An implementation of P<: for XSB-Prolog

Spyros Hadjichristodoulou

(in XSB-Prolog)

Table of Contents

Summary

`P<:` is a type system with *parametric* and *subtyping* polymorphism for Prolog [BibRef: spyros]. This module is an implementation of `P<:` in XSB; it preprocesses a `Prolog` program by performing type analysis, and prints out whether the program was `well` or `ill` typed, along with useful debugging information.

Usage

In order to (currently) use `psub`, simply load `psub.P` in `XSB`:

```
|?- [psub].
```

and then load your XSB source file:

```
|?- [foo].
```

Your source file will then be type checked, and an appropriate message will be printed out, depending on whether the program was well-typed, or ill-typed.

psub

Introduction

This module implements the `P<:` type system, with heavy usage of *attributed variables* for the representation of *annotations* on type variables, as described in [BibRef: spyros]. The user only needs to include *type declarations* in her program, and `psub` type checks the program using those declarations as *contracts* between the programmer and the system.

There are two kinds of types that `P<:` supports; *universal* and *existential* types, or more precisely, universally and existentially quantified type variables. Universally typed predicates can be thought as *black boxes* by the programmer; she does not need to know how the definition looks like, or how execution will move through the predicate's definition. The only thing she needs to know is the *bound* of each type variable in the type of such predicate, so that she can make sure each usage will be well-typed. Examples of such predicates are, among others, our friend `append/3`, `member/2`, `length/2` and so on.

On the other hand, existentially typed predicates can be thought as predicates where each definition uses a *different* type, but the types of all definitions can be gathered under the umbrella of a bigger supertype. In this case, *type-mode* information is needed to *statically* ensure well-typedness. Examples of such predicates include situations where relations are used to describe a database, hence all definitions appear as facts:

```
employee(0,alan,42).
employee(1,kurt,3.14).
employee(2,david,10).
```

In this case, the third argument can either be a `float` or `integer`, and we map these two possibilities under a bigger type, i.e. `number`.

Typing Declarations

Typing declarations in `psub` include both type signatures for predicates, and for type constructors. Type signatures for predicates are declared using the following syntax:

```
:- pred name(T1,T2,...).
```

Where `T1,T2,...` are the types corresponding to each argument. The format of each such type is

```
T:type
```

where `T` is the name of a type variable, and `type` is a *primitive* type, i.e. *integer, float, atom, atomic, number, any.* The above declaration states that `T` is a *universally* quantified type variable; *existentially* quantified type variables are declared by appending a *carret* character in front of the variable name:

```
^T:type
```

For example, the type of the `employee/3` predicate from above would be declared as

```
:- pred employee(^T:integer,^S:atom,^W:number).
```

And the type of `append/3` as

```
:- pred append(list(T:any),list(T:any),list(T:any)).
```

Types for type constructors are declared using the syntax

```
:- type fsym(T1,T2,...) -> (T).
```

Each of the `T1,T2,...` is the type of each argument, just like before, and `(T)` is the function symbol's *return type*. For example, the type of the list constructor is declared as

```
:- type [T:any|list(T:any)] -> list(T:any).
:- type [] -> list(T:any).
```

Where the first declaration is used for non-empty lists, and the second for empty lists.

Usage and interface (psub)

- **Exports:**
  - *Predicates:*

    `_$$type_cleanup/0`, `_$$type_type_decl/2`, `_$$type_typed_pred/2`, `at/2`, `create_env/2`, `env_finalize/1`, `find_in_env/3`, `get_annotation/4`, `get_base_type/2`, `ground/2`, `is_subtype/3`, `is_subtype_def/3`, `max_ground/2`, `subtype_unify/2`, `sunif_handler/2`, `term_expansion/2`, `type_check_file/1`, `type_check_program/3`, `type_def/2`, `type_from_env/3`, `typeof/2`, `typeof_body/2`, `typeof_head/2`, `well_typed/3`, `well_typed_head/3`, `well_typed_rule/3`, `well_typed_single_goal/3`, `write_type_error/3`.
- **New operators defined:**

  `^/1` [500,fx], `<:/2` [500,xfx], `res/1` [1150,fx], `env/1` [1150,fx], `pred/1` [1150,fx], `type/1` [1150,fx].
- **Other modules used:**
  - *Application modules:*

    `basics`, `gensym`, `machine`, `num_vars`, `setof`, `string`.

Documentation on exports (psub)

`is_subtype/3`: [TABLED_PREDICATE]

Implements the subtype checker. The subtype checking algorithm of `P<:` is an adaptation of the one lying in the heart of `F<:`, the polymorphic lambda calculus. The first argument is a typing environment, and the remaining two are types, i.e. `is_subtype([X<:Y],X,Y)` is true. Notice that this is the *only* tabled predicate in `psub`; when we get to play with attributed variables, sadly tabling breaks everything

The predicate uses *default* tabling.

`find_in_env/3`: [PREDICATE]

Finds the type of a type variable in the given environment. The first argument is the environment, the second is the type variable and the third is the returned type.

`is_subtype_def/3`: [PREDICATE]

Subtype relation between primitive types. Notice that these facts are true for *any* typing environment, hence the first argument here is always an unnamed variable.

`subtype_unify/2:` [PREDICATE]

Here is where the fun begins. `subtype_unif/2` is the custom unification routine implementing replacements and subtype unification in `P<:`. Each attributed variable gets an `atv/4` term, which is structured as follows (notice that in each case, at most **one** argument is bound to a type):

- `atv(T,_,_,_)`: `T` is a primitive type and all other arguments are unnamed variables. Represents the type of a program variable in the *typing environment*
- `atv(_,T,_,_)`: `T` is a primitive type and all other arguments are unnamed variables. Represents the type of a *universally* quantified type variable in the type signature of a predicate
- `atv(_,_,T,_)`: `T` is a primitive type and all other arguments are unnamed variables. Represents the type of an *existentially* quantified type variable in the type signature of a predicate
- `atv(_,_,_,T)`: `T` is a primitive type and all other arguments are unnamed variables. Represents the type of a *mutable* type variable (whose type might change later on)

To be precise, `subtype_unify/2` simply calls `subtype_unify1/2` with the exact same arguments, so all the work is being done in the later. However, using both predicates helps in debugging. Look at the comments for more information on what each definition is.

`sunif_handler/2:` [PREDICATE]

Handler for unifying attributed variables. We have only one handler with multiple nondeterministic definitions, so that we don't assert and retract a different handler and make things horribly inefficient. Look in the individual comments for information about what each definition is.

`get_annotation/4:` [PREDICATE]

Returns the annotation of a given type variable (represented as an attributed Prolog variable). The first argument is the given variable, the second is the annotation, the third is the functor name of the annotation, and the fourth its arity.

`get_base_type/2:` [PREDICATE]

Implements the *bt()* function, returning the *base type* of a quantified type. The base type is basically a *pattern* of the type variables appearing in the type. We handle existential variables the same way they are handled in the `setof` module, i.e. test, cut, unify.

`ground/2:` [PREDICATE]

Implements the *g()* function, grounding each type variable using its annotation. Its friend, `max_ground/2`, *maximally* grounds a type, by only grounding the universally quantified type variables, thus implementing the *mg()* function

`max_ground/2:` [PREDICATE]

Implements the *mg()* function, *maximally* grounding a type, by only grounding the universally quantified type variables. Used only when type checking clause heads.

`at/2:` [PREDICATE]

Implements the *at()* function, which returns the *annotated* version of a quantified type. The first argument is the quantified type, i.e. usually a type signature, and the second is its annotated counterpart, i.e. the same type, but with each type variable replaced with an appropriate annotated one.

`type_from_env/3:` [PREDICATE]

Similar to `find_in_env/3`, but a bit more general to handle attributed variables (i.e. type variables with annotations). It defines the following cases:

- When we want the type signature of a predicate from the environment
- The first time we get the type of a program variable, we need to appropriately add the (G,T) annotation by adding an atv(T,_,_,_) attribute on the variable
- Any time we need the type of a program variable after the first occurrence, it is already annotated, so we simply return it in the third argument

`typeof/2:` [PREDICATE]

Finds the types of program terms:

- For program variables, we simply call `type_from_env/3`
- For ground terms, we use the 'inference' algorithm implemented with `type_def/2` to find the least type
- Two cases for function symbols:
  - For 0-ary function symbols, we simply return the full quantified return type of the type declaration
  - For all other function symbols, we use the types of its arguments, subtype unify with the type declaration, and return the return type
- And a final special case; because we use the program variable itself as a type variable appropriately annotated to represent its type, sometimes they get bound to more complex types, like list(T), so if we encounter one, simply return it

`type_def/2:` [PREDICATE]

Implements the 'inference' algorithm that finds the least possible type of a ground term.

`typeof_body/2:` [PREDICATE]

Implements the ETBody rule used to type check clause bodies. Notice that the 'maximal grounding' test needs not be performed here, since universally typed predicates are allowed to be typed in any legitimate way. The procedure followed is simple enough:

- We get the type of the predicate's arguments from the environment, using `typeof/2`
- We perform two subtype unifications, one between the type of the predicate's arguments and the signature's base type (which we find using `bt/2`, and one between any of these two types (after the substitutions have been applied), and the signature's annotated type (which we find using `at/2`)
- We *ground* the type by substituting every variable with its annotation/attribute

`typeof_head/2:` [PREDICATE]

Implements the ETHead rule used to type check clause bodies. There are two differences between this predicate and `typeof_body/2`:

- First, the *maximal grounding* test performed in the end, which ensures that all universally typed arguments are *maximally* typed in the heads of clauses.
- Second, the *environment finalization* step, which makes sure that the types so far 'inferred' for program variables appearing in the place of universally typed arguments will not be changed in the future

`env_finalize/1:` [PREDICATE]

Implements the *e()* function, by changing the annotations of type variables used as universally typed arguments from immutable to mutable, in order to prevent their types to be changed while type checking in the remainder of the clause

`well_typed_rule/3:` [PREDICATE]

Implements the TClause1 rule. The first argument is a typing environment, and the second a term of the form `Head :- Body`, which is type checked by

- First type checking the head using `well_typed_head/2`
- Then type checking the body using `well_typed/2`

`well_typed/3:` [PREDICATE]

Implements the TClause1, TClause2, TClause3 and TProg rules for handling the different structural cases of a Prolog program. The first argument is a typing environment, and the second a different syntactic form appearing in a Prolog clause:

- The atoms `true` and `fail` are *always* well-typed
- Conjunctions of the form `(G1,G2)` are type checked by first type checking `G1` using the typing environment, and then `G2` by using the same typing environment, after any changes have been applied to it
- Single atoms are type checked by calling the `well_typed_single_goal/2` predicate

`well_typed_single_goal/3:` [PREDICATE]

Used to type check a literal in a clause body, using `typeof_body/3`

`well_typed_head/3:` [PREDICATE]
Used to type check a clause head, using `typeof_head/3`

`term_expansion/2:` [PREDICATE]
These are the program transformation rules, and they are used to read the typing declarations and assert appropriate facts:

- For `:- pred` declarations, facts of the form `_$$type_typed_pred/2` are asserted
- For `:- type` declarations, facts of the form `_$$type_type_decl/2` are asserted
- When the `end_of_file` atom is reached, type checking is performed on the program using `type_check_program/3`, and then the database is 'cleaned up' using `_$$type_cleanup/0`.

The predicate is of type *dynamic*.

`_$$type_typed_pred/2:` [PREDICATE]
Used to represent type signature information for each predicate. The first argument is a term of the form `Name/Arity`, and the second is a list representing the tuple-type that the types of its arguments comprise

The predicate is of type *dynamic*.

`_$$type_type_decl/2:` [PREDICATE]
Used to represent type declaration information for each function symbol used. The first argument is the function symbol's name, and the second a term of the form `ftype/2`, where the first argument is the argument type, and the second is the return type

The predicate is of type *dynamic*.

`create_env/2:` [PREDICATE]
Creates a new typing environment by setting the annotation of each type variable to *null*. Since each program variable's type is represented by itself, we only need to add an `atv(null)` to each program variable appearing in a clause to get the typing environment.

`type_check_program/3:` [PREDICATE]
Implements the TProg rule for type checking a program. First we create the typing environment for each clause using `create_env/2`, and then we type check each clause in turn using `well_typed_rule/3`

`type_check_file/1:` [PREDICATE]
Type checks a single Prolog source file. We first make sure it is indeed an XSB file (by looking at the '.P' extension), and then the file is loaded so that it can be type checked by term expansion when the `end_of_file` atom is encountered.

`_$$type_cleanup/0:` [PREDICATE]

Used for cleaning up the database after type checking is performed, by retracting all the dynamic facts asserted.

`write_type_error/3:` [TABLED_PREDICATE]

Prints type error information in the console, by first appropriately removing all annotations (attributes) from type (Prolog) variables. Information about the expected kind of each type is also printed:

- `u` states that a *universally* typed argument was expected
- `e` states that an *existentially* typed argument was expected
- `m` states that a *mutably* typed argument was expected

The predicate uses *default* tabling.

References

[**Had14**] Spyros Hadjichristodoulou.
Mode-Sensitive Type Analysis for Prolog Programs.
PhD thesis, Stony Brook University, 2014.

Predicate Definition Index

Operator Definition Index

Concept Definition Index

(Index is empty)

Global Index

This is a global index containing pointers to places where concepts, predicates, modes, properties, types, applications, etc., are referred to in the text of the document. Note that

due to limitations of the `info` format unfortunately only the first reference will appear in online versions of the document.

<

^

_

A

B

C

E

F

G

I

L

M

N

P

R

S

T

W

X