The Completion of Typed Logic Programs
and SLDNF-Resolution

P. M. Hill*

School of Computer Studies, University of Leeds, Leeds LS2 9JT, UK

Abstract. We consider logic programming languages with a parametric type system, first described by Mycroft and O'Keefe, that allows generic polymorphism. It is well known that provided certain conditions hold typed definite logic programs do not go wrong under SLD-resolution. Previous work has looked at how these conditions may be avoided by adding run-time type checking to the SLD-resolution. However, only definite programs have been considered and the program's theory was assumed to be given by the statements of the program and not its completion. This paper establishes results showing that the conditions are also necessary for almost all typed logic programs if the declarative semantics is the completion semantics and the procedural semantics is based on SLDNF-resolution.

1 Introduction

The type system considered here is parametric. Parametric type systems have been described in a number of papers, including [2], [3], [5], [6], [7], [9], [10], and [12]. Such a type system is one of the main features of the language Gödel [4].

The declarative semantics of untyped logic programs has traditionally been taken to be that of the completion of the program and the procedural semantics to be SLDNF-resolution (see [8]). Thus it is important to consider how these semantics can be adapted for typed logic programs. It is well-known that if the typed program satisfies two conditions (called here transparency and definitional), then SLD-resolution can be used for the procedural semantics of typed definite programs. In this paper, we show that, for the majority of programs (including definite ones), if the semantics is a typed form of the completion, these two conditions are also necessary.

The work in this paper is based on the ideas and results of [5], however all the propositions stated and proved here are new and not (to our knowledge) proved elsewhere.

The organisation of the rest of the paper is as follows. In the next section, we define a typed logic program. In Section 3, the Clark equality theory is considered in the context of typed logic programs. Sections 4 and 5 look at the declarative and procedural semantics of typed logic programs. Finally, in Section 6, there is a discussion of these results and, in particular, their significance with respect to meta-programming.

* The author gratefully acknowledges support from SERC under grant GR/H 79862
2 Typed Logic Programs

A typed logic program consists of a set of language declarations that define a first order typed language and a set of statements written in that language. A language declaration associates a function or predicate with a tuple of elements in another language, called the type language. Note that we use the notation \( \bar{\tau}, \bar{x}, \bar{t}, \bar{x}/\bar{t}, \bar{x} = \bar{t} \) to denote, respectively, \( \bar{\tau}_1, \ldots, \bar{\tau}_n, \bar{x}_1, \ldots, \bar{x}_n, \bar{t}_1, \ldots, \bar{t}_n, x_1/t_1, \ldots, x_n/t_n, x_1 = t_1, \ldots, x_n = t_n \).

In a type language for a parametric type system, the types are structured expressions defined over disjoint sets of constructors and parameters. Each constructor has an arity associated with it. A type in the type language \( T \) defined over a set of constructors \( \mathcal{C} \) is defined recursively, to be either a parameter or of the form \( C(\bar{\tau}) \) where \( C \) is a constructor of arity \( n \) in \( \mathcal{C} \) and \( \bar{\tau} \) are types in \( T \). A monotype (or ground type) is a type built entirely from constructors.

We define two kinds of language declarations: A function declaration \( F: \bar{\tau}_1 \times \cdots \times \bar{\tau}_n \rightarrow \tau \) assigns the \( n + 1 \)-tuple \( (\bar{\tau}, \tau) \) of types in \( T \) to a function \( F \) where \( n \) is the arity, \( (\bar{\tau}, \tau) \) the declared type and \( \tau \) the declared range type of \( F \). A predicate declaration \( P: \bar{\tau}_1 \times \cdots \times \bar{\tau}_n \) assigns the \( n \)-tuple of types \( \bar{\tau} \) in \( T \) to a predicate \( P \) where \( n \) is the arity and \( \bar{\tau} \) the declared type of \( P \).

A function or predicate can have further types which are obtained from the declared type by means of type substitution. Type substitution is defined for parameters and types in the same way that (term) substitution is defined for variables and terms in (untyped) languages. Thus, if \( \Theta \) is a type substitution, a function \( F \) with declared type \( (\bar{\tau}, \tau) \) has type \( (\bar{\tau}\Theta, \tau\Theta) \) and range type \( \tau\Theta \). A predicate \( P \) with declared type \( \bar{\tau} \) has type \( \bar{\tau}\Theta \).

A first order typed language \( \mathcal{L} \) is based on a type language \( T \), a set of function declarations \( \mathcal{F} \) over \( T \), a set of predicate declarations \( \mathcal{P} \) over \( T \), and a set of variables \( V \). It is assumed that each function in \( \mathcal{F} \) and each predicate in \( \mathcal{P} \) has a unique declared type and the sets \( \mathcal{F}, \mathcal{P}, \) and \( V \) are disjoint. Variables are also assigned types and a set of assignments of types to variables is called a variable typing and denoted here by \( \Gamma \). Terms and formulas for \( \mathcal{L} \) with variable typing \( \Gamma \) are defined as follows:

- A variable \( x \) is a term of type \( \tau \) if \( \Gamma \) assigns \( x \) to \( \tau \).
- \( F(\bar{t}) \) is a term of type \( \tau \) if \( F \) has type \( (\bar{\tau}, \tau) \) and \( \bar{t} \) has type \( \bar{\tau} \).
- \( P(\bar{t}) \) is an atomic formula if \( P \) has type \( (\bar{\tau}) \) and \( \bar{t} \) has type \( \bar{\tau} \).
- \( G \) is a formula if it is of the form \( \neg G_1, G_1 \land G_2, G_1 \lor G_2, G_1 \leftarrow G_2, G_1 \rightarrow G_2, G_1 \leftrightarrow G_2, \exists G_1, \) or \( \forall G_1 \), and \( G_1 \) and \( G_2 \) are formulas\(^2\).
- A term or formula is ground if it contains no variables.

\(^2\) A constant is a function of arity 0 with a declaration \( A : \tau \).

\(^3\) The formulas \( \exists G \) and \( \forall G \) denote the existential and universal quantification of all free variables in \( G \).