Data Types in Logic Programming

João Barbosa

Faculty of Science of the University of Porto

Abstract. Type systems are a powerful tool in modern programming languages. Types in logic programming are usually based on an over approximation of the program's semantics or they have to be declared by the programmer. None of these approached as been widely accepted by the community. Based on the principles of types in functional programming, our goal is to define a highly expressive type system that deals with data types and type constraints and a type inference algorithm that automatically infers these types and is sound with respect to the type system. We plan to integrate this new type inference algorithm in the YAP Prolog System.

1 Introduction

Recent years have seen huge progress in the design and implementation of highly expressive type systems. Innovations include dependent and refinement types for functional programming languages [13, 4], that allow for type-based program analysis and verification of general program properties. We belief that this progress can be applied to other declarative paradigms such as logic programming languages. Existing work on typing logic programs has not been widely adopted by the logic programming community and we argue that, to be successful, the first step is to design a type system that is able to catch obvious and relevant errors at compile-time. To do so we propose an initial type system that describes logical data while respecting the main properties that make types useful in functional programming. Then the intention is to expand this type system with more properties that allow for a more detailed and specific characterization of programs.

2 State of the art

There have been type systems for logic programming that describe types as a conservative approximation of the program's success set [17, 3, 9, 5, 16]. Some of these approaches are based on the notion of regular types, which can be written as monadic logic programs, using unary-predicate programs to describe types [5].

Type verification and type inference algorithms have been proposed before [2, 14, 14], but they differ on whether types are considered approximations of the success set of a logic program, or whether one wants to ensure that a type

signature will be respected. Mycroft and O'Keefe formulated a paradigmatic type system for Logic Programming, which Lakshman and Reddy later called Typed Prolog [11, 8]. In this system, types of function symbols in the program are declared by the programmer and there are algorithms that reconstruct the type of the predicate having type declarations for function symbols as input.

Ciao-Prolog [6] is one of the few examples of Prolog-based languages that uses type information, through the support of polymorphic types and the inclusion of libraries for regular types [7]. There has been a revival of interest in Hindley-Milner types for Prolog systems (SWI-Prolog and YAP [15]), where a new module was introduced for type checking that allows the mixture of typed and untyped code, with type declarations and run-time type checking.

3 Work Plan

We want to define a new type system for logic programming with features present in modern functional programming type systems and which goes far beyond the simple notion of regular types. First, we'll refine the notion of regular type to the cases where they denote exactly data types in other well-known languages, achieving the desired precision of type information. These types will be referred to as closed types [1]. Afterwards, we want to add enough power to the type language to be able to express logical properties of programs using refinement types and, in general, logically qualified types.

3.1 Strategy

Following common practice in functional languages [10], in a first step we want to define semantic for logic programming which captures the notion of run-time type error. This semantics should clearly separate the notion of a run-time type error from a false query or a true query. Then our goal is the definition of a type system sound with respect to this new semantics for logic programming. The type system will be extended afterwards with a new notion of refinement types based on type constraints.

Afterwards, we want to define a type inference algorithm which is based on constraint generation and constraint solving that infers types for programs. This algorithm has to be sound with respect to the type system defined previously. Our goal is also to add type definitions to programs, that can be seen as *data* declarations in Haskell, to help the type inference process and to enable the programmer to abstract data types in the logic programming language. These data definitions mean that datatypes are identified a priori, but there is no need to declare types for each function symbol and each predicate symbol, since data definitions can be used in several predicates with a single definition. Type inference will use the data definitions to infer types closer to the programmer's intention [12].

We call *closed types* to regular types that describe data types similar to Haskell's *data* declarations. One of the requirements of these closed types is that

none of the possibilities in the disjunction of possible types is a type variable. This would correspond to an open record. We have already defined a way to transform open types into closed types through a closure operation in [1].

The closure operation allows us to go from types such as the type inferred from previous type inference algorithms for the Prolog predicate *append*:
$\tau^1_{append}(\alpha) = [\,] + [\alpha \mid \tau^1_{append}]$,
$\tau^2_{append}(\beta, \gamma) = \beta + \gamma$,
$\tau^3_{append}(\beta, \alpha) = \beta + [\alpha \mid \tau^3_{append}]$,

where α, β and γ are type variables and the symbol "+" corresponds to type union, to the following closed types corresponding to the standard data type for parametric polymorphic lists:
$\tau^1_{append}(\alpha) = [\,] + [\alpha \mid \tau^1_{append}(\alpha)]$,
$\tau^2_{append}(\alpha) = [\,] + [\alpha \mid \tau^2_{append}(\alpha)]$,
$\tau^3_{append}(\alpha) = [\,] + [\alpha \mid \tau^3_{append}(\alpha)]$.

The YAP Prolog System is a competitive system in Prolog applications that require access to large amounts of data. We think that adding to YAP highly expressive type information will definitely put it as the standard Prolog system for applications which have serious safety demands. We will integrate our previously defined algorithm in YAP. There will be a number of challenges because types, to be useful both for implementers and programmers, must be implemented efficiently. We are in a good position to successful accomplish this task by using the implementation experience of YAP itself as a well-known efficient Prolog compiler.

4 Current State of Research

We have defined a semantics for logic programming which captures the notion of type-error, and we have defined a type system which we proved to be sound with respect to the semantics. The type system has the characteristics we wanted to have and it can be easily manipulated in order to have type become more restrictive or less restrictive. This allows us to search for the best equilibrium between having an expressive and general predicate, and having useful types. This work is now submited for publication.

We are also in the midst of defining a type inference algorithm. Our approach generates constraints from a logic program and solves them using term-rewriting into a normal form that can be seen as a substitution. Then we apply the substitution to the variables that correspond to types for the predicates in the program to conclude type inference. The equality theory that we will use for the term-writing is still on-going work, although most of its characteristics are now clear to us. The *data type* declarations defining data structures will be part of the constraints, when they exist, but not mandatory, and they will be achieved following the closed types restrictions defined previously in [1], but now with a new approach based on constraint generation and constraint solving.

References

1. J. Barbosa, M. Florido, and V. Santos Costa. Closed types for logic programming. In *25th Int. Workshop on Functional and Logic Programming (WFLP 2017)*, 2017.
2. Roberto Barbuti and Roberto Giacobazzi. A bottom-up polymorphic type inference in logic programming. *Sci. Comput. Program.*, 19(3):281–313, 1992.
3. Philip W. Dart and Justin Zobel. A regular type language for logic programs. In *Types in Logic Programming*, pages 157–187. 1992.
4. Tim Freeman and Frank Pfenning. Refinement types for ml. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, PLDI '91, pages 268–277, New York, NY, USA, 1991. ACM.
5. Thom W. Frühwirth, Ehud Y. Shapiro, Moshe Y. Vardi, and Eyal Yardeni. Logic programs as types for logic programs. In *Proc. of the Sixth Annual Symposium on Logic in Computer Science (LICS '91), Netherlands,1991*, pages 300–309, 1991.
6. Manuel V. Hermenegildo, Francisco Bueno, Manuel Carro, Pedro López-García, Rémy Haemmerlé, Edison Mera, José F. Morales, and Germán Puebla. An overview of the ciao system. In *Rule-Based Reasoning, Programming, and Applications - 5th International Symposium, RuleML 2011 - Spain, 2011*, page 2, 2011.
7. Manuel V. Hermenegildo, Francisco Bueno, Manuel Carro, Pedro López-García, Edison Mera, José F. Morales, and Germán Puebla. An overview of ciao and its design philosophy. *TPLP*, 12(1-2), 2012.
8. T. L. Lakshman and Uday S. Reddy. 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*, 1991.
9. Lunjin Lu. On dart-zobel algorithm for testing regular type inclusion. *SIGPLAN Not.*, 36(9):81–85, 2001.
10. Robin Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3):348–375, 1978.
11. Alan Mycroft and Richard A. O'Keefe. A polymorphic type system for prolog. *Artif. Intell.*, 23(3):295–307, 1984.
12. Lee Naish. Types and the intended meaning of logic programs. In *Types in Logic Programming*, pages 189–216. 1992.
13. Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. Liquid types. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 159–169, New York, NY, USA, 2008. ACM.
14. Tom Schrijvers, Maurice Bruynooghe, and John P. Gallagher. 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*, pages 152–167, 2008.
15. Tom Schrijvers, Vítor Santos Costa, Jan Wielemaker, and Bart Demoen. Towards typed prolog. In *Logic Programming, 24th International Conference, ICLP 2008, Udine, Italy, December 9-13 2008, Proceedings*, pages 693–697, 2008.
16. Eyal Yardeni, Thom W. Frühwirth, and Ehud Y. Shapiro. Polymorphically typed logic programs. In *Types in Logic Programming*, pages 63–90. 1992.
17. Justin Zobel. Derivation of polymorphic types for PROLOG programs. In *Logic Programming, Proceedings of the Fourth International Conference, Melbourne, 1987*, 1987.