An Optional Static Type System for Prolog – Online Appendix

Isabel Wingen, Philipp Körner

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

1 Formalisation of the Type System

During the analysis, different annotations declare type statements for terms. To be a valid program, all statements for a term must be compatible with each other; they all must be fulfilled. If we interpret these types as abstract domains, which describe an abstract set, compability means, that the intersection of the types is not empty. In fact, this intersection is the type, for which all given type statements are true. This is expressed in eq. (1) for arbitrary types A and B and a term T.

Because of this implication, the intersection of types is a central part of my static analysis. In *plspec* this was not necessary because all concrete values were known at run-time.

This section defines and describes the intersection of all implemented types. This list is exhaustive, which indicates, that if an intersection is **not** defined, the result is empty, which equals faulty.

$$\frac{\text{A is valid for T} \qquad \text{B is valid for T}}{A \cap B \text{ is valid for T}} \tag{1}$$

General If a type is a a subtype of another type, the intersection of those two types is the subtype itself. This is displayed in eq. (2). As known from the intersection operator, the intersection is symmetric, see eq. (3). To handle errors, I have created a special error type. The intersection with an error type, always yields an error type, see eq. (4).

$$\frac{A \leq B \qquad A \cap B}{A} \tag{2}$$

$$\frac{A \cap B}{B \cap A} \tag{3}$$

$$\frac{error \cap B}{error} \tag{4}$$

Errors in Compound Type Let *err* be a boolean function, which returns true if its parameter is an error type, and false otherwise.

A nested type is an error, if one of its components is an error.

In detail, that means, that a *List* type is faulty, if its type is faulty (see eq. (5)). Furthermore, eq. (6) and eq. (7) express, that a *Tuple* or a *Compound* type is faulty, if one of its children is faulty.

OneOf behaves a little different. Since only one of the types contained in a *OneOf* must be fulfilled, *error* types can be removed from the *OneOf* type, see eq. (8). But if a *OneOf* has no inner types left, it can not be valid, and therefore is marked faulty, see eq. (9).

And types are only valid, if all their children are valid, which is specified in eq. (10).

$$\frac{List(t) \qquad err(t)}{error} \tag{5}$$

$$\frac{Tuple(a_1,\dots,a_n) \qquad \exists k, 1 \leq k \leq n : err(a_k)}{error} \tag{6}$$

$$\frac{Compound(f(a_1,\dots,a_n)) \qquad \exists k, 1 \leq k \leq n : err(a_k)}{error} \tag{7}$$

$$\frac{OneOf(a_1,\dots,a_n) \qquad \exists k, 1 \leq k \leq n : err(a_k)}{OneOf(a_1,\dots,a_{k-1},a_{k+1},\dots,a_n)} \tag{8}$$

$$\frac{OneOf()}{error} \tag{9}$$

$$\frac{And(a_1,\dots,a_n) \qquad \exists k, 1 \leq k \leq n : err(a_k)}{error} \tag{10}$$

Intersection With Ground If a compound type is intersected with `Ground`, every component must be a ground type. This is displayed with the following rules:

$$\frac{Compound(f_1(a_1,\dots,a_n)) \cap Ground}{Compound(f_1(a_1 \cap Ground,\dots,a_n \cap Ground))} \tag{11}$$

$$\frac{Tuple(a_1,\dots,a_n) \cap Ground}{Tuple(a_1 \cap Ground,\dots,a_n \cap Ground)} \tag{12}$$

$$\frac{List(t) \cap Ground}{List(t \cap Ground)} \tag{13}$$

$$\frac{OneOf(a_1,\dots,a_n) \cap Ground}{OneOf(a_1 \cap Ground,\dots,a_n \cap Ground)} \tag{14}$$

$$\frac{And(a_1,\dots,a_n) \cap Ground}{And(Ground,a_1,\dots,a_n)} \tag{15}$$

Intersection of Compound Terms Let $n,m \in \mathbb{N}$, a_i,b_j,t,t_1,t_2 (with $1 \leq i \leq n, 1 \leq j \leq m$) types and f_1,f_2 atoms which describe the functors of compounds. As described in eq. (16), the intersection of *Tuples* is only valid, if both have the same number of arguments. The same holds for *Compounds*, but additionally, both have to share the same functor, see eq. (17). eq. (18) defines the intersection of a *List* and a *Tuple*, which always results in a *Tuple*, whose arguments are intersected with the type of the *List*. The intersection of a *List* and a *List* results in the *List*, whose type the intersection of the types of the original *Lists*, see eq. (19).

Actually, Prolog lists are only lists, if they are fully instantiated in the sense that no tail is a variable. Terms like $[1,2|T]$ are not recognized as lists, but as a compound $.(1,.(2,T))$. Consequently, *plstatic*s *List* type only describes fully instantiated lists. Partially instantiated lists are assigned an initial type using the dot-annotation. During the course of the analysis, a partially instantiated list can be stated to be fully-instantiated or vice versa. In both cases this results in a fully-instantiated list defined in eq. (20)

$$\frac{Tuple(a_1,\dots,a_n) \cap Tuple(b_1,\dots,b_m) \qquad n = m}{Tuple(a_1 \cap b_1,\dots,a_n \cap b_n)} \tag{16}$$

$$\frac{Compound(f_1(a_1,\dots,a_n)) \cap Compound(f_2(b_1,\dots,b_m)) \qquad n = m \qquad f_1 = f_2}{Compound(f_1(a_1 \cap b_1,\dots,a_n \cap b_n))} \tag{17}$$

$$\frac{List(t) \cap Tuple(a_1,\dots,a_n)}{Tuple(a_1 \cap t,\dots,a_n \cap t)} \tag{18}$$

$$\frac{List(t_1) \cap List(t_2)}{List(t_1 \cap t_2)} \tag{19}$$

$$\frac{Compound(.(a_1,a_2)) \cap List(t)}{List(t \cup a_1) \cup (a_2 \cap List(t))} \tag{20}$$

Special Case: Empty List In SWI Prolog, the empty list is both a list, and an atomic, yet not an atom. This is unique to SWI. Other Prolog dialects may handle the empty list differently. Nonetheless, it is a good idea to create a type for just the empty list. This type is called `EmptyList` and its interferences with other types is explained in the following rules:

$$\frac{List(t) \cap Atom \qquad \text{Prolog dialect is not SWI}}{EmptyList} \tag{21}$$

$$\frac{List(t) \cap Atomic}{EmptyList} \tag{22}$$

$$\frac{Tuple()}{EmptyList} \tag{23}$$

$$\frac{List(t) \cap EmptyList}{EmptyList} \tag{24}$$

OneOf and And Type Following basic logic rules, the intersection of *OneOf* and *And* types with other types behaves according to these rules:

$$\frac{OneOf(a_1)}{a_1} \tag{25}$$

$$\frac{OneOf(a_1,\dots,a_n) \cap B}{OneOf(a_1 \cap B,\dots,a_n \cap B)} \tag{26}$$

$$\frac{And(a_1)}{a_1} \tag{27}$$

$$\frac{And(a_1,\ldots,a_n)}{a_1 \cap \ldots \cap a_n} \tag{28}$$

$$\frac{And(a_1,\ldots,a_n) \cap B}{a_1 \cap \ldots \cap a_n \cap B} \tag{29}$$

Placeholder Types To express that a certain term has an arbitrary, but named type, which should be referenced somewhere else in the annotations, placeholder types are used. Whenever placeholder types are intersected with other types, these types are saved as aliases inside the placeholder type according to the following rules:

$$\frac{Placeholder(a)}{Placeholder(a|alias = Any)} \tag{30}$$

$$\frac{Placeholder(a) \cap X}{Placeholder(a|alias = X)} \tag{31}$$

$$\frac{Placeholder(a|alias = Y) \cap X}{Placeholder(a|alias = And(X,Y))} \tag{32}$$

Intersection with Variables The `Var` is an odd and curious case. Variable terms are able to change their type from `Any` to any other type if they become bound. For example, if you write `X = 1`, X is unified with `1` and therefore the type of `X` changes to `Int`. For the abstract analysis it is important, that a variable term with type `Var` does not match with a grounded precondition like `[Atom]`. Otherwise, the annotated spec would be bypassed, and an invalid call would be accepted. The conclusion of this thought is, that preconditions are not allowed to change the type of variable terms from `Var` to anything other. This is expressed in eq. (33).

$$\frac{Var \cap B \qquad B \neq Var \qquad During\ a\ Prespec}{error} \tag{33}$$

Naturally, if a variable is passed to a predicate that *expects* a variable, this rule does not apply:

$$\frac{Var \cap Var \qquad During\ a\ Prespec}{Var} \tag{34}$$

Obviously, there must be some way to change the type of a variable term with type `Var`, as this is certainly possible in Prolog, e.g. via a unification. Another example where the type changes from `Var` to something else is a call to a (simplified) `member/2`, with the preconditions `[Var,List(Atom)]` and `Atom, Var`. After a successful execution, the first argument is certainly bound to `Atom`. This type change is also a result of unification.

Unification and comparisons can be expressed using postconditions. The usage of `=` or `==` is nothing else than a call to a predicate named `=/2` or `==/2` and, therefore, can be annotated with postconditions $0 : placeholder(a) \rightarrow [1 : placeholder(a)]$ and $1 : placeholder(a) \rightarrow [0 : placeholder(a)]$. The indirect unification in other predicates is expressed by a postcondition, where a term is variable in the premise, and bound in the conclusion. For `member/2` this would be $true \rightarrow 0 : Atom, 1 : List(Atom)$. So, it must be possible to override a `var` type during the analysis of a postcondition. This is expressed in eq. (35).

$$\frac{Var \cap B \qquad During\ a\ Postspec}{B} \tag{35}$$

Lastly, there is third possible hurdle regarding variable types. A predicate can be annotated with a precondition that demands a term with a variable type as input, but this variable is instantly unified with another term in the head of the clause. This is the case in the well-known `member/2`, where the second argument is unified with `[H|_]` or `[_|T]`.

The assigned type `Var` clashes with the actual type of the non-variable term, but this assignment is a valid cause of action because of the instant unification in the head. The intersection algorithm needs to account for that, which is expressed in eq. (36).

$$\frac{Var \cap B \qquad In\ the\ Head\ of\ a\ Clause}{B} \tag{36}$$

Userdefined Types There is the possibility to create userdefined types, which then can be used in annotations. These types are defined using already existing types. Userdefined types can have type variables as arguments. To intersect a userdefined spec with another type, it is replaced by its definition. If the userdefined spec has arguments, the arguments in the definition are replaced by the arguments used in the annotations.

$$\frac{Userdefined(x) \cap B \qquad \text{x has no arguments}}{Definition(x) \cap B} \tag{37}$$

$$\frac{Userdefined(x) \cap B \qquad args \text{ is the arglist of x}}{Definition(x)|_{args} \cap B} \tag{38}$$