid \epsilon], \sigma_2 = \delta, \sigma_3 = \epsilon\}) \rightarrow_{9}$
$(\{\alpha = [\,], \beta = [\delta \mid \epsilon], \epsilon = \sigma_1\}, \{\sigma_1 \leq \sigma_3\}, \{\sigma_1 = [\,] + [\delta \mid \epsilon], \sigma_2 = \delta, \sigma_3 = \epsilon\}) \rightarrow_{2}$
$(\{\alpha = [\,], \beta = [\delta \mid \sigma_1], \epsilon = \sigma_1\}, \{\sigma_1 \leq \sigma_3\}, \{\sigma_1 = [\,] + [\delta \mid \sigma_1], \sigma_2 = \delta, \sigma_3 = \sigma_1\}) \rightarrow_{s}$
$(\{\alpha = [\,], \beta = [\delta \mid \sigma_1], \epsilon = \sigma_1\}, \{\sigma_1 \leq \sigma_1\}, \{\sigma_1 = [\,] + [\delta \mid \sigma_1], \sigma_2 = \delta\}) \rightarrow_{6}$
$(\{\alpha = [\,], \beta = [\delta \mid \sigma_1], \epsilon = \sigma_1\}, \emptyset, \{\sigma_1 = [\,] + [\delta \mid \sigma_1], \sigma_2 = \delta\})$

Note that the resulting set of constraints only contains constraints in solved form, that can be seen as a substitution. Step $\rightarrow_s$, stands for the following simplification step: if two type definitions are equal, we delete one of them and replace every occurrence of the type symbol by the other. Therefore, the resulting context Γ is $\{X : \sigma_1, Y : \sigma_2, Ys : \sigma_1\}$.

Type intersection is calculated as follows, $intersect(\tau_1, \tau_2, \Delta, I) = (\tau, Eq\prime, \Delta\prime)$, where:

- if both τ_1 and τ_2 are type variables, then $\tau = \tau_2, \Delta\prime = \Delta, Eq\prime = \{\tau_1 = \tau_2\}$.
- if $\tau_1 = \tau_2$, then $\tau = \tau_1, \Delta\prime = \Delta, Eq\prime = \emptyset$.
- if $(\tau_1, \tau_2, \tau_3) \in I$, then $\tau = \tau_3, \Delta\prime = \Delta, Eq\prime = \emptyset$.
- if τ_1 is a type variable, then $\tau = \tau_2, \Delta\prime = \Delta, Eq\prime = \emptyset$.
- if τ_2 is a type variable, then $\tau = \tau_1, \Delta\prime = \Delta, Eq\prime = \emptyset$.
- if $\tau_1 = \sigma_1, \tau_2 = \sigma_2$, and $(\bar{\tau}, Eq, \Delta_2) = cpi(\bar{\tau_1}, \bar{\tau_2}, \Delta, I \cup \{(\tau_1, \tau_2, \tau_3)\})$, then $\tau = \tau_3, \Delta\prime = \Delta_2 \cup \{\tau_3 = \bar{\tau}\}, Eq\prime = Eq$, where $\sigma_1 = \bar{\tau_1}, \sigma_2 = \bar{\tau_2} \in \Delta$ and τ_3 is fresh.
- if $\tau_1 = \sigma_1, \tau_2 = f(t_1, \ldots, t_n)$, and $(\bar{\tau}, Eq, \Delta_2) = cpi(\bar{\tau}, \tau_2, \Delta, I \cup \{(\tau_1, \tau_2, \tau_3)\})$, then $\tau = \tau_3, \Delta\prime = \Delta \cup \{\tau_3 = \bar{\tau}\}, Eq\prime = Eq$, where $\sigma_1 = \bar{\tau_1} \in \Delta$ and τ_3 is fresh. Same for $\tau_2 = \sigma_1$ and $\tau_1 = f(t_1, \ldots, t_n)$.
- if $\tau_1 = f(\tau_1, \ldots, \tau_n), \tau_2 = f(\tau\prime_1, \ldots, \tau\prime_n)$, then $\forall i.1 \leq i \leq n$, $(\tau\prime\prime_i, Eq_i, \Delta_i) = intersect(\tau_i, \tau\prime_i, \Delta, I)$, $\tau = f(\tau\prime\prime_1, \ldots, \tau\prime\prime_n), \Delta\prime = \Delta_1 \cup \ldots \cup \Delta_n, Eq\prime = Eq_1 \cup \ldots \cup Eq_n$.
- otherwise fail.

$cpi(\bar{\tau_1}, \bar{\tau_2}, \Delta, I)$ is a function that applies $intersect(\tau, \tau\prime, \Delta, I)$ to every pair of types $\tau, \tau\prime$, such that $\tau \in \bar{\tau_1}$ and $\tau\prime \in \bar{\tau_2}$, and gathers all results as the output.

This intersection algorithm is based on the one presented in [Zob87], with a few minor changes. The difference is that our types can be type variables, which could not happen in Zobel's algorithm, since intersection was only calculated between ground types. To deal with this extension, in our algorithm type variables are treated as Zobel's *any* type, except when both types are type variables, in which case we also unify them. Termination and correctness of type intersection for a tuple distributive version of Zobel's algorithm was proved previously in [Lu01] and replacing the *any* type with type variables maintains the same properties, because our use of type intersection considers types where type variables occur only once, thus they can be safely replaced by Zobel's *any* type. Note that we deal with type variables which occur more than once but with calls to type unification.

5.4 Decidability

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

Theorem 1 (Termination). *solve always terminates, and when solve terminates, it either fails or the output is a pair of a substitution and a new set of type definitions.*

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

To guarantee that the output set of equality constraints is in normal form, in order to be interpreted as a substitution, we also prove the lemma below.

Lemma 1. *If* $solve(Eq, Ineq, \Delta) \rightarrow^* (S, \Delta\prime)$*, then* S *is in normal form.*

5.5 Soundness

Here we prove that the type inference algorithm is sound, in the sense that inferred types are derivable in the type system, which defines well-typed programs. For this we need the following auxiliary definitions and lemmas which are used in the proofs of the main theorems.

The following lemmas state properties of the constraint satisfaction relation $\models$, subtyping, and the type intersection operation.

Lemma 2. *If we have* S *such that* $S \models C \cup C\prime$*, then* $S \models C$ *and* $S \models C\prime$*.*

Lemma 3. *If* $S \models Eq$ *such that* $(\Gamma, \Delta, Eq) = \otimes((\Gamma_1, \ldots, \Gamma_n), (\Delta_1, \ldots, \Delta_n))$*, and* $\forall i. \Gamma_i, S(\Delta_i) \vdash M_i : S(\tau_i)$ *then* $\forall i. \Gamma, S(\Delta) \vdash M_i : S(\tau_i)$*.*

Lemma 4. *If we know for all* $i = 1, \ldots, n$ *that* $\Gamma_i, S(\Delta_i) \vdash b_i : bool$ *and we know* $(\Gamma, \Delta) = \otimes((\Gamma_1, \ldots, \Gamma_n), (\Delta_1, \ldots, \Delta_n))$*, then* $\Gamma, S(\Delta) \vdash b_i : bool$*.*

Lemma 5. *Let* $\tau_1, \ldots, \tau_n, \tau$ *be types such that* $\forall i. \tau_i \sqsubseteq \tau$*. Then* $\tau_1 + \ldots + \tau_n \sqsubseteq \tau$*.*

Lemma 6. *Let* $\tau_1, \ldots, \tau_n, \tau$ *be types such that* $\exists i. \tau \sqsubseteq \tau_i$*. Then* $\tau \sqsubseteq \tau_1 + \ldots + \tau_n$*.*

Lemma 7. *If* $intersect(\tau_1, \tau_2, I, \Delta) = (\tau, Eq, \Delta\prime)$*, then* $\tau \sqsubseteq \tau_1$*, and* $\tau \sqsubseteq \tau_2$*.*

Proposition 1. *If* Eq *is a set of equality constraints in normal form, then* $Eq \models Eq$*.*

Now we have a theorem for the soundness of constraint generation which states that if one applies a substitution which satisfies the generated constraints to the type obtained by the constraint generation function, we get a well-typed program.

Theorem 2 (Soundness of Constraint Generation). *For a program, query, or term* P*, if* $generate(P) = (\tau, \Gamma, Eq, Ineq, \Delta)$*, then for any* $S \models Eq, Ineq$*, we have* $\Gamma, S(\Delta) \vdash P : S(\tau)$*.*

We also proved the soundness of constraint solving, which basically shows that the solved form returned by our constraint solver for a set of constrains C satisfies C.

Theorem 3 (Soundness of Constraint Solving). *Let Eq be a set of equality constraints, $Ineq$ a set of subtyping constraints, and Δ a set of type definitions. If $solve(Eq, Ineq, \Delta) \rightarrow^* (S, \Delta\prime)$ then $S \models Eq, Ineq$.*

Finally, using the last two theorems we prove the soundness of the type inference algorithm. The soundness theorem states that if one applies the substitution corresponding to the solved form returned by the solver to the type obtained by the constraint generation function, we get a well-typed program.

Theorem 4 (Soundness of Type Inference). *Given P, if $generate(P) = (\tau, \Gamma, Eq, Ineq, \Delta)$ and $solve(Eq, Ineq, \Delta) \rightarrow^* (S, \Delta\prime)$, then $\Gamma, S(\Delta\prime) \vdash P : S(\tau)$.*

6 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 [SBG08], 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 predicate [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, or data type declarations.

A different approach relied on ideas coming from functional programming 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, function symbols and predicate symbols used in a program. Key differences from our work are: 1) in previous works 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]. Although these approaches differ from ours since they follow the line of conservative approximations to the success set, we were inspired from general techniques from this area to define our type constraint solvers.

7 Final Remarks

In this paper, we present a sound type inference algorithm for pure Prolog. Inferred types are semantic approximations by default, but the user may tune the algorithm, quite easily, to automatically infer types which correspond to the usual 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. We proved the soundness of the algorithm, but completeness (meaning that the inferred types are a finite representation of *all* types which make the program well-typed) is an open problem for now. We strongly suspect that the algorithm is complete without closure, but could not prove it yet. On the implementation side we are now extending YAP^T to deal with full Prolog to be able to apply it to more elaborated programs. This includes built-ins and mutually recursive predicates. For this, we will have predefined rules for every built-in predicate and we are also extending the algorithm to generate constraints not for single predicates, but for each strongly connected component on the dependency graph of the program.

References

[BFC19] Barbosa, J., Florido, M., Costa, V.S.: 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, 20–25 September 2019. EPTCS, vol. 306, pp. 36–51 (2019)

[BFSC17] Barbosa, J., Florido, M., Costa, V.S.: Closed types for logic programming. In: 25th International Workshop on Functional and Logic Programming (WFLP 2017) (2017)

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

[BJ88] Bruynooghe, M., Janssens, G.: An instance of abstract interpretation integrating type and mode inferencing. In: 1988 Fifth International Conference and Symposium, Washington, pp. 669–683 (1988)

[CP98] Charatonik, W., Podelski, A.: Directional type inference for logic programs. In: Levi, G. (ed.) SAS 1998. LNCS, vol. 1503, pp. 278–294. Springer, Heidelberg (1998). https://doi.org/10.1007/3-540-49727-7_17

[Dam84] Damas, L.: Type assignment in programming languages. Ph.D. thesis, University of Edinburgh, UK (1984)

[DMP00] Drabent, W., Małuszyński, J., Pietrzak, P.: Locating type errors in untyped CLP programs. In: Deransart, P., Hermenegildo, M.V., Małuszynski, J. (eds.) Analysis and Visualization Tools for Constraint Programming. LNCS, vol. 1870, pp. 121–150. Springer, Heidelberg (2000). https://doi.org/10.1007/10722311_5

[DMP02] Drabent, W., Małuszyński, J., Pietrzak, P.: Using parametric set constraints for locating errors in CLP programs. Theory Pract. Logic Program. **2**(4–5), 549–610 (2002)

[DZ92] Dart, P.W., Zobel, J.: A regular type language for logic programs. In: Pfenning, F. (ed.) Types in Logic Programming, pp. 157–187. The MIT Press (1992)

[FD92] Florido, M., Damas, L.: Types as theories. In: Proceedings of post-conference workshop on Proofs and Types, Joint International Conference and Symposium on Logic Programming (1992)

[FSVY91] Frühwirth, T.W., Shapiro, E.Y., Vardi, M.Y., Yardeni, E.: Logic programs as types for logic programs. In: 1991 Proceedings of the Sixth Annual Symposium on Logic in Computer Science (LICS 1991), Netherlands, pp. 300–309 (1991)

[GdW94] Gallagher, J.P., de Waal, D.A.: Fast and precise regular approximations of logic programs. In: 1994 Logic Programming, International Conference on Logic Programming, Italy, pp. 599–613 (1994)

[Han89] Hanus, M.: Polymorphic high-order programming in prolog. In: Levi, G., Martelli, M. (eds.) Logic Programming, Proceedings of the Sixth International Conference, Lisbon, Portugal, 19–23 June 1989, pp. 382–397. MIT Press (1989)

[Han13] Hanus, M.: Functional logic programming: from theory to curry. In: Voronkov, A., Weidenbach, C. (eds.) Programming Logics. LNCS, vol. 7797, pp. 123–168. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37651-1_6

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

[HJ92] Heintze, N., Jaffar, J.: Semantic types for logic programs. In: Pfenning, F. (ed.) Types in Logic Programming, pp. 141–155. The MIT Press (1992)

[HL94] Hill, P.M., Lloyd, J.W.: The Gödel Programming Language. MIT Press, Cambridge (1994)

[LR91] Lakshman, T.L., Reddy, U.S.: 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)

[Lu01] Lunjin, L.: On Dart-Zobel algorithm for testing regular type inclusion. SIGPLAN Not. **36**(9), 81–85 (2001)

[Mah88] Maher, M.: Complete axiomatizations of the algebras of finite, rational and infinite trees. In: Proceedings Third Annual Symposium on Logic in Computer Science, Los Alamitos, CA, USA, pp. 348–357. IEEE Computer Society, July 1988

[Mis84] Mishra, P.: Towards a theory of types in Prolog. In: Proceedings of the 1984 International Symposium on Logic Programming, Atlantic City, New Jersey, USA, 6–9 February 1984, pp. 289–298. IEEE-CS (1984)

[MO84] Mycroft, A., O'Keefe, R.A.: A polymorphic type system for Prolog. Artif. Intell. **23**(3), 295–307 (1984)

[Nai92] Naish, L.: Types and the intended meaning of logic programs. In: Pfenning, F. (ed.) Types in Logic Programming, pp. 189–216. The MIT Press (1992)

[PCPH08] Pietrzak, P., Correas, J., Puebla, G., Hermenegildo, M.V.: 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, pp. 61–70, January 2008

[PR89] Pyo, C., Reddy, U.S.: Inference of polymorphic types for logic programs. In: 1989 Proceedings of the North American Conference on Logic Programming, USA, 2 Volumes, pp. 1115–1132 (1989)

[SBG08] Schrijvers, T., Bruynooghe, M., Gallagher, J.P.: From monomorphic to polymorphic well-typings and beyond. In: Hanus, M. (ed.) LOPSTR 2008. LNCS, vol. 5438, pp. 152–167. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00515-2_11

[SCWD08] Schrijvers, T., Santos Costa, V., Wielemaker, J., Demoen, B.: Towards typed Prolog. In: Garcia de la Banda, M., Pontelli, E. (eds.) ICLP 2008. LNCS, vol. 5366, pp. 693–697. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-89982-2_59

[SG95] Sağlam, H., Gallagher, J.P.: Approximating constraint logic programs using polymorphic types and regular descriptions. In: Hermenegildo, M., Swierstra, S.D. (eds.) Programming Languages: Implementations, Logics and Programs, pp. 461–462. Springer, Heidelberg (1995)

[SHC96] Somogyi, Z., Henderson, F., Conway, T.C.: The execution algorithm of Mercury, an efficient purely declarative logic programming language. J. Log. Program. **29**(1–3), 17–64 (1996)

[TTD97] Talbot, J.M., Tison, S., Devienne, P.: Set-based analysis for logic programming and tree automata. In: Van Hentenryck, P. (ed.) SAS 1997. LNCS, vol. 1302, pp. 127–140. Springer, Heidelberg (1997). https://doi.org/10.1007/BFb0032738

[Ull88] Ullman, J.D.: Principles of Database and Knowledge-Base Systems, Computer Science Press Inc. (1988)

[VB02] Vaucheret, C., Bueno, F.: More precise yet efficient type inference for logic programs. In: Hermenegildo, M.V., Puebla, G. (eds.) SAS 2002. LNCS, vol. 2477, pp. 102–116. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45789-5_10

[VR90] Van Roy, P.L.: Can logic programming execute as fast as imperative programming? Ph.D. thesis, EECS Department, University of California, Berkeley, November 1990

[YFS92] Yardeni, E., Frühwirth, T.W., Shapiro, E.: Polymorphically typed logic programs. In: Pfenning, F. (ed.) Types in Logic Programming, pp. 63–90. The MIT Press (1992)

[Zob87] Zobel, J.: Derivation of polymorphic types for Prolog programs. In: 1987 Proceedings of the Fourth International Conference on Logic Programming, Melbourne (1987)