An Optional Static Type System for Prolog

Isabel Wingen, Philipp Körner ⊠[0000−0001−7256−9560]

Institut für Informatik, Universität Düsseldorf
Universitätsstr. 1, D-40225 Düsseldorf, Germany
{isabel.wingen,p.koerner}@uni-duesseldorf.de

Abstract. Benefits of static type systems are well-known: typically, they offer guarantees that no type error will occur during runtime and, inherently, inferred types serve as documentation on how functions are called. On the other hand, many type systems have to limit expressiveness of the language because, in general, it is undecidable whether a given program is correct regarding types. Another concern that was not addressed so far is that, for logic programming languages such as Prolog, it is impossible to distinguish between intended and unintended failure and, worse, intended and unintended success without additional annotations.

In this paper, we elaborate on and discuss the aforementioned issues. As an alternative, we present a static type analysis which is based on *plspec*. Instead of ensuring full type-safety, we aim to statically identify type errors on a best-effort basis without limiting the expressiveness of Prolog programs. Finally, we evaluate our approach on real-world code featured in the SWI community packages and a large project implementing a model checker.

Keywords: Prolog, static verification, optional type system, data specification

1 Introduction

Dynamic type systems often enable type errors during development. Generally, this is not too much of an issue as errors usually get caught early by test cases or REPL-driven development. Prolog programs however do not follow patterns prevalent in other programming paradigms. Exception are thrown rarely and execution is resumed at some prior point via backtracking instead, before queries ultimately fail. This renders it cumbersome to identify type errors, their location and when they occur.

There has been broad research on type systems offering a guarantee about the absence of type errors. Yet, in dynamic programming languages such as Prolog, a complete well-typing of arbitrary programs is undecidable [14]. Thus, in order for the type system to work, the expressiveness of the language often is limited. This hinders adaptation to existing code severely, and, as a consequence, type errors are often ignored in larger projects.

This paper contributes the following:

- A type analysis tool that can be used for *any* Prolog program without modification. It can handle a proper “any” type and can easily be extended for any Prolog dialect.
- An empirical evaluation of the amount of inferred types during type analysis.
- Automatic inference and generation of pre- and postconditions.

2 A Note on Type Systems and Related Work

Static type systems have a huge success story, mostly in functional programming languages like Haskell [6], but also in some Prolog derivatives, such as Mercury [4]. Even similar dynamic languages such as Erlang include a type specification language [5]. Many static type systems for logic programming languages have been presented [13], including the seminal works of [12], which also influenced Typed Prolog [8], and a pluggable type system for Yap and SWI-Prolog [16].

All type systems have some common foundations, yet usually vary in expressiveness. Some type systems suggest type annotations for functions or predicates, some require annotations of all predicates or those of which the type cannot be inferred automatically to a satisfactory level. Yet, type checking of logic programs is, in general, undecidable [14]. This renders only three feasible ways to deal with typing:

1. Allow only a subset of types, for which typing is decidable, e.g., regular types [2] or even only mode annotations [15].
2. Require annotations where typing is not decidable without additional information.
3. Work on a best-effort basis which may let some type errors slip through.

Most type systems fall into the first or the second category. Yet, this usually limits how programs can be written: some efficient or idiomatic patterns may be rejected by the type system. As an example, most implementations of the Hindley-Milner type system [11] do not allow heterogeneous lists. Additionally, most type system refuse to handle a proper "any" type, where not enough information is available and arguments may, statically, be any arbitrary value. Such restrictions render adaptation of type systems to existing projects infeasible. Annotations, however, can be used to guide type systems and allow more precise typing. The trade-off is code overhead introduced by the annotations themselves, which are often cumbersome to write and to maintain.

Into the last category falls the work of Schrijvers et al. [16], and, more well-known, the seminal work of Ciao Prolog [3] featuring a rich assertion language which can be used to describe types. Unfortunately, [16] seems to be abandoned after an early publication and the official release was removed. Ciao's approach, on the other hand, is very powerful, yet is incompatible with other Prolog dialects.

We strongly agree with the reasoning and philosophy behind Ciao stated in [3]: type systems for languages such as Prolog must be optional in order retain the usefulness, power and expressiveness of the language. Mycroft-O'Keefe identified two typical mistakes that can be uncovered: firstly, omitted cases and, secondly, transposed arguments. We argue that omitted cases might as well be intended failure and, as such, should not be covered by a type system at all. Additionally, traditional type systems such as the seminal work of Mycroft-O'Keefe [12] often are not a good fit, as typing in Prolog is a curious case: due to backtracking and goal failure, type errors may lead to behaviour that is valid, yet unintended.

Backtracking Prolog predicates are allowed to offer multiple solutions which is often referred to as non-determinism. Once a goal fails, execution continues at the last choice point where another solution might be possible. Thus, if a predicate was called incorrectly, the program might still continue because another solution is found, e.g., based on other input. Consider an error in a specialised algorithm: if there is a choice point, a solution might still be found if another, slower, fall-back implementation is invoked via backtracking. Such errors could go unnoticed for a long time as they cannot be uncovered by testing if a correct solution is still found in a less efficient manner.

Goal Failure Most ISO Prolog predicates throw an exception if they are called with incorrect types. However, non-ISO predicates (such libraries as code written by programmers) usually fail as no solution is found because the input does not match with any clause. E.g., consider a predicate as trivial as `member`:

```
member(H, [H|_]).     member(E, [_|T]) :- member(E, T).
```

Querying `member(1, [2,3,4])` will fail because *the first argument is not in the list*, which is the second argument. We name this *intended failure*. Yet, if the second argument is not a list, e.g., when called as `member(1, 2)`, it will fail because *the second argument is not a list*. We call this *unintended failure*, as the predicate is called *incorrectly*. The story gets even worse: additionally to failure cases, there can also be unintended *success*. Calling `member(2, [1, 2|foo])` is not intended to succeed, as the second argument is not a list, yet the query returns successfully. Distinguishing between intended and unintended behaviour is impossible as they use the same signal, i.e. goal failure (or success). We argue that the only proper behaviour would be to raise an error on unintended input instead because this most likely is a programming error.

In this paper, we investigate the following questions: Can we implement an optional type system that supports *any Prolog dialect*? How well does such a type system perform and is a subset of errors that are identified on *best-effort basis* sufficient? We think that the most relevant class of errors is that an argument is passed incorrectly, i.e. the type is wrong. Thus, an important question is how precise type inference by such a type system could be. If it works well enough, popular error classes such as transposed arguments, as described by [12], can be identified in most cases.

For this, we build on top of *plspec* [7] which offers some type annotations that can be instrumented as runtime checks. Other libraries that provide annotations as presented, e.g. in [3,16], can be supported by including minor syntactic transformations as *plspec*'s feature set is very similar overall. In the following, the library and its annotations are presented, as they form the foundations for a static type analysis on top.

3 Foundation: *plspec*

plspec is an ad-hoc type system that executes type checks at runtime via co-routining. With *plspec*, it is possible to add two kinds of annotations. The first kind of annotation allows introduction of new types. *plspec* offers three different ways for this. For our type system, we currently focus only on the first one and implement shipped special cases that fall under the third category, i.e. tuples, lists and compound terms:

1. recombination of existing types
2. providing a predicate that acts as characteristic function
3. rules to check part of a term and generate new specifications for sub-terms

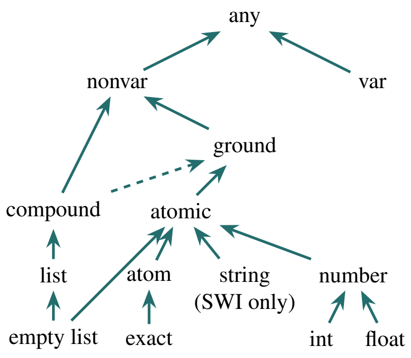

Fig. 1. Abstract Type Domain

plspec's built-in types are shown in Fig. 1. They correspond to Prolog types, with the addition of “exact”, which only allows a single specified atom (like a zero-arity compound), and “any”, which allows any value. Some types are polymorphic, e.g. lists can be instantiated to lists of a specific type. There are also two combinators, `one_of` that allows union types as well as `and`, which is the intersection of two types.

Combination of built-in types is certainly very expressive. While such structures cannot be inferred easily without prior definition, as a realistic example, it is possible to define a tree of integer values by using the `one_of` combinator as follows:

```
defspec(tree, one_of([int, compound(node(tree, int, tree))])).
```

Valid `trees` are `1`, `node(1, 2, 3)`, `node(node(0, 1, 2), 3, 4)` but not, e.g. `tree(1, 2, 3)`, where the functor does not match, or `node(a, b, c)` which stores atoms instead of integer values. Note that it is also possible to use a wildcard type to define a tree `tree(specvar(X))`, which passes the variable down into its nodes. `specvars` are a placeholder to express that two or more terms share a common, but arbitrary type. This can be used to define template-like data structures which can be instantiated as needed, e.g., as a `tree(int)`.

The second kind of annotations specifies how predicates may be called and, possibly, what parameters are return values. We re-use two different annotations for that:

1. *Preconditions* specify types for all arguments of a predicate. For a call to be valid, at least one precondition has to be satisfied.
2. *Postconditions* add promises for a predicate: if the predicate was called with certain types and if the call was successful, specified type information holds on exit.

Both pre- and postconditions must be valid for every clause of the specified predicate. Consider a variation of `member/2`, where the second argument *has to be* a list of atoms, and the first argument can either be an `atom` or `var`:

```
atom_member(H,[H|_]).   atom_member(E,[_|T]) :- atom_member(E,T).
```

Instead of checking the terms in the predicate, type constraints describing intended input are added via *plspec*'s pre- and postconditions. The following preconditions express the valid types one has to provide: the first argument is either a variable or an atom, and the second argument must be a list of atoms.

```
:- spec_pre(atom_member/2, [var, list(atom)]).
:- spec_pre(atom_member/2, [atom, list(atom)]).
```

As the second argument is always a ground list of atoms, we can assure callers of `atom_member/2`, that the first term is bound after the execution using a postcondition:

```
:- spec_post(atom_member/2, [var, list(atom)], [atom, list(atom)]).
```

Postconditions for a predicate are defined using two argument lists: they are read as an implication. For `atom_member/2` above, this means that "if the first argument is a variable and the second argument is a list of atoms, and if `atom_member/2` succeeds, it is guaranteed that the second argument is still a list of atom, but also that the first argument will be bound to an atom". If the premise of the postcondition does not hold or the predicate fails, no information is gained.

Extensions to plspec The traditional understanding if there are two instances of the same type variable, e.g. in a call such as `spec_pre(identity/2, [X, X])`, is that both arguments *share all types*. Yet, we want to improve on the expressiveness of, say, `spec_pre(member/2, [X, list(X)])`, and allow heterogeneous lists. This extension is not yet implemented in *plspec* itself and is only part of the static analysis in *plstatic*. In order to express how the type of type variables is defined, we use `compatible` for the homogeneous and `union` for the heterogeneous case.

If a list is assigned the type `list(compatible(X))`, every item in the list is assigned the type `compatible(X)`. Now *plstatic* checks whether all these terms share all types, thus enforcing a homogeneous list. Analogously, if a list is assigned the type `list(union(X))`, every item in the list is assigned the type `union(X)`. But instead of intersecting, *plstatic* collects the types of these terms and builds a union type.

To give an example for the semantics of compatible and union, the list `[1, a]` has the *inner* type `one_of([int, atom])` under the semantics of a union, and results in a type error (as the intersection of int and atom is empty) if its elements should be compatible. A correct annotation for `member/2` would be the following postcondition: `spec_post(member/2,[any,list(any)],[compatible(X),list(union(X))])`, i.e., the list is heterogeneous, and the type of the first argument must occur in this list.

4 Our Type System

In the following, we describe a prototype named *plstatic*. It uses an abstract interpreter in order to collect type information on Prolog programs and additionally to identify type errors on a best-effort basis, without additional annotations. The tool is available at `https://github.com/isabelwingen/prolog-analyzer`.

Purpose and Result The tool *plstatic* performs a type analysis on the provided code. All inferred information can be written out in form of annotations in *plspec* syntax, or HTML data that may serve, e.g., as documentation. Naturally, *plstatic* shows an overview of type errors, which were found during the analysis.

As typing can be seen as a special case of abstract interpretation [1], we use *plspec*'s annotations to derive an abstract value, i.e. a type, for terms in a Prolog clause. Abstract types correspond to the types shown in Fig. 1, where a type has an edge pointing to a strict supertype. However, as distinguishing ground from nonvar terms often is important, compound terms are tried to be abstracted to the ground type first, represented by the dashed edge. We use the least upper bound and greatest lower bound operations as they are induced by the type subset relation. This analysis is done statically and without concrete interpretation of Prolog code, based on *plspec* annotations and term literals.

A Note on Annotations *plstatic* works without additional annotations in the analysed code. We are able to derive type information from (a subset of) built-in (ISO) predicates. For those predicates, we provided pre- and postconditions, which are also processed by the term expander. We also annotated a few popular libraries, e.g. the lists library.

For predicates lacking annotations, types can be derived if type information exists for predicates called in their body or can be inferred from unification with term structure in the code. Derived types describe intended success for the unannotated predicate. Naturally, precision of the type analysis improves with more annotations.

4.1 Tool Architecture

plstatic is implemented in Clojure, a Lisp dialect running on the JVM. It might as well have been implemented as a meta-interpreter in Prolog, but Clojure allows easier integration into text editors, IDEs and potentially also web services. However, this requires to extract a representation of the Prolog program. We decided against parsing Prolog as operator definitions are hard to add during runtime and it is guaranteed to lose transformations done by term expanders[1]. Instead, we add a term expander ourselves before we load the program. It implements *plspec*'s syntax for annotations and extracts those alongside the program itself. All gathered information is written into a separate file in edn syntax[2], which can easily be read back in Clojure.

plstatic consists of two parts: an executable jar containing the tool for static analysis and the aforementioned Prolog term expander. The architecture of *plstatic* is pictured in Fig. 2. The analysis core is started with parameters specifying the path to a Prolog source file or directory and a Prolog dialect (for now, “swipl” or “sicstus”). Additionally, the path to the term expander can be passed as an argument as well, if another syntax for annotations than *plspec*'s is desired.

Special care has to be taken when an entire directory is analysed: when modules are included, it is often not obvious where a predicate is located. In particular, it can be hard to decide whether a predicate is user-defined, shipped as part of a library or part of the built-in predicates available in the user namespace. Thus, when the edn-file is imported, a data structure is kept in order to resolve calls correctly.

As our evaluation in Section 5 uses untrusted third-party code, we take care that the Prolog code, that may immediately run when loaded, is not executed. Instead, the term expander does not return any clause, effectively removing the entire program during compilation. Trusted term expanders can be loaded beforehand if required.

4.2 Analysis

Our approach to type inference implements a classical abstract interpreter. In the first phase, every clause is analysed individually. We use *plspec*'s annotations of the clause and the sub-goals to derive an abstract type domain for all terms in the clause.

In the second phase, we broaden the analysis to a global scope: After the first phase, we have obtained a typing for every clause, which describes the types that the terms

---

[1] Term expansion is a mechanism that allows source-to-source transformation.

[2] https://github.com/edn-format/edn

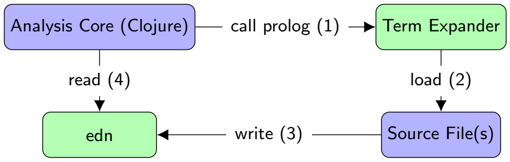

Fig. 2. Tool Architecture

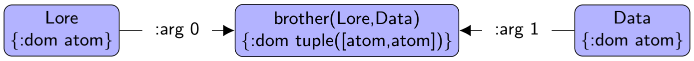

Fig. 3. An Example Environment (Using edn-Formatted Maps)

have after a successful execution of the clause. The inferred type information for all clauses of a predicate, can be stored as a postcondition. Perhaps this postcondition is more accurate than the already provided one. In this case, the analysis of a predicate p would in turn improve the analysis result for clauses that call p.

So, we create new postconditions for every predicate. Afterwards, we repeat the first and the second phase, until, eventually, a fixpoint is reached. As the type domains are finite, the types of the single terms can only reach a certain degree of accuracy. Therefore, a fixpoint will be reached eventually. Then, no further information can be obtained and the type analysis terminates.

Example: Rate My Ship The following code will accompany us during this section.

```
ship(Ship) :- member(Ship, [destiny, galactica, enterprise]).
rating(stars(Rate)) :- member(Rate, [1,2,3,4,5]).
rate_my_ship(S,R) :- ship(S), rating(R).
```

Preparation Before the analysis, we prepare the `edn` produced by the term expander. To determine the module of a sub-goal, a mapping of modules to predicates as well as imports is kept in order find the correct module of each predicate call. For every loaded predicate, we check, if there are pre- and postconditions specified. Otherwise, they are created containing any-types during the preparation as follows:

The construction uses the literals in the clause heads. Lists, compounds and the different atomic terms are recognised as such. For variable terms, we assume the type `any`. Consider the clause head `check(nil, foo(X,Y))`, for which *plstatic* creates `[one_of([var,atom(nil)]), one_of([var, compound(foo(any, any))])]` as a precondition, and `[any,any] => [atom(nil),compound(foo(any, any))])` as a postcondition. It is important that the created pre- and postconditions are general enough to be valid for every clause.

Below, we show the generated specs for our example after the preparation step:

```
:- spec_pre(ship/1, [any]).
:- spec_post(ship/1, [any], [any]).
:- spec_pre(rating/1, [one_of([var, compound([stars(any)])])]).
:- spec_post(rating/1, [any], [compound([stars(any)])]).
:- spec_pre(rate_my_ship/2, [any, any]).
:- spec_post(rate_my_ship/2, [any, any], [any, any]).
```

Phase 1: Clause-Local Analysis Because of the logic nature of prolog, it is not sufficient to store the current type for a variable at a given point, as relationships between terms caused, e.g., by unification, have to be considered as well.

Contrary to traditional approaches, we use an environment in form of a directed graph to store relationships between variables per clause. Every term that occurs in the currently considered clause is represented as a node in the environment. The inferred types of the terms are saved as attributes in the corresponding nodes. Relationships between terms and sub-terms (e.g. `[H|T]`) or postconditions are saved as labelled edges between the term nodes. An example is given in Fig. 3, where the structure of a compound term `brother(Lore, Data)` is shown.

During the analysis of a clause, the type domains of the terms are updated and their precision is improved. For this, annotations of called predicates and the relationships between the terms are considered. When new type information about a term is gained, the greatest lower bound is calculated by intersecting both domains. When considering variables in Prolog however, this comes with some pitfalls that are discussed in more details in *Step 2*. If the type intersection is empty, no concrete value is possible for the Prolog term and a type error is reported. However, we have to assume that all given annotations are correct.

Step 1: Clause Head The environment is initialised with all terms occurring in the head of the clause. Information about the head of the clause can be derived from the preconditions. According to *plspec*, at least one precondition must be fulfilled.

Consider the following example: the predicate `cake(X, Y)` is annotated with the preconditions `[atom, int]` and `[int, atom]`. This means that `cake/2` expects an atom and an integer, no matter the order. We might derive `one_of([atom, int])` as type for both `X` and `Y`. Then, `X=1,Y=2` would be valid input, as both fulfil their individual type constraint, but together, they violate the original precondition. To prevent such errors, we create an artificial tuple containing all arguments, whose domain is a union-type containing all supplied preconditions. This artificial term functions as a “watcher”, and ensures all type constraints. For the `cake` predicate, the term `[X,Y]` is added to the environment, with `one_of([tuple([atom,int]), tuple([int,atom])])` as its type. Once we know a more specific type for, e.g., `Y`, we can derive which of the two options must be valid for the “watcher”, and therefore, we can derive a type for `X`. The environment is pictured in Fig. 4.

Due to page limitations, we only present the environment of `rating/1`.

```
[Rate]   domain: tuple([compound(stars([any]))]).
Rate     domain: compound(stars([any])).
```

A Note on Compound and List Terms Whenever a type is added to the domain of a compound or list term, types for the children of this term are automatically derived and added to their domain, if possible.

Step 2: Evaluate Body We analyse the body step by step, making use of (generated or annotated) pre- and postconditions of all sub-goals one after another using abstract

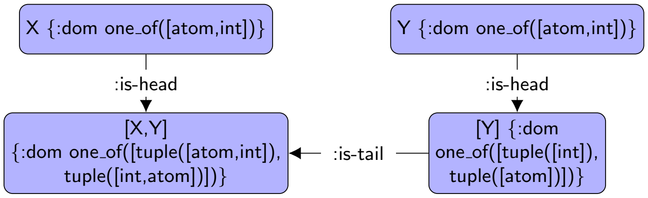

Fig. 4. Environment with a Watcher (Using edn-Formatted Maps)

Table 1. Environment for `rate_my_ship/2`

Variable Term	Clause Head	after 1st sub-goal	after 2nd sub-goal
[S, R]	tuple([any, any])	tuple([any, any])	tuple([any, any])
[R]	tuple([any])	tuple([any])	tuple([any])
R	any	any	compound(star([any]))
S	any	any	any

types as values. On the first occurrence of a term, it is added to the environment. Similarly, to the clause head, at least one precondition of the sub-goal must be compatible with the combination of the arguments it is called with.

The analysis does not step into the sub-goal, and only uses pre- and postconditions. A postcondition specifies type constraints on a term after the called predicate succeeds. Thus, it is checked which premises of postconditions are fulfilled. Then, the greatest lower bound of the current type domain and the possible conclusion of the postconditions is calculated in order to improve precision. An example is shown in Table 1.

A Note on Type Variables We have introduced two new kinds of type variables (cf. Section 3): `union` and `compatible`. It is possible to use `union(X)` or `compatible(X)`, where `X` is a type variable. Both are placeholders for yet unknown types and express two different relationships between terms:

Every term that is assigned the type `union(X)` contributes to the definition of the type that is `X`. The connection is made by adding a labelled edge `:union` between the term and `X`. Then, the domains of the terms are filled as described. At the end of the analysis step, the union type is inferred via the least upper bound of all connected terms.

On the other hand, terms that are assigned the type `compatible(X)`, must be compatible with all other occurrences of this type, and the inferred union type of `X`, implying that their intersection is not empty. As with the `union` type, we create a labelled edge `:compatible` connecting the term to `X`. These edges are processed *after* all union edges have been visited. We create the intersection of all involved terms, and assign it to the domain of all terms connected via a `:compatible` edge.

In order to determine the type for a union type variable, it is required to know all contributing terms. If a term is known to be a compound or a list of a known size, the assigned type is passed down to its sub-terms using the mechanisms described above. Yet, consider a term that is a Prolog variable, e.g. `T`, with the inferred type `list(union(X))`. Then, the length of the list and its possible elements are unknown, as the Prolog variable might be bound at a later point.This requires an additional step in order to ensure that the domain for the type variable `X` is compiled correctly: we opted to add a `:has-type` edge to the environment, which connects a Prolog variable, e.g `T`, to an artificially created variable `T__<uuid>` storing the inner type, i.e. `union(X)` in the example above. Whenever the domain of the variable is updated, so is its list type. The artificial list type variable then is connected with `union(X)`.

For `compound` and `tuple` *type specifications*, an artificial term is created and linked to the variable term via an edge labelled `:artificial`. This is required to mimic unification of Prolog variables. Whenever the domain of the variable term is updated, the artificial term's domain is updated as well. Finally, the information is propagated into the corresponding sub-terms if required.

Have a look at `member/2` used in the body of `ship/1`. The provided postcondition is `post_spec(member/2, [any, any], [compatible(X), list(union(X))])`. Therefore, after analysing the body of `ship/1`, we obtain the following environment:

```
galactica      atom            galactica  ----union---> X
destiny        atom            destiny    ----union---> X
enterprise     atom            enterprise ----union---> X
Ship           atom            Ship       -compatible-> X
```

Step 3: Term Relationships After analysing the body, all terms in the clause are included in the environment. Then, nodes that may be destructured, i.e. lists and compound terms, are looked up in the graph. As sub-terms, e.g. `X` in `a(X)`, can be used individually, i.e. without their enclosing compound term, in subsequent sub-goals, inferred information has to be propagated back to the larger compound term. We introduce the following edges in order to provide the necessary mechanism:

For lists, we extract the head and tail terms and add them to the environment, if they are not already contained. Those terms are marked with special edges `:is-tail` and `:is-head` (cf. Fig. 4) pointing to the original list. For compounds, we add the argument terms to the environment and store the position of every term in the compound by adding an edge `:pos` (cf. Fig. 3).

The following edges are added to the environment for `rate_my_ship/2`:
`R --is-head--> [R]`,`S --is-head--> [S, R]` and `[R] --is-tail--> [S, R]`

A Note on Prolog Variables The any-type can be split into two disjoint sets: variables and non-variable terms. Non-variable terms may only be inferred to be more concrete in every step unless a type error is determined. Yet, Prolog variables have the unique property that their type can change, as they can be bound to, say, an atom, which is not a sub-type. To take this into account, a different intersection mechanism is required for variables:

- Preconditions of the *currently analysed* predicate may render a variable non-variable.
- Preconditions of a *called sub-goal* cannot render a variable term non-variable.
- Postconditions of a *called sub-goal* may render a variable term non-variable.
- Once a Prolog variable is bound to a non-variable, it behaves like any non-variable.

Step 4: Fixed-Point Algorithm During the prior steps, we have added some edges. These are now used to update the domains of the linked terms using the knowledge expressed by the edges. If the environment no longer changes, we have consumed all collected knowledge and have found a preliminary result.

`R` links back to the list it is part of in the environment of `rate_my_ship/2`. We can therefore update those terms containing `R`, as we have found a more precise type for `R`.

```
[S, R]  tuple([any, compound(star([any]))]).
[R]     tuple([compound(star([any]))])   R   -head-> [R]
R       compound(star([any]))            S   -head-> [S, R]
S       any                              [R] -tail-> [S, R]
```

Phase 2: Global Propagation of Type Information During the local analysis, each clause was inspected in isolation. The type domains in the returned environments contain the types after a successful execution of a clause *with the initial knowledge*. The gathered information can be propagated to the caller of the corresponding predicate in order to improve the precision of the type inference.

Each environment can be used to generate a conclusion of a postcondition. If a predicate succeeds, at least one of its clauses succeeded. As a postcondition must be valid for the entire predicate, the conclusion of a new postcondition is the union of the all gained conclusions of the corresponding clauses. This newly gained knowledge (in form of a postcondition) is added to the analysed data for every predicate. Afterwards, both local analysis and global propagation are triggered, until a fixed point is reached. Inferred pre- and postconditions can be written out after analysis in *plspec*'s syntax.

Example: append/2 Consider the append program:

```
append([], Y, Y).    append([H|T], Y, [H|R]) :- append(T, Y, R).
```

For the first clause, *plstatic* would derive the types `[list(any), any, any]`. For the second clause, we gain no additional information from the body, because `append/2` is calling itself, so we derive the types `[list(any), any, list(any)]`. To create a conclusion of a postcondition for the predicate, we need to combine the results of the two clauses. Unfortunately, as the type of the third argument is `any` in one case, it swallows the more precise type `list(any)`. We obtain the following conclusion: `[list(any), any, any]`. While the *intention* is that the second and third arguments are lists as well, this cannot be inferred without annotations.

As you have probably noticed, *plstatic* has not yet found the accurate type `atom` for `S` or `R` in `rate_my_ship/2`. This is because the pre- and postconditions of `ship/1` have not been updated yet, so *plstatic* has no way of knowing that `S` is an atom. In the first phase, we have concluded that the argument given to `ship/1` must be of type `atom` after a successful execution. As `ship/1` has only one clause, we can infer the postcondition: `:- post_spec(ship/1, [any], [atom]).` Analogously, we obtain `:- post_spec(rating/1, [any], [compound(stars([atom]))]).`

The propagation of the newly gained knowledge is shown in Table 2. Afterwards we can update the pre- and postconditions for `rate_my_ship/2`, but `ship/1` and `rating/1` are not affected from this. If our program has no more clauses, the fixpoint is reached, and the analysis stops.

Table 2. Environment for `rate_my_ship/2`

Variable Term	Newly Gained Knowledge	After Propagation
[S, R]	tuple([any, compound(star([any]))])	tuple([*atom*, compound(star([*atom*]))])
[R]	tuple([compound(star([any]))])	tuple([compound(star([*atom*]))])
R	*compound(star([atom]))*	compound(star([atom]))
S	*atom*	atom

A Note on Backtracking Preconditions specify a condition which must be fulfilled at the moment of the call, and postconditions can provide information about the type of the used terms after a successful execution. The caller of a predicate is unaware which clause provided the result. Thus, the union of all gained type information has to be considered in the second phase. As a result, it is safe to ignore backtracking: yet, precision could in some cases be improved if clause ordering and cuts (`!`) were considered.

5 Evaluation

To our knowledge, papers on type systems for Prolog usually omit an evaluation of their applicability for existing, real-world Prolog code and offer insights on their type inference mechanisms on small toy examples, such as the well-known `append` predicate. However, we want to consider code that is more involved than homework assignments. There is no indication to what extent type inference approaches are applicable to the real world, or how much work has to be spent re-writing code for full-fledged type systems.

In contrast, we baptise *plstatic* by fire and evaluate for how many variables in the code we can infer a type that is more precise than `any`. For this, we use smaller SWI community packages[3], as well as ProB [9], a model checker and constraint solver that currently consists of more than 120 000 lines of Prolog code.

5.1 Known Limitations

Currently, we face three limitations in *plstatic*: firstly, as we try to avoid widening whenever possible, i.e., we try to use the most precise type like a `one_of` instead of generalising to their common supertype, performance is not too good. Analysis of small projects runs neglectably fast, yet ProB requires several hours to complete a full analysis. Secondly, libraries throw a wrench into our scheme: modern Prolog systems pre-compile the code. Hence, meta-programs, such as term expanders, cannot access their clauses. Thus, library code is not considered and *plstatic* has to rely on annotations. Currently, we only provide annotations for large parts of the lists library (for both *SWI Prolog* and *SICStus Prolog*) and the AVL tree library (for *SICStus Prolog* only). Otherwise, for all library predicates that are not annotated, an `any` type has to be assumed. Thirdly, we currently do not consider disjunctions and if-then-else constructs, but may gain additional precision once this is implemented.

---

[3] `http://www.swi-prolog.org/pack/list`

Table 3. Amount of Inferred Types for Variables

Repository	# Variables	Inferred Types	Unknown Calls
bddem	196	31.63 %	57.6 %
dia	400	68.5 %	8.23 %
maybe	32	6.25 %	70.0 %
plsmf	67	37.31 %	37.5 %
quickcheck	122	42.6 %	34.1 %
thousands	19	94.73 %	0.0 %
∅ SWI Community Packages	68344	21.8 %	39.0 %
ProB	81893	21.2 %	20.8 %

Additionally, there is an inherent limitation in our analysis strategy: some predicates may really work on *any* type, e.g. term type checking predicates (such as `ground/1` or `nonvar/1`) or the `member/2` predicate regarding the first argument. As no similar analysis for Prolog programs exists yet and type inference by hand is infeasible for large programs, it is certainly hard to gauge the precision of our type inference.

5.2 Empirical Evaluation

In Table 3, the results of some repositories[4] and the mean value of the 198 smallest community packages is shown. We give the amount of Prolog variables, and the percentage of which we can infer a type that is a strict sub-type of `any`. For reference, we also give the amount of calls to unknown predicates in order to give an idea how many missing types are caused by, e.g., library predicates lacking annotations. Though, once a variable is assigned an `any` type, the missing precision typically is passed on to terms that are interacting with the `any` term as the predicate is implemented in a library.

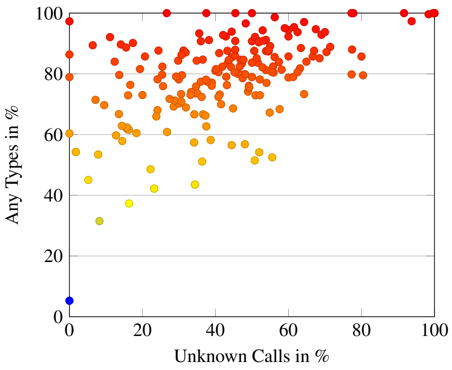

Fig. 5. Correlation Between Unknown Calls and Inferred Types

At first glance, the fraction of inferred types seems to be rather low. For some repositories, such as “dia” and “thousands”, a specific type could be inferred for a large percentage of variables. Note that in return, the amount of unknown calls is relatively low. Then, there are repositories such as “bddem” and “plsmf”, which both are wrappers of a C library. As such, the interop predicates are unknown and the inferred types are significantly lower. Finally, there are packages like “maybe”, “quickcheck” and projects such as ProB, that make use of other libraries, conditional compilation, meta-calls and other features that decrease accuracy of type inference.

---

[4] Full results: `https://github.com/pkoerner/plstatic-results/tree/lopstr19`

Overall, we were surprised how small the amount of inferred types was. Though, one has to consider that a large amount of predicates are library calls, e.g. into the popular CLP and CHR libraries. In Fig. 5, we show this relation. One can clearly recognise that (unknown) library calls negatively impact the results of our type analysis. Yet, many auxiliary predicates are written to be polymorphic and deal with any type.

plstatic was able to find several errors: many SWI libraries have been broken with changes introduced in *SWI Prolog* 7 [18]. Strings now are proper strings, where legacy code relies on the assumption that they are represented as code lists. Furthermore, *plstatic* located calls in ProB that were guaranteed to fail every time due to type errors. These calls decide whether a backend is usable in order to solve a given predicate and always fail. Thus, the errors have gone unnoticed for eight years, as the backend simply was not used. One error was reported due to missing term expansion as we did not execute untrusted Prolog code. We found another false-positive due to meta predicate annotations which add the module to a goal, thus altering the term structure. Additionally, we found some extensions *SICStus Prolog* made to the ISO standard that we were not aware of: e.g., arithmetic expressions allow expressions such as `X is integer(3.14)` or `log(2, 42)` that are not part of ISO Prolog but valid in *SICStus Prolog* were reported as errors in our type describing arithmetic expressions.

6 Conclusion and Future Work

In this paper, we presented *plstatic*, a tool that re-uses its annotations in order to verify types statically where possible. *plstatic* was able to locate type errors in several existing Prolog repositories. Yet, without annotations of further libraries, the amount of actual inferred types remains relatively low. We invite the Prolog community to discuss whether such type annotations are desired and should be shipped as part of packages.

There remains some work in *plstatic*: performance bottlenecks need to be reviewed to reduce the time required for analysis. Furthermore, the analysis would heavily benefit from a mechanism for the term expander to hook into library packages or manual annotations. It might also be possible to analyse some pre-compiled library beforehand and re-use those results in the analysis of the main program. We also plan to implement semantics for new types, for which the structure is not specified, but they may only be created by libraries. E.g., Prolog streams are impossible to create one without calling the corresponding predicates. Other examples include ordered sets or AVL trees, where it is possible to create or manipulate such a term, but it is heavily discouraged as it is very easy to introduce subtle errors.

Moreover, it would be exciting to compare the amount of inferred types to similar implementations such as CiaoPP. We assume their analysis to be stronger, but suspect that Ciao's approach might not scale as well for larger programs. Yet, comparison might be hindered, again, because features of other Prolog systems are not supported.

In [17] and also in the evaluation of *plspec* [7], it was determined that the overhead of run-time type checks can be enormous, especially if applied to recursive predicates. With additional type information, a large amount of run-time checks can be eliminated, as, e.g., proposed by [17]. It is fairly straightforward to generate a list of already discharged annotations and use that as a blacklist in *plspec*.

It is well-known that compilers often benefit heavily from type information. An interesting research question is to investigate the impact of type information, e.g. gained by *plstatic* or by annotations, when added to the binding-time analysis of a partial evaluator, such as LOGEN [10]. This might greatly reduce the work required of manually improving generated annotations in order to gain additional performance.

As a more pragmatic approach to future work, it would be greatly appreciated if the state-of-the-art of Prolog development tooling could be improved. Currently, IDEs and editor integrations are lacking. Including type information would be a great start.

References

1. P. Cousot. Types as abstract interpretations. In *Proceedings POPL*, pages 316–331. ACM, 1997.
2. J. P. Gallagher and K. S. Henriksen. Abstract domains based on regular types. In *Proceedings ICLP*, pages 27–42. Springer, 2004.
3. M. V. Hermenegildo, F. Bueno, M. Carro, P. López-García, E. Mera, J. F. Morales, and G. Puebla. An overview of Ciao and its design philosophy. *TPLP*, 12(1-2):219–252, 2012.
4. D. Jeffery. *Expressive Type Systems for Logic Programming Languages*. PhD thesis, Department of Computer Science and Software Engineering, The University of Melbourne, 2002.
5. M. Jimenez, T. Lindahl, and K. Sagonas. A Language for Specifying Type Contracts in Erlang and Its Interaction with Success Typings. In *Proceedings ERLANG*, pages 11–17. ACM, 2007.
6. S. P. Jones. *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003.
7. P. Körner and S. Krings. plspec – A Specification Language for Prolog Data. In *Proceedings WFLP*, volume 10997 of *LNAI*, pages 198–213. Springer, 2017.
8. T. Lakshman and U. S. Reddy. Typed Prolog: A Semantic Reconstruction of the Mycroft-O'Keefe Type System. In *ISLP*, volume 91, pages 202–217, 1991.
9. M. Leuschel and M. J. Butler. Prob: A model checker for B. In *Proceedings FME*, volume 2805 of *LNCS*, pages 855–874. Springer, 2003.
10. M. Leuschel, S. J. Craig, M. Bruynooghe, and W. Vanhoof. Specialising interpreters using offline partial deduction. In *Program Development in Computational Logic*, pages 340–375. Springer, 2004.
11. R. Milner. A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3):348–375, 1978.
12. A. Mycroft and R. A. O'Keefe. A polymorphic type system for Prolog. *Artificial intelligence*, 23(3):295–307, 1984.
13. F. Pfenning. *Types in logic programming*. MIT Press Cambridge, Massachusetts, USA, 1992.
14. F. Pfenning. On the undecidability of partial polymorphic type reconstruction. *Fundam. Inform.*, 19(1/2):185–199, 1993.
15. E. Rohwedder and F. Pfenning. Mode and termination checking for higher-order logic programs. In *European Symposium on Programming*, pages 296–310. Springer, 1996.
16. T. Schrijvers, V. S. Costa, J. Wielemaker, and B. Demoen. Towards typed Prolog. In *Proceedings ICLP*, volume 5366 of *LNCS*, pages 693–697. Springer, 2008.
17. N. Stulova, J. F. Morales, and M. V. Hermenegildo. Reducing the overhead of assertion run-time checks via static analysis. In *Proceedings PPDP*, pages 90–103. ACM, 2016.
18. J. Wielemaker. SWI-Prolog version 7 extensions. In *Proceedings CICLOPS-WLPE*, page 109, 2014.