Effectiveness of Annotation-Based Static Type Inference

Isabel Wingen, Philipp Körner ✉

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: 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. Exceptions are thrown rarely and execution is resumed at some prior point via backtracking instead, before queries ultimately fail (or succeed due to the wrong reason). 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 (briefly discussed in Section 2). 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.

At DECLARE'17, we presented *plspec* [7], a type system that uses annotations in order to insert run-time type checks (cf. Section 3). During discussions, the point was raised that some type checks could be made statically even with optional types. This paper thus contributes the following:

– A type analysis tool usable for *any* unmodified Prolog program. It handles a proper "any" type and is extensible for any Prolog dialect (Section 4).
– An empirical evaluation of the amount of inferred types using this tool (Section 5).
– Automatic inference and generation of pre- and postconditions of *plspec*.

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], which uses type and mode information in order to achieve major performance boosts. 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 Mycroft and O'Keefe [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, though always results in a well-typing of the program. Additionally, most type systems 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, but suffers due to incompatibilities with other Prolog dialects.

We share the reasoning and philosophy behind Ciao stated in [3]: type systems for languages such as Prolog must be optional in order retain usefulness, power and expressiveness of the language, even if it comes at the cost that not all type errors can be detected. Mycroft-O'Keefe identified two typical mistakes type systems uncover: 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

81 at all. Traditional type systems such as the seminal work of Mycroft-O'Keefe [12] often
82 are not a good fit, as typing in Prolog is a curious case: due to backtracking and goal
83 failure, type errors may lead to behaviour that is valid, yet unintended.

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

92 *Goal Failure.* Most ISO Prolog predicates raise an error if they are called with incorrect
93 types. However, non-ISO predicates usually fail as no solution is found because the
94 input does not match with any clause. E.g., consider a predicate as trivial as `member`:

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

96 Querying `member(1, [2,3,4])` will fail because *the first argument is not in the list*,
97 which is the second argument. We name this *intended failure*. Yet, if the second argu-
98 ment is not a list, e.g., when called as `member(1, 2)`, it will fail because *the second*
99 *argument is not a list*. We call this *unintended failure*, as the predicate is called *in-*
100 *correctly*. The story gets even worse: additionally to failure cases, there can also be
101 unintended *success*. Calling `member(2, [1, 2|foo])` is not intended to succeed, as
102 the second argument is not a list, yet the query returns successfully. Distinguishing be-
103 tween intended and unintended behaviour is impossible as they use the same signal, i.e.
104 goal failure (or success). We argue that the only proper behaviour would be to raise an
105 error on unintended input instead because this most likely is a programming error.
106 In this paper, we investigate the following questions: Can we implement an optional
107 type system that supports *any Prolog dialect*? How well does such a type system per-
108 form and is a subset of errors that are identified on *best-effort basis* sufficient? We think
109 that the most relevant class of errors is that an argument is passed incorrectly, i.e. the
110 type is wrong. Thus, an important question is how precise type inference by such a
111 type system could be. If it works well enough, popular error classes such as transposed
112 arguments, as described by [12], can be identified in most cases.

113 3 Foundation: *plspec*

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

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

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.

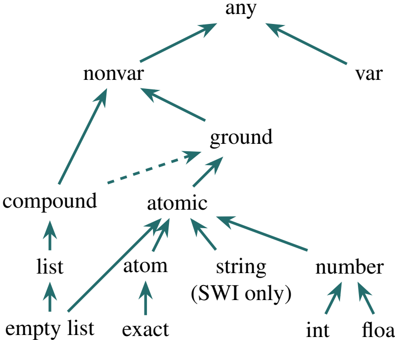

Fig. 1. Abstract Type Domain

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:

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

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

170 *Extensions to plspec.* The traditional understanding if there are two instances of the
171 same type variable, e.g. in a call such as `spec_pre(identity/2, [X, X])`, is that
172 both arguments *share all types*. Yet, we want to improve on the expressiveness of,
173 say, `spec_pre(member/2, [X, list(X)])`, and allow heterogeneous lists. This ex-
174 tension is not yet implemented in *plspec* itself and is only part of the static analy-
175 sis in *plstatic*. In order to express how the type of type variables is defined, we use
176 `compatible` for the homogeneous and `union` for the heterogeneous case.
177 If a list is assigned the type `list(compatible(X))`, every item in the list is as-
178 signed the type `compatible(X)`. Now *plstatic* checks whether all these terms share all
179 types, thus enforcing a homogeneous list. If a list is assigned the type `list(union(X))`,
180 every item in the list is assigned the type `union(X)`. But instead of a type intersection,
181 *plstatic* collects the types of these terms and builds a union type.
182 To give an example for the semantics of compatible and union, the list `[1, a]` has
183 the *inner* type `one_of([int, atom])` under the semantics of a union, and results in a
184 type error (as the intersection of int and atom is empty) if its elements should be com-
185 patible. A correct annotation for `member/2` would be the following postcondition:
186 `spec_post(member/2,[any,list(any)],[compatible(X),list(union(X))])`,
187 i.e., the list is heterogeneous, and the type of the first argument must occur in this list.

188 4 Our Type System

189 In the following, we describe a prototype named *plstatic*. It uses an abstract interpreter
190 in order to collect type information on Prolog programs and additionally to identify
191 type errors on a best-effort (i.e., based on available type information due to annota-
192 tions) basis, without additional annotations. The tool is available at https://github.com/
193 isabelwingen/prolog-analyzer. Due to page limitation, we can only present some points
194 we deem important.

195 *Purpose and Result.* The tool *plstatic* performs a type analysis on the provided code.
196 All inferred information can be written out in form of annotations in *plspec* syntax,
197 or HTML data that may serve, e.g., as documentation. Naturally, *plstatic* shows an
198 overview of type errors, which were found during the analysis. *plstatic* is not intended
199 to uncover all possible type errors. Instead, we are willing to trade some false negatives
200 for the absence of false positives, as they might overwhelm a developer in pure quantity.
201 Whether true programming errors can be discovered is discussed in Section 5.
202 As typing can be seen as a special case of abstract interpretation [1], we use *plspec*'s
203 annotations to derive an abstract value, i.e. a type, for terms in a Prolog clause. Abstract

204 types correspond to the types shown in Fig. 1, where a type has an edge pointing to a
205 strict supertype. However, as distinguishing ground from nonvar terms often is impor-
206 tant, compound terms are tried to be abstracted to the ground type first, represented by
207 the dashed edge. We use the least upper bound and greatest lower bound operations as
208 they are induced by the type subset relation. This analysis is done statically and without
209 concrete interpretation of Prolog code, based on *plspec* annotations and term literals.

210 *Annotations.* *plstatic* works without additional annotations in the analysed code. It de-
211 rives type information from (a large subset of) built-in (ISO) predicates, that we manu-
212 ally provided pre- and postconditions for. We also annotated a few popular libraries, e.g.
213 the lists library. For predicates lacking annotations, types can be derived if type infor-
214 mation exists for predicates called in their body, or can be inferred from unification with
215 term structure in the code. Derived types describe intended success for the unannotated
216 predicate. Naturally, precision of the type analysis improves with more annotations.

217 4.1 Tool Architecture

218 *plstatic* is implemented in Clojure. An alternative was to implement a meta-interpreter
219 in Prolog. A JVM-based language allows easier integration into text editors, IDEs and
220 potentially also web services. However, this requires to extract a representation of the
221 Prolog program. We decided against parsing Prolog due to operator definitions and
222 loss of term expansion[1]. Instead, we add a term expander ourselves before we load the
223 program. It implements *plspec*'s syntax for annotations and extracts those alongside the
224 program itself. All gathered information is transformed to edn[2].
225 *plstatic* consists of two parts pictured in Fig. 2: a binary (jar) that contains the static
226 analysis core, and a term expander written in Prolog, The analysis core is started with
227 parameters specifying the path to a Prolog source file or directory and a Prolog di-
228 alect (for now, "swipl" or "sicstus"). Additionally, the path to the term expander can be
229 passed as an argument as well, if another syntax for annotations than *plspec*'s is desired.
230 Regarding module resolution, special care has to be taken when an entire directory
231 is analysed: when modules are included, it is often not obvious where a predicate is
232 located. It can be hard to decide whether a predicate is user-defined, shipped as part of
233 a library or part of the built-in predicates available in the user namespace. Thus, when
234 the edn-file is imported, a data structure is kept in order to resolve calls correctly.
235 As our evaluation in Section 5 uses untrusted third-party code, we take care that the
236 Prolog code, that may immediately run when loaded, is not executed. Instead, the term
237 expander does not return any clause, effectively removing the entire program during
238 compilation. Trusted term expanders can be loaded beforehand if required.

239 4.2 Analysis

240 Our approach to type inference implements a classical abstract interpreter. Each clause
241 is analysed individually in a first phase. We use *plspec*'s annotations of the clause and

[1] Term expansion is a mechanism that allows source-to-source transformation.
[2] https://github.com/edn-format/edn

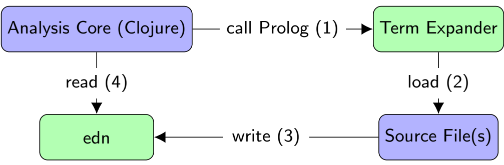

Fig. 2. Tool Architecture

the sub-goals to derive an abstract type domain for all terms in the clause. In a second phase, those results are combined: After the first phase, we have obtained a typing for every clause, which describes the types that the terms have after a successful execution of the clause. The inferred type information for all clauses of a predicate, can be stored as a postcondition. This postcondition may be more accurate than the already provided one. In this case, the analysis of a predicate would in turn improve the analysis result for clauses that call that predicate.

For this reason, *plstatic* works in two phases: first, clause-local analysis that is based on already known information, and, second, merging information of all clauses of a single predicate, propagating newly gained information to the caller(s). Without the presence of a `one-of` combinator, this would guarantee a fixed point as a result of the analysis. As we cannot infer recursive datatypes yet, which might result in infinite `one-of`-sequences, we limit the number of steps in order to ensure termination.

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 For every loaded predicate, we check, if there are pre- and postconditions already specified, ones provided by the user or our own manual annotations of ISO predicates. Otherwise, they are created containing any-types during the preparation as follows: all literals, e.g., lists, compound or atomic terms, in the clause head are considered: their type is already known after loading the program. For variable literals, however, we initially assume the type `any`. Additionally, if not annotated otherwise, we assume that a clause may be called by a variable. Based on this information, we create initial pre- and postconditions for all predicates, considering the *entire* argument vector.

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 nondeterministic nature of Prolog, it is not sufficient to store the current type for a variable at a given point: we also

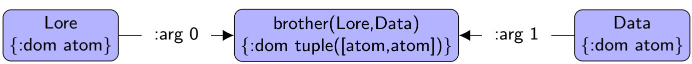

Fig. 3. An Example Environment (Using edn-Formatted Maps)

have to consider relationships between several terms that are caused by unification. Such relationships are stored in an environment, for which we use a directed graph per clause. The inferred types of the terms are stored in the vertices. Relationships between terms and sub-terms e.g. `[H|T]`, where head and tail might have a dependency on the entire list term (e.g., `list(int)`), or postconditions are saved as labelled edges between the term vertices. An example showing the structure of a compound term `brother(Lore, Data)` is given in Fig. 3.

During the analysis of a clause, the type domains of the terms are updated and their precision is improved. We assume that each predicate call in the body has to succeed, and gather information from their pre- and postconditions. 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, this relies on the assumption 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.

This raises the issue of tuple distributivity. Consider a predicate `cake(X, Y)` that 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. For both `X` and `Y`, one could derive `one_of([atom, int])` as type information. However, this would render `X=1,Y=2` to be valid input, as the individual type constraint are fulfilled, yet, the original precondition is violated.

As we aim at keeping the most precise type information possible, 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, along with its type `one_of([tuple([atom,int]), tuple([int,atom])])`. Once we know a more specific type for, e.g., `Y`, we can derive which option must be valid for the "watcher", and we can derive a type for `X`. The environment is pictured in Fig. 4.

Due to page limitations, we only consider the environment of `rate_my_ship/2` here: in this step, it infers types for `S`, `R` and the entire argument vector `[S,R]`.

Step 2: Evaluate Body. We analyse the body step by step, making use of (generated or annotated) pre- and postconditions of all encountered sub-goals. This allows us to refine the type step by step: for example, if `member(X,L)` is called, one can infer that `L` must be a list on success, even if no information on the variable was known before. On the first occurrence of a term, it is added to the environment. Similarly to the clause head, at

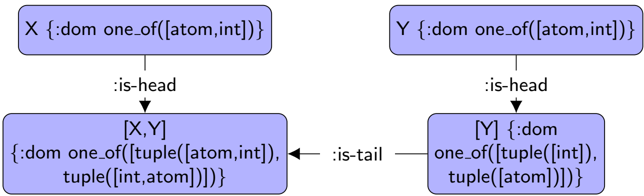

Fig. 4. Environment with a Watcher (Using edn-Formatted Maps)

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

314 least one precondition of the sub-goal must be compatible with the combination of the
315 arguments it is called with. Otherwise, for the example calling member, if `L` is known
316 not to be a list but, e.g., an integer, a type error is raised.
317 The analysis does not step into the sub-goal, and only uses pre- and postconditions.
318 A postcondition specifies type constraints on a term after the called predicate succeeds.
319 Thus, it is checked which premises of postconditions are fulfilled. Then, the greatest
320 lower bound of the current type domain and the possible conclusion of the postcondi-
321 tions is calculated in order to improve precision. An example is shown in Table 1.

322 *Type Variables.* We have introduced two new kinds of type variables (cf. Section 3):
323 `union` and `compatible`. It is possible to use `union(X)` or `compatible(X)`, where `X` is
324 a type variable. Both are placeholders for yet unknown types and express two different
325 relationships between terms:
326 Every term that is assigned the type `union(X)` contributes to the definition of the
327 type that is `X`. The connection is made by adding a labelled edge `:union` between the
328 term and `X`. Then, the domain of all contributing terms is calculated as described. At the
329 end of the analysis step, the union type of the variable `X` is inferred via the least upper
330 bound of all connected terms. As an example, if an integer and an atom is part of the
331 same union type, it will result in `one_of(int, atom)`.
332 On the other hand, terms that are assigned the type `compatible(X)`, must be
333 compatible with all other terms that are assigned that type. This implies that their
334 intersection must not be empty. As with the `union` type, we create a labelled edge
335 `:compatible` connecting the term to `X`. These edges are processed *after* all union edges
336 have been visited. For example, if a known atomic value and a known integer have to
337 be compatible within the same type variable, we can infer that both values are integer,
338 as it is the intersection of both types.
339 In order to determine the type of a type variable, it is required to know all con-
340 tributing terms. Thus, for compound or a list terms of a known size, the assigned type

is passed down to its sub-terms using the mechanisms described above. Yet, even if we know that L is a list of union(X), we do not know the list items yet – even worse, the variable may only be bound later on! 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 a connected variable is updated, so is the type variable itself. Effectively, this delays the computation of the actual type variable. 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 a special edge. 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 know the following:

1. The second argument of member contributes to the variable X in form of a *union*. We learn that X is either destiny, galactica or enterprise.
2. We learn that the variable Ship must be compatible with X, so it must be one of the three atoms named above.

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 in subsequent sub-goals, i.e. without the wrapping functor a(...), 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).

For rate_my_ship/2, three edges are added due to this step: the environment already contains the argument vector [S,R] after *Step 1*. We add that S is the head item, that [R] is the tail of the list, and that R is the head of the tail [R].

Prolog Variables. The any-type can be split into two disjoint sets: variables and non-variable terms. After processing a sub-goal, non-variable terms can only gain precision. Variables, however, 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.

384 *Step 4: Fixed-Point Algorithm.* During the prior steps, we added edges to the environ-
385 ment. These are now used to update the types of the linked terms. If the environment no
386 longer changes, we have consumed all collected knowledge and have found a prelimi-
387 nary result for a clause.
388 For example, in `rate_my_ship/2`, we will update the tuples `[R]` and `[S,R]` once
389 we learn that `R` must be of the form `compound(stars([any]))`.

390 **Phase 2: Global Propagation of Type Information** During the local analysis, each
391 clause was inspected in isolation. The type domains in the returned environments con-
392 tain the types after a successful execution of a clause *with the knowledge gained so far*.
393 The gathered information then must be propagated to the caller of the corresponding
394 predicate in order to improve the precision of the type inference.
395 Each resulting environment can be used to generate the conclusion of a postcon-
396 dition. If a predicate succeeds, at least one of its clauses succeeded. As postconditions
397 must be valid for the entire predicate, the conclusion of a new postcondition is the union
398 of all conclusions of the corresponding clauses. This newly gained knowledge (in form
399 of a postcondition) is added to the analysed data for every predicate. Afterwards, both
400 local analysis and global propagation are triggered, until a fixed-point is reached. In-
401 ferred pre- and postconditions can be written out after analysis in *plspec*'s syntax.

402 *Example: append/2.* Consider the append program:

403
```
append([], Y, Y).    append([H|T], Y, [H|R]) :- append(T, Y, R).
```

404 For the first clause, *plstatic* would derive the types `[list(any), any, any]`. For the
405 second clause, we gain no additional information from the body, because `append/2`
406 is calling itself, so we derive the types `[list(any), any, list(any)]`. To create
407 a conclusion of a postcondition for the predicate, we need to combine the results of
408 the two clauses. Unfortunately, as the type of the third argument is `any` in one case,
409 it swallows the more precise type `list(any)`. We obtain the following conclusion:
410 `[list(any), any, any]`. While the *intention* is that the second and third arguments
411 are lists as well, this cannot be inferred without annotations.
412 As you have probably noticed, *plstatic* has not yet found the accurate type `atom`
413 for `S` or `R` in `rate_my_ship/2`. This is because the pre- and postconditions of `ship/1`
414 have not been updated yet, so *plstatic* has no way of knowing that `S` is an atom. In
415 the first phase, we have concluded that the argument given to `ship/1` must be of type
416 `atom` after a successful execution. As `ship/1` has only one clause, we can infer the
417 postcondition: `:- post_spec(ship/1, [any], [atom]).` Analogously, we obtain
418 `:- post_spec(rating/1, [any], [compound(stars([atom]))]).`
419 The propagation of the newly gained knowledge is shown in Table 2. Afterwards
420 we can update the pre- and postconditions for `rate_my_ship/2`, but `ship/1` and
421 `rating/1` are not affected from this. If our program has no more clauses, the fixed-
422 point is reached, and the analysis stops.

423 *Backtracking.* Preconditions specify a condition which must be fulfilled at the moment
424 of the call, and postconditions can provide information about the type of the used terms

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

425 after a successful execution. The caller of a predicate is unaware which clause provided
426 the result. Thus, the union of all gained type information has to be considered in the
427 second phase. As a result, it is safe to ignore backtracking: yet, precision could in some
428 cases be improved if clause ordering and cuts (`!`) were considered.

429 5 Evaluation

430 To our knowledge, papers on type systems for Prolog usually omit an evaluation of their
431 applicability for existing, real-world Prolog code and offer insights on their type infer-
432 ence mechanisms on small toy examples, such as the well-known `append` predicate.
433 However, we want to consider code that is more involved than homework assignments.
434 There is no indication to what extent type inference approaches are applicable to the real
435 world, or how much work has to be spent re-writing code for full-fledged type systems.
436 In contrast, we baptise *plstatic* by fire and evaluate for how many variables in the
437 code we can infer a type that is more precise than `any`. For this, we use smaller SWI
438 community packages[3], as well as ProB [9], a model checker and constraint solver that
439 currently consists of more than 120 000 lines of Prolog code.

440 5.1 Known Limitations

441 Currently, we face three limitations in *plstatic*: firstly, as we try to avoid widening when-
442 ever possible, i.e., we try to use the most precise type like a `one_of` instead of generalis-
443 ing to their common supertype, performance is not too good. Analysis of small projects
444 runs neglectably fast, yet ProB requires several hours to complete a full analysis. Sec-
445 ondly, libraries throw a wrench into our scheme: modern Prolog systems pre-compile
446 the code. Hence, meta-programs, such as term expanders, cannot access their clauses.
447 Thus, library code is not considered and *plstatic* has to rely on annotations. Currently,
448 we only provide annotations for large parts of the lists library (for both *SWI Prolog* and
449 *SICStus Prolog*) and the AVL tree library (for *SICStus Prolog* only). Otherwise, for all
450 library predicates that are not annotated, an `any` type has to be assumed. Thirdly, we
451 currently do not consider disjunctions and if-then-else constructs, but may gain addi-
452 tional precision once this is implemented.
453 Additionally, there is an inherent limitation in our analysis strategy: some predicates
454 may really work on *any* type, e.g. term type checking predicates (such as `ground/1`
455 or `nonvar/1`) or the `member/2` predicate regarding the first argument. As no similar

[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 %

456 analysis for Prolog programs exists yet and type inference by hand is infeasible for
457 large programs, it is certainly hard to gauge the precision of our type inference.

458 ### 5.2 Empirical Evaluation

459 In Table 3, the results of some repos-
460 itories[4] and the mean value of the
461 198 smallest community packages is
462 shown. We give the amount of Prolog
463 variables, and the percentage of which
464 we can infer a type that is a strict sub-
465 type of `any`. For reference, we also
466 give the amount of calls to unknown
467 predicates in order to give an idea how
468 many missing types are caused by, e.g.,
469 library predicates lacking annotations.
470 Though, once a variable is assigned an
471 `any` type, the missing precision typically is passed on to terms that are interacting with
472 the `any` term as the predicate is implemented in a library.

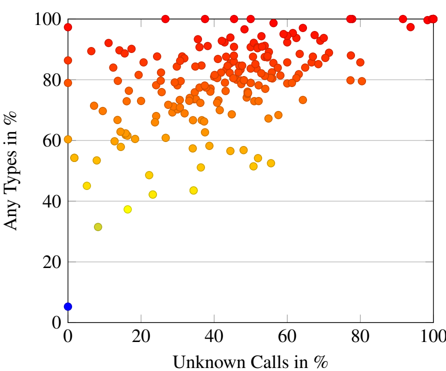

Fig. 5. Correlation Between Unknown Calls and Inferred Types

473 At first glance, the fraction of inferred types seems to be rather low. For some repos-
474 itories, such as “dia” and “thousands”, a specific type could be inferred for a large per-
475 centage of variables. Note that in return, the amount of unknown calls is relatively low.
476 Then, there are repositories such as “bddem” and “plsmf”, which both are wrappers of
477 a C library. As such, the interop predicates are unknown and the inferred types are sig-
478 nificantly lower. Finally, there are packages like “maybe”, “quickcheck” and projects
479 such as ProB, that make use of other libraries, conditional compilation, meta-calls and
480 other features that decrease accuracy of type inference.
481 Overall, we were surprised how small the amount of inferred types was. Though,
482 one has to consider that a large amount of predicates are library calls, e.g. into the
483 popular CLP and CHR libraries. In Fig. 5, we show this relation. One can clearly recog-
484 nise that (unknown) library calls negatively impact the results of our type analysis. Yet,
485 many auxiliary predicates are written to be polymorphic and deal with any type.

[4] Full results: https://github.com/pkoerner/plstatic-results/tree/wflp-20

With *plstatic*, we were able to find several errors: many SWI libraries have been broken with changes introduced in *SWI Prolog* 7 [19]. 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 in *SICStus Prolog* allow expressions such as `X is integer(3.14)` or `Y is log(2, 42)`. Thus, *plstatic* raised type errors for terms that did not match our type describing ISO 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. In several existing Prolog repositories, *plstatic* was able to locate type errors. 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 on *plstatic*: performance bottlenecks need to be reviewed. Furthermore, the analysis would heavily benefit from a mechanism for the term expander to hook into library packages, manual annotations or generated annotations based on library source code as far as it is available. 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 cannot be created manually and one of the built-in predicates *must* be called. 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. It might also be interesting to see whether our semantics can be integrated into CiaoPP.

In [18] 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 [18]. It is fairly straightforward to generate a list of already discharged annotations and use that as a blacklist in *plspec*. This could move the tool towards gradual typing [17], combining benefits of static typing and reducing overhead of static checks with the potential for many optimisations.

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*, volume 3049 of *LNCS*, 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 *Proceedings ESOP*, volume 1058 of *LNCS*, 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. J. G. Siek, M. M. Vitousek, M. Cimini, and J. T. Boyland. Refined criteria for gradual typing. In *Proceedings SNAPL*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.
18. 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.
19. J. Wielemaker. SWI-Prolog version 7 extensions. In *Proceedings CICLOPS-WLPE*, page 109, 2014.