Polymorphic Types in Higher-Order Logic Programming

Weidong Chen*
Computer Science and Engineering
Southern Methodist University
Dallas, Texas 75275-0122, U.S.A.
wchen@seas.smu.edu

Michael Kifer†
Department of Computer Science
SUNY at Stony Brook
Stony Brook, NY 11794-4400, U.S.A.
kifer@cs.sunysb.edu
phone: +1-516-632-8459
fax: +1-516-632-8334

December 20, 1993

Technical Report 93/20
December 1993

Department of Computer Science
SUNY at Stony Brook

Abstract

This paper analyses the requirements to the notion of type correctness in logic programming and proposes several "adequacy" criteria for such a system. We then present a type theory for a *higher-order* logic programming language, HiLog [5], which is adequate in that sense. The proposed type system not only captures type errors of syntactic origin, but also ensures that all atoms true in a canonical model (such as the perfect model or the well-founded partial model) of a well-typed program are well-typed. Furthermore, type dependencies among arguments of functions and/or predicates are incorporated into the definition of well-typed terms and atoms, so that the benefits of both parametric and inclusion polymorphism are preserved. Finally types are treated as first-class objects and type declarations can be queried directly by users, making it a suitable framework for schema integration of heterogeneous databases.

Keywords: types, inclusion and parametric polymorphism, type dependency, argument constraint, reflexive type system, semantic and syntactic adequacy of type system.

*Work supported in part by the NSF grant IRI-9212074.
†Work supported in part by the NSF grant CCR-9102159.

1 Introduction

Logic programming is derived from an essentially untyped framework, namely predicate calculus. Any ground atom, no matter how ill-typed it is in the intuitive sense (e.g., a predicate *person* applied to a city), can be legitimately assigned a truth value in a Herbrand model. This raises a fundamental question of which logic programs are to be considered well-typed.

The theory of types in logic programming has been an active area of research for a decade now. Stimulated by the pioneering work of Bruynooghe [3], Mycroft and O'Keefe [23], and Mishra [21], a number of researchers extended these approaches to handle more general classes of logic programs and type systems. These works generally fall into two categories: those extending the prescriptive theory of Mycroft and O'Keefe (e.g., [7, 17, 31, 12, 16, 10, 27, 13]), and those inspired by the descriptive type inference system of Mishra [22, 18, 39, 1, 9, 26, 36, 37, 15, 11].

In the prescriptive approach, Mycroft and O'Keefe [23] adopted Milner's slogan that "well-typed programs do not go wrong", meaning that if a logic program and a query is well-typed, then any SLD-derivation of the query with respect to the program will remain well-typed. This, however, bases a theory of types upon a particular operational strategy, and thus compromises the declarative semantics of logic programs. In the descriptive approach, Mishra [21] proposed a principle for type inference of logic programs by which types of predicates must cover the denotations of the predicates in the canonic models of logic program. This principle has been generalized by Yardeni *et al* [38] for type checking in the sense that the denotations of predicates should be covered by type declarations of predicates. The purely model-theoretic approach cannot account for type errors of syntactic origin. For instance, if a predicate, p, is supposed to take arguments of type *string* only, then the rule $p(123) \longleftarrow false$ would be considered type-correct because the offending atom $p(123)$ is never derived.

Besides the very notion of well-typed logic programs, there is also an important question how expressive type systems need to be. Extensions of the type system of Mycroft-O'Keefe [23] have been proposed to support both parametric and inclusion polymorphism. These approaches tend to be too lax in enforcing parametric polymorphism. Consider a standard polymorphic type specification for **append** as an example:

$$\texttt{append} : (\texttt{list}(\alpha) \times \texttt{list}(\alpha) \times \texttt{list}(\alpha) \Rightarrow \texttt{bool}) \tag{1}$$

This is usually read as a statement that all three arguments of **append** must be lists of exactly the same type. The question arises as to the quantification of the variable α in (1). One formal way of reading this statement is:

> *For every type, α, the function* append *must belong to the functional type* $\texttt{list}(\alpha) \times \texttt{list}(\alpha) \times \texttt{list}(\alpha) \Rightarrow \texttt{bool}$ *(where a 3-ary function belongs to such a type if and only if whenever it gets three arguments of type* $\texttt{list}(\alpha)$*, its result is an element of type* `bool`*.)*

Clearly, such reading suggests universal quantification, which turns out to work as expected where types are not organized in a hierarchy. Unfortunately, with type hierarchies the inclusion polymorphism frustrates the whole plot. For instance, let **integer** be a subtype of **number**. Then letting an atom, such as `append([1.1,2.3,X],[X,1.2],[4,5])`, evaluate to *false* still leaves **append** in conformance with the type specification (1) (as interpreted above).