Sound Type Assignment for Logic Programming

João Barbosa Mário Florido Vítor Santos Costa
Dep. of Computer Science, Faculty of Science
University of Porto, Portugal

Type systems are a powerful tool in modern programming languages. There is a lot of previous work on typing logic programs (some examples include [20, 6, 12, 7, 19, 14, 11, 17, 8, 18, 16, 15]). We argue that, to be successfully adopted by the logic programming community, the first step is to design a type system that is able to catch obvious and relevant errors at compile-time. To do so, in [1] we defined a type system which describes polymorphic well-typings for logic programs while respecting the main properties that make types useful and semantically sound.

Most of the work on types for logic programming was based on conservative approximations of the program's success set [20, 6, 19, 5, 7]. The resulting types of this approach were what we call open types [2]. Open types give little useful information, in a lot of even simple cases, such as the predicate `append`. The following are the types for each argument of the predicate `append`, when using type inference as a conservative approximation of the success set of the predicate, where "+" is type disjunction and "A" and "B" are type variables:

```
t1 = [] + [A | t1],
t2 = B,
t3 = B + [A | t3].
```

The types for the second and third argument do not filter any possible term, since they have a type variable as a member of the type definition, which can be instantiated with any type. And in fact, some calls to `append` succeed even if unintended, such as `append([],1,1)`.

The solution we found for these arguably over-general types is the definition of closed types [2], which are types where every occurrence of a type variable in constrained. We also defined a closure operation, which transforms open types into closed types, using only information provided by the set of types themselves. Applying the closure to the above set of types, the resulting set would be:

```
t1 = [] + [A | t1],
t2 = [] + [A | t2],
t3 = [] + [A | t3],
```

which are the "intended" types for the `append` predicate.

Semantics We first defined a formal semantics for descriptive types, meaning that the semantics of (untyped) programs and the semantics for types are defined separately. This differs from previous formal semantics for prescriptive types in logic programming which rely on a typed semantics for explicitly typed logic programs [11, 9]. We then prove the soundness of our type system with respect to a semantic typing relation [1]. Our semantics for programs uses a three-valued logic, with a third value *wrong*, representing a type error in run-time (following Milner semantic value *wrong* for functional programming [13]). The logic itself was previously defined by Kleene [10] and further interpreted by Bochvar [4] and Beall [3] to catch the notion of nonsense.

Example 1: Let p be a predicate with the following predicate definition:

Submitted to:
TEASE-LP 2020

```
p(X) :- X = 1 ; X = a.
```

Let $[\![]\!]_{I,\sigma}$ denote the semantic function that takes a logic program, or part of it, and returns a logic value in the set $\{true,\ false,\ wrong\}$. When we have an interpretation function I, such that:
$I(1) = 1$, $I(a) = a$, where B_1 and B_2 are two semantic domains such that $1 \in B_1$ and $a \in B_2$,
$I(p) = f_p$, such that $f_p :: B_1 \cup B_2 \to Bool$,
Given two semantic assignments for variable, σ_1 and σ_2, the semantics of this predicate is $[\![p(X) : -X = 1; X = a.]\!]_{I,[\sigma_1,\sigma_2]} = [\![X = 1]\!]_{I,\sigma_1} \vee [\![X = a]\!]_{I,\sigma_2} \Rightarrow [\![p(X)]\!]_{I,\sigma_1} \wedge [\![p(X)]\!]_{I,\sigma_2}$.
The semantic typing relation $\{X : int + atom\} \models_{P,I} (p(X) : -X = 1; X = a.) : bool$, means that we can type the predicate definition giving the type *int + atom* to the variable X. This corresponds to $\exists[\Gamma_1,\Gamma_2].\forall[\sigma_1,\sigma_2].\Big[[\![\Gamma_1]\!]_{I,[\sigma_1]} \wedge [\![\Gamma_2]\!]_{I,[\sigma_2]} \Rightarrow [\![p(X) : -X = 1; X = a.]\!]_{I,\bar{\sigma}} \in \mathbf{T}[\![bool]\!]_I\Big]$.
Note that the semantics of type *bool* is given by $\mathbf{T}[\![bool]\!]_I = \{true, false\}$. This means that the value *wrong* cannot be attributed to the predicate definition when left hand side is true. Now suppose that $\Gamma_1 = \{X : int\}$ and $\Gamma_2 = \{X : atom\}$, where B_1 is the domain of integers and B_2 the domain of atoms.
If $\sigma_1(X) \in B_1$ and $\sigma_2(X) \in B_2$, the left hand side of the implication is true. The right hand side is either true or false, since applying f_p to any of the $\sigma_i(X)$ does not return *wrong* and neither does any of the unifications in the bodies of the clause. Therefore the semantic value of the clause is not *wrong*.
If one of the $\sigma_i(X)$ does not yield a value in the previous domains, the right hand side of the implication is *wrong*, since one of the unifications yields *wrong*. But the left hand side is false, since $[\![X]\!]_{I,\sigma_1} \notin \mathbf{T}[\![int]\!]_I$ or $[\![X]\!]_{I,\sigma_2} \notin \mathbf{T}[\![atom]\!]_I$, thus the implication is still true. □

Type Assignment Using the semantics described above, in [1] we defined a type system and proved its semantics soundness. We now tackle the question of automatically producing a well typing for a program. We defined a type inference algorithm and we would like to prove that it is syntactically sound, in the sense that whenever it succeeds it produces a well typing for the program. We believe the algorithm is sound, although we are still working on the proof. We concentrated first on the implementation and on extending it to deal with richer languages. Our type inference algorithm assumes the base types $int, float, char, string$ and $atom$. It returns either open or closed types, determined by the programmer. There is an optional definition of type declarations (like data declarations in Haskell), which, if declared by the programmer, are used by the type inference algorithm. One example of such a declaration is the list datatype `:- type list(A) = [] | [A | list(A)]`. Using the previous type definition, and reading "::" as "has type", our type inference algorithm gives the following results:

PREDICATES:	TYPES:

`l(X) :- (X = [] ; X = [Y|Ys], l(Ys)).`	`l :: T1`
	`T1 = list(C)`
	`list(B) = [] + [ B | list(B) ]`

`p(X) :- X = a ; X = 3.`	`p :: T2` `T2 = atom + int`
`q(Y) :- Y =  1.23; Y = 5.`	`q :: T3` `T3 = float + int`
`h(Z) :- Z = W, Z = Q, p(W), q(Q).`	`h :: T4` `T4 = int`

References

[1] João Barbosa, Mário Florido & Vítor Santos Costa (2019): *A Three-Valued Semantics for Typed Logic Programming*. In: *Proceedings 35th International Conference on Logic Programming (Technical Communications), ICLP 2019 Technical Communications, Las Cruces, NM, USA, September 20-25, 2019*, *EPTCS* 306, pp. 36–51.

[2] João Barbosa, Mário Florido & Vítor Santos Costa (2017): *Closed Types for Logic Programming*. In: *25th Int. Workshop on Functional and Logic Programming (WFLP 2017)*.

[3] Jc Beall (2016): *Off-Topic: A New Interpretation of Weak-Kleene Logic*. *The Australasian Journal of Logic* 13(6).

[4] D.A. Bochvar & Merrie Bergmann (1981): *On a three-valued logical calculus and its application to the analysis of the paradoxes of the classical extended functional calculus*. *History and Philosophy of Logic* 2(1-2), pp. 87–112.

[5] Maurice Bruynooghe & Gerda Janssens (1988): *An Instance of Abstract Interpretation Integrating Type and Mode Inferencing*. In: *Fifth International Conference and Symposium, Washington, 1988 (2 Volumes)*, pp. 669–683.

[6] Philip W. Dart & Justin Zobel (1992): *A Regular Type Language for Logic Programs*. In: *Types in Logic Programming*, pp. 157–187.

[7] Thom W. Frühwirth, Ehud Y. Shapiro, Moshe Y. Vardi & Eyal Yardeni (1991): *Logic Programs as Types for Logic Programs*. In: *Proc. of the Sixth Annual Symposium on Logic in Computer Science (LICS '91), Netherlands,1991*, pp. 300–309.

[8] Nevin Heintze & Joxan Jaffar (1992): *Semantic Types for Logic Programs*. In: *Types in Logic Programming*, pp. 141–155.

[9] Patricia M. Hill & John W. Lloyd (1994): *The Gödel programming language*. MIT Press.

[10] S. C. Kleene (1952): *Introduction to Metamathematics*.

[11] T. L. Lakshman & Uday S. Reddy (1991): *Typed Prolog: A Semantic Reconstruction of the Mycroft-O'Keefe Type System*. In: *Logic Programming, Proceedings of the 1991 International Symposium, San Diego, California, USA, 1991*.

[12] Lunjin Lu (2001): *On Dart-Zobel Algorithm for Testing Regular Type Inclusion*. *SIGPLAN Not.* 36(9), pp. 81–85.

[13] Robin Milner (1978): *A Theory of Type Polymorphism in Programming*. *J. Comput. Syst. Sci.* 17(3), pp. 348–375.

[14] Alan Mycroft & Richard A. O'Keefe (1984): *A Polymorphic Type System for Prolog*. *Artif. Intell.* 23(3), pp. 295–307.

[15] Gopalan Nadathur & Frank Pfenning (1992): *The Type System of a Higher-Order Logic Programming Language*. In Frank Pfenning, editor: *Types in Logic Programming*, MIT Press, pp. 245–283.

[16] Frank Pfenning (1992): *Types in Logic Programming*. MIT Press, Cambridge, MA, USA.

[17] Tom Schrijvers, Maurice Bruynooghe & John P. Gallagher (2008): *From Monomorphic to Polymorphic Well-Typings and Beyond*. In: *Logic-Based Program Synthesis and Transformation, 18th International Symposium, LOPSTR 2008, Spain, July 17-18, 2008*, pp. 152–167.

[18] Tom Schrijvers, Vítor Santos Costa, Jan Wielemaker & Bart Demoen (2008): *Towards Typed Prolog*. In: *Logic Programming, 24th International Conference, ICLP 2008, Udine, Italy, December 9-13 2008, Proceedings*, pp. 693–697.

[19] Eyal Yardeni, Thom W. Frühwirth & Ehud Y. Shapiro (1992): *Polymorphically Typed Logic Programs*. In: *Types in Logic Programming*, pp. 63–90.

[20] Justin Zobel (1987): *Derivation of Polymorphic Types for PROLOG Programs*. In: *Logic Programming, Proceedings of the Fourth International Conference, Melbourne, 1987*.