plspec — A Specification Language for Prolog Data

Philipp Körner, Sebastian Krings

The final publication is available at Springer via
https://dx.doi.org/10.1007/978-3-030-00801-7_13.

plspec – A Specification Language for Prolog Data

Philipp Körner[0000−0001−7256−9560] and Sebastian Krings[0000−0001−6712−9798]

Institut für Informatik, Universität Düsseldorf
Universitätsstr. 1, D-40225 Düsseldorf
p.koerner@uni-duesseldorf.de krings@cs.uni-duesseldorf.de

Abstract. In general, even though Prolog is a dynamically typed language, predicates may not be called with arbitrarily typed arguments. Assumptions regarding type or mode are often made implicitly, without being directly represented in the source code. This complicates identifying the types or data structures anticipated by predicates. In consequence, Covington et al. proposed that Prolog developers should implement their own runtime type checking system.
In this paper, we present a re-usable Prolog library named *plspec*. It offers a simple and easily extensible DSL used to specify type and structure of input and output arguments. Additionally, an elegant insertion of multiple kinds of runtime checks was made possible by using Prolog language features such as co-routining and term expansion. Furthermore, we will discuss performance impacts and possible future usages.

Keywords: Prolog, runtime checks, type system, data specification

1 Introduction

In general, even though Prolog is a dynamically typed language, predicates may not be called with arbitrarily typed arguments. Assumptions regarding type or mode are often made implicitly, without being directly represented in the source code. In general, calling a predicate with an unintended argument might lead to stack overflows, infinite loops or any kind of undesired behavior. This complicates identifying the types or data structures anticipated by predicates.

For instance, assume you want to call a Prolog predicate in a newly acquired library. Documentation reveals that it implements the desired functionality, yet the call fails. The cause is ambiguous: it could be that the input was as intended, but no solution exists. Another possibility is that the input is unintended, but a call to a transformation predicate beforehand would have solved the issue.

Ideally, available documentation can be used to resolve any ambiguities. However, documentation in natural language has its limits: it cannot convey the entirety of information precisely and often gets outdated when changes are made to the code. As an example, consider the following excerpt taken from the documentation of `member/2` as implemented in SWI-Prolog [21]:
“member(?Element, ?List) is true if Element occurs in the List.”

One issue is that behavior is entirely undefined in case the second argument is not a list. In consequence, one cannot distinguish between failures such as `member(a, [b,c,d])`, where the second argument is a list but does not contain the element `a`, and `member(a, a)`, where the second argument is not a list.

In its current implementation, the predicate succeeds even if the second argument is not a proper list, i. e., a list not terminated by `[]`. In consequence, a call such as `member(a, [a,b|x])` is successful. Judging by the documentation alone, it remains unclear whether this is intended.

To overcome the limitations of documentation and to gain automatic verification, Covington et al. proposed that Prolog programmers should implement their own ad-hoc runtime type system [3]. Instead, we argue that by making use of Prolog language features, a simple and easily extensible DSL can be shipped as a reusable library called *plspec*.

The library is open source and freely available under MIT license. It can be downloaded from the GitHub Repository found at `https://github.com/wysiib/plspec`. It has been tested with both SWI Prolog and SICStus Prolog.

plspec is heavily influenced by *clojure.spec* [5], which was recently added to Clojure. The motivation for *clojure.spec* is similar to the one for *plspec*. Both languages are dynamically typed, often rendering it hard to identify which data should be passed to functions and what values are returned. Additionally, nested data structures can be large and confusing to inspect without tool support. Both libraries enable describing data based on construction out of small and simple building blocks. *clojure.spec* utilizes functions as building blocks, while *plspec* maintains a database of specifications described by Prolog terms.

In Prolog, we can insert runtime checks in order to distinguish between failures due to the absence of solutions and failures caused by to malformed input data. Furthermore, we can check whether variables are bound to invalid values inside of the called predicate. These kinds of errors might be hidden if the predicate fails later on due to unrelated reasons. Finally, we can add guarantees that if a predicate was called in a certain way and succeeds, variables will be bound to data in a specific format.

Note that *plspec* is more than a simple type checker for Prolog's type system. Rather, it can be seen as an additional optional [2] dependent type system:

- *plspec* does not change the semantics of annotated Prolog programs in any way.
- *plspec*'s annotations are entirely optional. In particular, one can only partially annotate predicates.
- Specs may be instrumented in order to take into account runtime values. In this case, *plspec* specifications define a system of dependent types.

In the following, we will focus on how *plspec*'s annotations can be instrumented for different types of runtime checks, including traditional contracts [12] by specifying pre- and postconditions as well as invariants on variables.

2 Usage and Semantics

Our goal is to associate predicates with information regarding type, form and mode of arguments, most importantly what a valid argument looks like.

In order to describe data, we use so-called *specs*. A spec is either defined by a programmer by registering it via an interface predicate, a combination of multiple existing specs or one of following built-ins.

2.1 Built-in Specs

We implemented most predicates that can be used to examine terms as atomic specs. These are `float`, `integer`, `number`, `atomic`, `atom`, `var`, `nonvar` and `ground`. To verify that a term matches its spec, we call the built-in Prolog predicates with the same name, ensuring that these specs bear the common meaning and are easy to understand. Additionally, we add `any` to describe any Prolog term.

Furthermore, one can describe non-scalar data using recursive specs. The spec `list(X)` is matched if and only if the value is a (potentially empty) list of elements satisfying the spec `X`. Lists with a fixed length can be described via `tuple(X)`, where `X` is a list of specs which describe the element in that position. As an example, `tuple([integer, atom])` is matched by the value `[3, a]`, but neither `[a, 3]` nor `[3, a, b]`.

Compound terms can be described via `compound(X)`, where X is a compound term with the functor the term shall have. Its arguments have to be specs that describe what kind of data should be contained in that position of the term. For example, `compound(foo(atom, var))` is matched by `foo(bar, X)`.

Finally, specs can be combined with so-called connectives. So far, built-ins are `and(X)` and `one_of(X)`, where `X` is a list of specs. In the case of `and`, all specs have to be matched. For `one_of`, it is sufficient if at least one spec is fulfilled.

2.2 Preconditions

Preconditions are a way to overcome the problems presented in Section 1. The idea is that all valid combinations of arguments to a predicate should be enumerated by the developer. In Prolog, there are multiple ways to call a predicate regarding instantiation of variables. However, with preconditions the developer can clearly state which calls were considered during implementation and testing.

In consequence, when using specs we can be sure that a failure of a predicate with a fulfilled precondition is intended behavior and, analogously, if the precondition is violated it is a type error.

In order to define a precondition, the interface predicate `spec_pre/2` is used. Apart from the predicate, it takes a list of specs as an argument which can be understood as the argument vector passed to the predicate. It is allowed to specify multiple preconditions with the semantics that at least one precondition has to be matched. Otherwise, the error handler is called. For preconditions, the value a predicate is called with is passed to the predicate implementing the spec immediately.

```
:- plspec:spec_pre(even_pred/1, [integer]). % the precondition
:- enable_spec_check(even_pred/1).      % instrumenting it
                                        % for runtime checks
even_pred(X) :-
    0 is X mod 2.

?- even_pred(0).
true  % intended success

?- even_pred(1).
false % intended failure

?- even_pred(_).
! plspec: no precondition was matched in even_pred/1
! plspec: specified preconditions were: [[integer]]
! plspec: however, none of these is matched by: [_G1322]
ERROR: Unhandled exception: plspec_error
```

Fig. 1. An Example for Preconditions

An example is shown in Fig. 1. We define a predicate `even_pred/1` that succeeds if the parameter is an even integer and fails for odd integers. In particular, the meaning of the spec is that only integer values are valid parameters. Otherwise, no guarantees are made whether there is correct behavior in this call, may it be failure or throwing an exception.

Thus, if we pass a variable to the annotated predicate, we do not get an exception from `is/2` that the arguments are not sufficiently instantiated but rather a print and an exception from *plspec*. This standard error handler can be replaced by a custom one, for example one that calls `trace` in order to start the debugger at this particular point in the program.

2.3 Invariants

Invariants have a more sophisticated semantic: intuitively, they specify the data structures that the predicate should work with. As soon as variables are bound to a value, they are checked as far as possible according to the spec. If the binding involves other variables, their check will be delayed until they get bound.

When a variable is bound to anything that cannot satisfy the spec anymore, the error handler will be called. One can specify invariants via `spec_invariant/2`. Again, the second argument is a list of specs with the same interpretation as above, i. e., for invariants the spec predicate is only called with ground values.

This allows uncovering the kind of programming error shown in Fig. 2: there, we call the predicate `invariant_violator` with an anonymous variable. In the first rule, it will be bound to the list `[1]`. However, the specification of the argument to `invariant_violator` says that it should be atomic if bound. Since `[1]` is neither a variable nor atomic, the error handler is called.

```
:- plspec:spec_invariant(invariant_violator/1, [atomic]).
:- enable_spec_check(invariant_violator/1).
invariant_violator(X) :-
    X = [1], X == [2]. % fail in a sophisticated way
invariant_violator(a).

?- invariant_violator(a).
true.

?- invariant_violator(_).
! plspec: an invariant was violated in invariant_violator/1
! plspec: the spec was: atomic
! plspec: however, the value was bound to: [1]
ERROR: Unhandled exception: plspec_error
```

Fig. 2. An Example for Invariant Violations

If we would not specify this invariant, the first rule would fail since `[1]` is not equal to `[2]`. Thus, Prolog would backtrack into the second rule and bind the variable to the atomic value `a`. The invalid binding of `X` to `[1]` could not be determined without reading the source code. In particular, unit tests could never expose this issue. This kind of programming errors might trigger unintended co-routines whose effects might be hard to pinpoint.

Invariants are implemented by making use of co-routines. Thus, if the Prolog implementation does not support this feature, only pre- and postconditions are available. If the application itself uses co-routines, the effect depends on the execution order. However, as long as these co-routines do not fail beforehand, it has no influence on *plspec*.

2.4 Postconditions

While we ensured correct calls of predicates with preconditions and that variables are never bound to "incorrect" values with invariants, postconditions are an important contract that if a certain condition held upon entry of a predicate, a second condition is implied on success. As for preconditions, the resulting value is used in order to call the predicate implementing the spec.

In particular, this allows to specify a promise that variables will be bound to values of a specified type. No promise is made if the predicate fails since no variables are bound then.

In *plspec*, one can use one or more instances of `spec_post/3` for postconditions. Apart from the predicate, it takes two lists of specs understood as argument vectors. The semantic is that if the first list of specs matches when the predicate is called, the second list of specs has to match if the predicate succeeds.

In Fig. 3, we define two postconditions for an implementation of the member predicate. The first postcondition guarantees that if the predicate succeeds and

```
:- spec_post(my_member/2, [var, any], [list(any), any]).
:- spec_post(my_member/2, [list(int), var], [list(int), int]).
my_member([H|_], H).
my_member([_|T], E) :-
    my_member(T, E).
```

Fig. 3. An Example for Postconditions

```
:- defspec(tree(X), one_of([compound(node(tree(X),X,tree(X))),
                            atom(empty)])).
```

Fig. 4. A Spec for a Tree of a Given Type

the first argument was a variable, then it will be bound to a list. A different promise is made in the second precondition: if now the first parameter of the call is a homogeneous list of type `int`, the second one is a variable and the predicate succeeds, then the variable will be bound to a value of type `int`.

3 Implementation

Specifications which are readable and easy to understand are useful for documentation purposes without any additional code being executed. In this section, we will explain how we maintain the spec database, how specs are validated and how we instrument the annotations described in Section 2 for runtime checks.

3.1 Maintenance and Addition of Specs

Specs are stored in Prolog's fact database. For simplicity, we distinguish between different kinds of specs that are handled separately. The reason for this is that they have different roles. Since *plspec* was designed with extensibility in mind, users can define specs themselves and add them to *plspec* dynamically.

In the following, we present the reason for distinguishing between different kinds of specs and present each of them. Built-in specs are implemented in the same way users could implement them without modifying *plspec*'s source code.

Aliasing `defspec/2` allows defining new specs via composing existing ones. The first argument is an alias for the resulting spec, while the second argument consists of other specs. Recursive specs are allowed. However, they should consume at least one bit of information of a term in order to avoid infinite loops.

A built-in alias for `integer` is `int`. In the database, they are stored as a dynamic fact that maps the alias to the composition of specs. If an alias is encountered by the verification predicate, it just looks up its definition and continues with that spec.

Newly defined specs might also be compound terms which pass information, e. g., inner specs, to the other specs in form of variables. As an example, Fig. 4 shows how to define a spec for a tree of elements of a given type. A tree is defined to be either the atom `empty` or a compound term with the functor `node` and three arguments: the first and last argument are trees of the same type, whereas the middle argument is any value of the given type.

Valid values for `tree(int)`, a tree of integers, include `node(empty,1,empty)` and `empty`. Neither `node(empty,not_an_integer,empty)`, where the middle value is not of the given type, nor `tree(empty,1,empty)`, where the functor does not match, are valid.

Verification via Predicates Another option is to implement a spec via a predicate that succeeds if a value is valid and fails otherwise. This can be achieved with `defspec_pred/2`, where the first argument is the new spec and the second is the predicate used for validation, possibly with some arguments specified.

Again, new specs might be compound terms and pass information to the predicate. The value that should be checked will always be appended as last argument to the predicate call.

Note that this implementation of specs is only suitable for values that are bound in a single unification step. Otherwise, another mechanism should be used as shown below. As an example, we can reuse the predicate `even_pred/1` from Fig. 1 which tests whether an integer is even or not. In order to use this predicate as a spec, it can be defined by `:- defspec_pred(even, even_pred)`.

Then, every time the spec `even` is used, `even_pred/1` is called with the value as argument. If it fails, the value is considered invalid. Since `even_pred/1` was annotated earlier, it will throw an exception if the value is not an integer.

Regarding built-ins, most atomic specs like `integer` or `nonvar` are implemented this way. When such a spec is encountered in *plspec*, the predicate is simply called with the current value.

Thus, this predicate should not have any side-effects or bind variables used in the passed term which might fire additional co-routines. In fact, checking specifications at runtime should not interfere with the execution of the annotated program in any way. In order to ensure this, we copy each term before using it to check a *plspec* annotation. If the spec predicate succeeds, the original term is compared to its copy. If a variable was bound, an error message will be printed.

Recursive Spec Predicates The third way to define specs is more involved. If a value is not bound in a single unification step but rather "consumes" only some part of the value, an appropriate spec can be registered by calling `defspec_pred_recursive/4`.

Recursive specs can be implemented based on a predicate verifying a part of the property, the "consumption" mentioned above. Afterwards, it hands back control to *plspec* and exposes new specs and variables that should be checked.

This predicate is the second argument to `defspec_pred_recursive/4`. It will be called with all arguments directly wired in the spec definition. Additionally,

the value is passed to the predicate. The last two arguments to that predicate are two variables. The first variable should be bound to a list of specs and the second variable to a list of values which might still be variables themselves. *plspec* will take these values and check them against the returned specs.

The third argument to `defspec_pred_recursive/4` is a predicate which merges the results of those checks. The basic operations `and` as well as `or` already are implemented and can be used. If a property like "exactly m out of n specs shall be true" is desired, this predicate has to be implemented by the user.

Finally, the fourth and last argument is the merge predicate which is called for invariant checks. It has to account for the fact that values might not be fully instantiated yet. In *plspec*, this predicate is implemented using co-routines in order to wait for further instantiation of the data to be verified. `and_invariant` as well as `or_invariant` are already implemented.

Internally, we implemented the checks for compound terms, lists and tuples like this. The functor of a compound term is immediately checked. Following, the specs of its arguments and the current values are returned because they might involve variables that are bound later.

As an example, consider the spec `list(int)` and the value `[1,X|T]`. A given list is deconstructed as far as possible in order to check the outer spec, i. e., the value is actually is a list. Then, the inner spec `int` is repeated for all elements. Here, we check that both `1` and `X` are integers. Since `X` is a variable, this check is handled by a co-routine that fires once `X` is bound. In presence of non-instantiated tails, the outer spec is kept and delayed until further instantiation. This means, a co-routine is set up that recursively checks that `T` also matches the spec `list(int)`. The spec `tuple(_)` is implemented similarly. In both cases, the resulting specs need to be merged with `and`.

Connectives Connectives are specs that do not consume any part of a value. While they are implemented exactly like the recursive specs above, they are stored separately. Many connectives might have infinite equivalent specs, e. g., `int` is the same as `or([int])` and `or([int, int])`. Thus, connectives are avoided when enumerating possible specs for a value.

These kind of specs are registered by calling `defspec_connective/4`, where arguments and semantics exactly match those of `defspec_pred_recursive/4`. As above, built-in examples are `one_of` as well as `and`, which allow specifying at least one or all specs have to match a value. `one_of` is implemented with `or` as the merge predicate.

3.2 Instrumenting Specifications for Runtime Checks

In order to insert runtime checks for the properties specified in *plspec* annotations, we make use of term expansion, i. e., source-to-source transformation.

Since annotations can also function as plain documentation, the user can explicitly state which predicates should be expanded by inserting runtime checks utilizing the given annotations.

```
1  my_member(A, B) :-
2    ([[var, list(any)], ...]=[] -> true
3      ; plspec_some(spec_matches([A, B], true), [[var, list(any)], ...])
4        -> true
5        ;  error_handler_pre(my_member/2, [A, B], [[var, list(any)], ...])),
6    ([[any, list(any)]]=[C]
7      -> lists:maplist(plspec:invariant_check(my_member/2), C, [D, [E|F]])
8      ;  true),
9    [A, B]=[D, [E|F]],
10   plspec:which_posts([[var, any]], [[list(any), any]], [D, [E|F]], G, H),
11   my_member(D, F),
12   lists:maplist(plspec:check_posts([D, [E|F]]), G, H).
```

Fig. 5. Expanded Recursive Rule

We will explain the term expansion on the example of the second, recursive rule of our `my_member/2` predicate shown in Fig. 3.

Consider Fig. 5: in lines 2–5, we check whether any precondition is specified. If there is at least one precondition, the `plspec_some` call will check whether at least one precondition is satisfied and an error is thrown. If no precondition was satisfied, no check will be performed. The check will simple try to conform each spec with each value the predicate was called with.

Afterwards, specified invariant checks are set up in lines 6–8. Note that there is no call to an error handler yet. Instead, the check and potential error handling happens inside of co-routines which will be described in more detail later.

The unification with the head of the rule happens in line 9. Note that `A` and `B` in line 1 are fresh variables. Otherwise, if the arguments do not unify with the head, we would not have an opportunity to catch potential errors there.

In line 10, the premises of the implications stated for postconditions are verified. Conclusions of the postconditions and whether they hold are checked again in line 12. The error handling for postconditions is not shown here because it is part of the `check_posts` predicate. Between these two steps that verify the postcondition, the original goal remains in line 11. This ensures the correct values are used for both parts of the postcondition.

3.3 Co-Routining for Invariants

Invariants are violated as soon as variables are bound to incorrect values. This can be checked by setting up a number of co-routines.

`defspec_pred` is a special case of `defspec_pred_recursive`: it consumes the entire value in one go without producing new values. The trade-off is that values for this kind of spec must be bound in a single step. Otherwise, the co-routine that blocks until the value is not a variable anymore fires on a partially instantiated term and fails. On the other hand, blocking until a value is ground does not catch errors where partial instantiation is undesired. This allows easy implementations because no internal structure of a term has to be exposed.

On the other hand, `defspec_pred_recursive` produces new specs and new values. For example, one can bind a variable to a compound term with a given

```
and_invariant([], [], _, true).
and_invariant([HSpec|TSpec], [HVal|TVal], Location, R) :-
    setup_check(Location, ResElement, HSpec, HVal),
    and_invariant(TSpec, TVal, Location, ResTail),
    both_eventually_true(ResElement, ResTail, R).

both_eventually_true(V1, V2, Res) :-
    when((nonvar(V1); nonvar(V2)),
          (V1 == true -> freeze(V2, Res = V2)
         ; nonvar(V1) -> Res = V1
         ; V2 == true -> freeze(V1, Res = V1)
         ; nonvar(V2) -> Res = V2)).
```

Fig. 6. An Implementation of `and` Based on Co-Routines

functor but bind its arguments later on. These arguments as well as their corresponding specs have to be exposed to *plspec*, that will set up new co-routines on them in return. This way, all invalid bindings of variables can be accounted for.

The tricky part is that results of subterms usually only propagate one at a time. If the third argument of a compound term is bound incorrectly, but the first argument remains a variable, *plspec* has to immediately fail. Otherwise, the first variable might not be bound at all and the error would go unnoticed.

Thus, there is a need for a second merge predicate that is able to deal with co-routines. An implementation that merges the results with the connective `and` is shown in Fig. 6.

The predicate `setup_check` will set up co-routines in the same way as the original spec did, using the exposed structure of terms. If the check succeeds, `ResElement` is bound to `true` or, otherwise, an error term containing a reason.

The connective is chained between the results. For example, if the term `foo(1, a, X)` is matched against `compound(foo(int, atom, var))`, the predicate `int(1), atom(a), var(X)` is formed. Each of the three calls is set up individually using its own co-routine. As soon as one fails, the entire formula is false and all co-routines are terminated by unifications in `both_eventually_true`.

Analogously, in order to implement `or`, a single `true` suffices in order for the formula to be true and to terminate all co-routines that were set up on the other disjuncts. Additionally, it has to be propagated when all disjuncts fail in order to throw an error. However, it is enough to check all alternatives only when we can determine *all of them*. Because we only want to raise an error if the entire disjunction evaluates to false but one alternative cannot be evaluated yet, we can understand non-termination as "still possible".

4 Performance Impact

Since all specs are checked at runtime, naturally there is an overhead. In this section, we discuss which predicates should be annotated by measuring the per-

```
member(Element, [Element|_Tail]).
member(Element, [_Head|Tail]) :-
    member(Element, Tail).
```

```
member_entry(Element, List) :-
    member(Element, List).
```

Fig. 7. Definition of `member/2`

```
:- spec_pre(member/2, [any, one_of([var, list(any)])]).
:- spec_invariant(member/2, [any, list(any)]).
:- spec_post(member/2, [any, any], [any, list(any)]).
```

Fig. 8. Possible Specs of member/2

```
:- spec_pre(reverse/3, [list(any), list(any), var]).
:- spec_pre(reverse/3, [var, list(any), list(any)]).
:- spec_invariant(reverse/3, [list(any), list(any), list(any)]).
:- spec_post(reverse/3, [list(any), list(any), var],
                        [list(any), list(any), list(any)]).
reverse(L, Rev) :-
    reverse(L, [], Rev).
reverse([], Acc, Acc).
reverse([H|T], Acc, Rev) :- !,
    reverse(T, [H|Acc], Rev).
```

Fig. 9. Annotated Version of `reverse`

formance impact caused by the runtime checks of *plspec*. As a first example, we consider `member/2` that succeeds if the second argument is a list and this list contains the first argument.

In Fig. 7, the definition of `member/2` is shown. Additionally, we define a predicate `member_entry/2` that wraps the `member/2` predicate. One could argue, that valid calls to `member/2` should have a list as a second argument. While it is totally sound that the predicate just fails if the second argument is not a list, in most cases such a call indicates a programming error somewhere in the code.

Thus, we add annotations to `member/2` and, analogously, to `member_entry/2` as shown in Fig. 8. The `spec_pre` directive allows that the element might be of any type, but the second argument is either a variable or a proper list. Secondly, `spec_invariant` ensures that if the second argument is bound, it still has to be possible for it to become a proper list. Lastly, `spec_post` guarantees that if the predicate succeeded for any input, the second argument will be a proper list.

We consider three benchmark configurations: first, the predicate is not annotated with a spec. Second, a spec is applied to the entry point, but not the recursion Third, the spec is checked in each recursion step.

These calls are made to `member/2` with an integer $Index$ and a list of integers ranging from 1 to N, and to `reverse/2` with the same list and a variable. Additionally, we benchmarked calls to `reverse` with an accumulator that is

Table 1. Runtimes and Inference Count of Multiple Kinds of Annotations.

Program	Index	Runtime (msecs)			Inferences		
		len=10	len=100	len=1000	len=10	len=100	len=1000
member	5	0	0	0	6	6	6
	10	0	0	0	11	11	11
	50	0	0	0	13	51	51
	100	0	0	0	13	101	101
	500	0	0	0	13	103	501
	1000	0	0	0	13	103	1001
member-entry	5	0	0	3	680	5180	50180
	10	0	0	3	685	5185	50185
	50	0	0	3	525	5225	50225
	100	0	0	3	525	5275	50275
	500	0	0	3	525	3945	50675
	1000	0	0	3	525	3945	51175
member-recur	5	0	2	22	4036	31216	303016
	10	0	4	46	6815	62316	617616
	50	0	20	222	6226	254416	3077716
	100	0	30	435	6226	352820	6011091
	500	0	25	1903	6226	284416	23808091
	1000	0	25	2588	6226	284416	31879370
reverse		0	0	0	13	103	1003
reverse-entry		0	0	7	1160	10250	101150
reverse-recur		2	121	11903	15675	1171905	113417205

implemented and annotated as in Fig. 9. For the entry level benchmark, we only annotate `reverse/2`, dropping the second spec in each of the argument vectors. Each run is repeated ten times and the median runtime is given.

All benchmarks were run on an Intel(R) Core(TM) i7-7700HQ CPU running at 2.80GHz We used SWI Prolog version 7.2.3 and configured it to use increased stack size by starting it with the parameters `-G100g -T20g -L2g`. Benchmarks were run sequentially to avoid issues due to scheduling or hyper-threading.

Table 1 depicts the results of the benchmarks. Columns show the length of the list split by runtime of the query as well as amount of inferences. For the member predicates, lookups of different indices are benchmarked in each row.

The programs "member" and "reverse" stand for the original predicates without annotations, whereas the suffix "entry" and "recur" distinguish between the annotation at entry-level and recursion-level respectively.

As depicted in Table 1, for both `member` and `reverse`, the amount of additional inferences and runtime is about constant if only the entry level is annotated. While still growing linearly in size with the list, impact remains reasonable.

However, if the specs are checked in every single recursion step, for `member`, the overhead quickly grows linearly in the length l of the list as well as linearly in the index i that is looked up, causing a quadratic overhead of $i * l$.

The overhead for `reverse` actually grows quadratic in the size of l. This is because in every step, the entire list without its head is validated against the spec again. We can clearly see that this becomes very slow even if list size increases moderately and such instrumentation should be avoided.

Since this overhead is enormous, recursive predicates should not be annotated. Instead of checking the same property again and again, one can annotate an invariant on entry level. Then, the performance impact is barely noticeable.

5 Related Work

The idea of integrating runtime checks based on annotations into Prolog is not new. In [18], the authors present the library `type_check` that implements an optional Mycroft-O'Keefe type system [14] for SWI and YAP Prolog. In comparison to *plspec*, `type_check` supports type variables as well as static type checks. However, mode annotations are not enforced. Thus, it is not possible to ensure that variables are instantiated before or after a call to a predicate and the semantics for (runtime) type checking is similar to invariants in *plspec*.

On the other hand, annotating pre- and postconditions has, for instance, been suggested in [9]. In contrast to our approach, the authors extend the usual notion of pre- and postconditions by annotations attached to the Prolog ports for fail and redo. In consequence, they work closer to the execution model of the underlying Prolog interpreter. Furthermore, the author provides the calling context, e. g., the parent predicate, to the specification under test. This allows for more fine-grained reasoning. Our approach on the other hand provides checking of invariants at any point of Prolog execution by means of co-routines.

The work around assertion checking in CiaoPP [17], uses abstract interpretation to try to discharge assertions at compile time. Assertions which cannot be checked statically are performed at runtime, using program transformation. To our knowledge, CiaoPP only supports Ciao Prolog. While *plspec* requires coroutining for its full functionality, pre- and postconditions work with any Prolog implementation that supports term expansion.

A different approach to testing has been followed in [13]. In contrast to our approach, the authors do not focus on the introduction of runtime checks into Ciao Prolog, but rather try to unify unit testing and runtime checking. This way, only one kind of annotation is needed for different testing purposes. We extend upon this work by the introduction of invariance annotations and the ability to use connectives as discussed in Section 3.1. So far, we have not evaluated if we can extract unit tests from our annotations, but intend to do so.

Documentation of Prolog code has been considered in [20], where the authors introduce *PlDoc*, a documentation format used for literate programming. The corresponding Prolog package has since been included in SWI Prolog. Instead of integrating documentation into the Prolog code itself, the LaTeX package *pl* [15]

embeds code into the documentation. Using the package, a single source file can be run both by any LaTeX binary and a Prolog interpreter.

Aside of Prolog, other declarative logic programming languages feature comparable systems. Mercury [19] includes a type system [7,4] together with a set of mode annotations [16]. However, the type system implemented in Mercury differs from the one we suggested: Though it supports higher-order functions, it neither allows types to be defined by a predicate nor to define a union of two types. In contrast to *plspec*, Mercury allows for type variables to be used. This makes it possible to specify, for instance, that the output of a function will have the exact same type as the input, regardless of the type itself.

Similar annotations to those in *plspec* can be found in Erlang's type specification language [8]. These are used, e. g., in the program analyzer Dialyzer [11]. In Erlang, it is only possible to create new types by defining a union of two existing types, which may be pre-defined or an atomic singleton like the number `42` or the atom `foo`. As discussed in Section 3.1, *plspec* allows to define a type for all values that fulfill a given predicate.

Furthermore, Erlang allows specifying types for higher-order functions which *plspec* does not support. Function specifications in Erlang can be regarded as pre- and postconditions in *plspec*. Just like Mercury, Erlang supports type variables.

6 Future Work

While *plspec* is capable of exposing real errors in real world Prolog applications, several improvements to the library should be made:

- The default error messages have room for improvement. Whenever possible, the smallest subterm that makes a spec invalid should be included separately. This allows developers to identify faster and easier what went wrong.
- We can imagine adding further annotations. For example, it can be desired that co-routines are terminated when a certain predicate succeeds or that predicates must never fail given their precondition is fulfilled.
- In Section 4, we found that checking annotations of recursive predicates is very slow. If we added static analysis or used gradual typing, most of that overhead could be avoided. For example, a meta-interpreter that makes use of *plspec*'s annotations could be employed for static type checking.

Apart from documentation and runtime checks, there are several applications that could benefit from these annotations and may be subject of future research.

It is desirable that for existing code, one does not have to write specs by hand. Due to the logic and declarative nature of Prolog, we can easily find matching specs to a given value by calling the verification predicate with a variable for the spec. While this allows us to generate a spec for a given value, it is not yet possible to generate a spec that matches all elements in a *series* of data.

If this functionality existed, one can think further: with additional tool support, specs as well as entire contracts could be inferred, for example, simply by running unit tests that contain only calls which are known to be valid.

Furthermore, some of these annotations could be re-usable for a partial evaluator such as LOGEN [10]. An issue with LOGEN is that even though its binding-time analysis already generates annotations, usually its user has to improve them manually. Some information that *plspec* covers, e. g., how predicates are intended to be called, might reduce the manual work required.

Another area is data generation based on a spec. We could use our annotations to generate arbitrary data featuring a certain structure or other properties.

This could be achieved by linking *plspec* to existing test frameworks for Prolog such as [1]. The authors follow an approach to test case generation and shrinking similar to Erlang's QuickCheck [6]. However, we would regard test failures as failing predicates if a spec is matched. In consequence, we would not describe actual output values in terms of input values.

Besides, often predicates only transform data into a different structure. With annotations that precisely describe different data structures passed to and returned from a predicate, it might be feasible to both repair incorrect and synthesize new programs solely based on *plspec*'s annotations.

Finally, *plspec* could make use of existing annotations, for example mode or meta-predicate annotations. They could be converted directly into our format.

7 Conclusion

In this paper, we presented the library *plspec*. It provides a DSL that can be used to document Prolog predicates in a way that is straightforward. This DSL is easily extensible without getting involved with internal implementation details and flexible enough to suit the needs of a broad range of Prolog programs. Furthermore, these annotations can be used in order to quickly and effortlessly enable runtime checks if required.

While the performance hit might be too big for recursive predicate, we argue that, firstly, most checks suffice to be made at the entry level because of the recursive implementation of specs for recursive data. Furthermore, invariants are powerful enough to catch incorrect bindings at a deeper recursion level. Secondly, *plspec* is a tool intended to catch errors during development. Our runtime checks should not be deployed as production code and if so, only very carefully.

References

1. C. Amaral, M. Florido, and V. Santos Costa. Prologcheck – Property-Based Testing in Prolog. In *Proceedings FLOPS*, volume 8475 of *LNCS*, pages 1–17. Springer, 2014.
2. G. Bracha. Pluggable Type Systems. In *OOPSLA workshop on revival of dynamic languages*, 2004.
3. M. A. Covington, R. Bagnara, R. A. O'Keefe, J. Wielemaker, and S. Price. Coding Guidelines for Prolog. *Theory and Practice of Logic Programming*, 12(6):889–927, Nov. 2012.

4. T. Dowd, Z. Somogyi, F. Henderson, T. Conway, and D. Jeffery. Run Time Type Information in Mercury. In *Proceedings PPDP*, volume 1702 of *LNCS*, pages 224–243. Springer, 1999.
5. R. Hickey. *clojure.spec - Rationale and Overview*, 2016. Available at `https://clojure.org/about/spec`.
6. J. Hughes. QuickCheck Testing for Fun and Profit. In *Proceedings PADL*, volume 4354 of *LNCS*, pages 1–32. Springer, 2007.
7. D. Jeffery. *Expressive Type Systems for Logic Programming Languages*. Dissertation, Department of Computer Science and Software Engineering, The University of Melbourne, 2002.
8. M. Jimenez, T. Lindahl, and K. Sagonas. A Language for Specifying Type Contracts in Erlang and Its Interaction with Success Typings. In *Proceedings of the 2007 SIGPLAN Workshop on ERLANG*, ERLANG '07, pages 11–17. ACM, 2007.
9. M. Kulaš. Annotations for Prolog – A Concept and Runtime Handling. In *Proceedings LOPSTR*, volume 1817 of *LNCS*, pages 234–254. Springer, 2000.
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. T. Lindahl and K. Sagonas. Detecting Software Defects in Telecom Applications Through Lightweight Static Analysis: A War Story. In *Proceedings APLAS*, volume 3302 of *LNCS*, pages 91–106. Springer, 2004.
12. D. Mandrioli and B. Meyer. Design by Contract. *Advances in Object-Oriented Software Engineering*, page 1, 1991.
13. E. Mera, P. Lopez-García, and M. Hermenegildo. Integrating Software Testing and Run-Time Checking in an Assertion Verification Framework. In *Proceedings ICLP*, volume 5649 of *LNCS*, pages 281–295. Springer, 2009.
14. A. Mycroft and R. A. O'Keefe. A polymorphic type system for Prolog. *Artificial intelligence*, 23(3):295–307, 1984.
15. G. Neugebauer. *pl – Literate Programming for Prolog with LaTeX*, 1996. Available at `https://www.ctan.org/pkg/pl`, version 3.0.
16. D. Overton. *Precise and Expressive Mode Systems for Typed Logic Programming Languages*. Dissertation, Department of Computer Science and Software Engineering, The University of Melbourne, 2003.
17. G. Puebla, F. Bueno, and M. Hermenegildo. Combined Static and Dynamic Assertion-Based Debugging of Constraint Logic Programs. In *Proceedings LOPSTR*, volume 1817 of *LNCS*, pages 273–292. Springer, 2000.
18. 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.
19. Z. Somogyi, F. J. Henderson, and T. C. Conway. Mercury, an Efficient Purely Declarative Logic Programming Language. In *Proceedings ASCS*, pages 499–512, 1995.
20. J. Wielemaker and A. Anjewierden. PlDoc: Wiki style Literate Programming for Prolog. *CoRR*, abs/0711.0618, 2007.
21. J. Wielemaker, T. Schrijvers, M. Triska, and T. Lager. SWI-Prolog. *Theory and Practice of Logic Programming*, 12(1-2):67–96, 2012.

A Changes

A.1 Since Initial Submission

- Rewritten introduction
- Improve discussion regarding correctness of approach
- Added related work regarding type systems for Prolog and to the type systems of Mercury and Erlang
- Rerun benchmarks, used median instead of average
- Addressed all minor issues pointed out by the reviewers
- Cleaned up bibliography

A.2 Since Conference

- Add related work suggested by other attendees
- Improve section on future work, discuss suggestions on how to implement static checks using meta interpreter
- Add author's orcid IDs
- Addressed some minor issues