HEINRICH HEINE
UNIVERSITÄT DÜSSELDORF

plspec

A Type System for Prolog

Philipp Körner and Sebastian Krings

Institut für Informatik
Heinrich-Heine-Universität Düsseldorf

September 20, 2017

HEINRICH HEINE
UNIVERSITÄT DÜSSELDORF

plspec

~~A Type System for Prolog~~
A Specification Language for Prolog Data

Philipp Körner and Sebastian Krings

Institut für Informatik
Heinrich-Heine-Universität Düsseldorf

September 20, 2017

What This is about

- Documentation
- Change and growth (see “Spec-ulation” by Rich Hickey)

What This is about

- Documentation
- Change and growth (see “Spec-ulation” by Rich Hickey)
- A rant

Bold Claim

HEINRICH HEINE
UNIVERSITÄT DÜSSELDORF

Non-ISO Prolog is broken.

Documentation of `member/2` in SWI

HEINRICH HEINE
UNIVERSITÄT DÜSSELDORF

member(?Elem, ?List)
True if Elem is a member of List.

```
?- member(1, [1,2,3]).
true .

?- member(0, [1,2,3]).
false.

?- member(X, [1,2,3]).
X = 1 ;
X = 2 ;
X = 3.
```

What if the second argument is not a list?
Reminder:

- A list terminator (e.g., `[]`) is a list.
- `.(X, L)` (or `'[|]'(X, L)`) is a list, iff `L` is a list.

```
?- member(a, a).
false.
```

```
?- member(a, a).
false.

?- is_list([a|b]).
false.
```

```
?- member(a, a).
false.

?- is_list([a|b]).
false.

?- member(a, [a|b]).
true.
```

Documentation of `member/2`, 2nd attempt

HEINRICH HEINE
UNIVERSITÄT DÜSSELDORF

member(?Elem, ?List)
True if List is a proper list and Elem is a member of List.
False if List is a proper list and Elem is *not* a member of List.
Arguments might not be instantiated.
Behaviour is undefined if List is not a proper list.

Possible Behaviour

- success and solution (`yes` + bindings)
- failure without solution (`no`)
- exception (ka-boom!)

Possible Behaviour

- success and solution (`yes` + bindings)
- failure without solution (`no`)
- exception (ka-boom!)
- inifinite loop

Real-World Example

- I worked on a version of a CSP [1] interpreter
- evaluate the output of a channel
- code for this already exists!

[1] Communicating Sequential Processes

```
eval_chan_out(Vals,ChanExpr,
       EvaldValueList,Chan,Span,WF) :-
  evaluate_dot_tuple([ChanExpr|Vals],Res,WF),
  (Res = tuple([Ch|VL])
     -> (EvaldValueList,Chan) = (VL,Ch)
     % an actual comment, not helpful though
      ; add_error_with_span(...),fail
  ),
  (is_a_channel_name(Chan)  -> true
     ;  add_error_with_span(...) ).
```

What are my options here?

What are my options here?

- sigh loudly and read more code

What are my options here?

- sigh loudly and read more code
- go ask my boss

What are my options here?

- sigh loudly and read more code
- go ask my boss who will go read more code

What are my options here?

- sigh loudly and read more code
- go ask my boss who will go read more code
- flip a table and go home

ISO Prolog . . .

. . . usually raises errors if an argument is the wrong type.
Why can’t we have nice things as well?

ISO Prolog . . .

. . . usually raises errors if an argument is the wrong type.
Why can't we have nice things as well?
When did `no` become sexier than an error?

ISO Prolog . . .

. . . usually raises errors if an argument is the wrong type.
Why can't we have nice things as well?
When did `no` become sexier than an error?
Even an error is not useful enough.

Rationale

- Documentation is not enough.
- Documentation gets outdated quickly.
- Documentation should be (somewhat) enforcable.

Rationale

- Documentation is not enough.
- Documentation gets outdated quickly.
- Documentation should be (somewhat) enforcable.
- can we describe (this part of) our program with Prolog data?

Introducing. . .

HEINRICH HEINE
UNIVERSITÄT DÜSSELDORF

plspec

https://www.github.com/wysiib/plspec/

Related Work (excerpt)

- clojure.spec
- design by contract (Racket, . . .)
- Mercury
- Erlang's type specification language
- typed Prolog

plspec's Built-ins

- `any`
- `var`, `nonvar`, `ground`
- `int`, `float`, `number`
- `atom`, `atomic`
- `compound(X)`, `list(X)`, `tuple(X)` [2]
- `one_of(X)`, `and(X)`

[2] fixed-size list

Describing Data

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

Describing Data

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

tree(int):

```
empty
node(empty, 1, empty)
tree(empty, 1, empty)
tree(empty, empty, empty)
```

Dependent Types

```
even_pred(X) :-
    0 is X mod 2.

:- defspec_pred(even, even_pred).
```

Dependent Types

```
even_pred(X) :-
    0 is X mod 2.

:- defspec_pred(even, even_pred).
```

even:

```
-2
-1
0
1
```

Kinds of Runtime Checks

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

:- spec_invariant(my_member/2,
                  [any, list(any)]).

:- spec_post(my_member/2,
             [any, any],
             [any, list(any)]).
```

Use Case: Runtime Checks

```
?- my_member(1, a).

! plspec: no precondition was matched
          in my_member/2
! plspec: specified preconditions were:
          [[any,one_of([var,list(any)])]]
! plspec: however, none of these is matched by:
          [1,a]
! plspec_error
```

Invariants

```
:- spec_invariant(inv_violator/1, [atomic]).
inv_violator(X) :-
    X = [1], X == [2].
inv_violator(a).

?- inv_violator(a).
true.

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

Empirical Evaluation

- performance impact of instrumentation not too bad, *but*
- do not annotate recursive predicates
- instead: wrap predicate, use invariants
- do not ship enabled specs

Empirical Evaluation

- used in parts of ProB
- able to expose known errors in old revisions
- exposed incorrect test cases

Features (for now)

required	offered
-	documentation
term expansion	run-time checks
co-routines	invariant checks

Future Work

required	offered
term expansion, co-routining	inference of specs
-	generation of conforming data
-	test-case generation
term expansion, co-routining	annotation for partial evaluation
-	synthesis of programs
term expansion	gradual typing
term expansion, ?	determinacy checker

Summary

- I’m a bad programmer and cannot cope with lots of code
- Goal: improve maintainability of Prolog programs
- optional typing can be shipped as a library
- maybe you will find it helpful