Abstract interpretation for type checking

G. Filè and P. Sottero
Dep. of Pure and Applied Mathematics
University of Padova
Italy
email: mat01@unipad.infn.it

Abstract. The typed logic languages are more expressive than the usual untyped ones, but run-time type-checking is in general quite costly. Compile-time type checking is a classical application of the abstract interpretation paradigm. We describe a general abstract interpretation framework and inside it we develop two new methods for the compile-time type-checking of typed logic programs. The first method applies to a restricted class of programs (those that are type-preserving and use a finite number of types) and it detects the programs that need no type-checking at all. The second one applies to any program, but, in general, it only avoids part of the run-time type-checking.

Introduction

The addition of types to logic programs makes them more expressive and it also allows error detection at compile time. The first attempt of defining such a type system for logic languages is that of Mycroft and O'Keefe [MO'K84] and it is based on Milner's work on typing applicative languages [Mil78]. Later Hanus [Han89] has deepened the semantical aspects of this type system.

In all typed languages type-checking must be enforced and already Bakus et al. [Bak57] in the Fortran compiler and Naur [Nau65] in the Algol compiler describe compile time type-checking techniques. In the present paper we use the technique of abstract interpretation for performing compile-time type-checking of typed logic languages. In [MO'K84] it is stated that the typed programs and goals satisfying the following two conditions are type-correct: terminology of [Han89] is used,

(a) all the type-declarations of the functor symbols are type-preserving, i.e., all the type variables present in the types of the arguments are also in the type of the result;
(b) all clauses satisfy the genericity condition, i.e., the head of the clause has type equal (modulo renaming) to the type of the corresponding predicate symbol,

Unfortunately these conditions are quite restrictive. Condition (a) forbids very natural functions such as, equal:a,a->bool, cf., [Han89]. Condition (b) forbids, for instance, to have ground facts in a program, such as, append([a],[b],[a,c]), when the type of append is, [list(a), list(a), list(a)].

In this paper abstract interpretation methods are described that avoid completely or partially run-time type-checking for typed logic programs that do not meet both conditions (a) and (b) above. Type declarations associate types to every node of a term t. The type of the root of t is called the global type of t, whereas the types of any other node of t is an inside type of t.

First, programs that satisfy only condition (a) are considered, these programs are called preserving. It is easy to see that for preserving programs the run-time type-checking that must be done is much less than that of the general case: when unifying two atoms A and B, as a type-checking it suffices, in fact, to pairwise unify their global types. However, no type-checking at all would be better! This is possible for an
important class of the preserving programs, those that are type-recurrent. Intuitively, a program is type-recurrent when the set of the types of the atoms appearing in its (typed) SLD-resolutions is finite. This definition is inspired by the notion of recurrent programs of [Bez89]. It is important to understand that whereas recurrent programs always terminate (when starting from a bounded goal, [Bez89]), type-recurrent programs do not necessarily terminate.

For a preserving type-recurrent program there is a compile-time test that, when satisfied, guarantees that the program analyzed is type-correct. This static test is realized by a particularly simple abstract interpretation that uses the set of all type substitutions as abstract domain and the abstract unification is just the normal unification applied to the types.

When one considers unrestricted programs, i.e., neither condition (a) nor (b) holds, it seems impossible to avoid run-time type-checking of the global and the inside types of the arguments of the atoms produced in the SLD-derivations. However, also in this case abstract interpretation can be of help. Using an appropriate abstract domain, it is possible to synthesize the following information about any given program P: for any atom A of P that is selected during an SLD-derivation, what variables of A may be instantiated to terms containing only type-preserving symbols. These terms do not require type-checking in the inside-types and thus, based on this information, at run-time some type-checking can be avoided.

Several abstract interpretation frameworks have been proposed in the past few years. The most successful are [MSo89,Bru88,KKa90]. The goal of a framework is to facilitate the task of describing a specific abstract interpretation application: one has only to define the abstract domain, the abstract unification and to show that domain and unification meet some natural conditions (relating them to the concrete domain and unification). For describing our abstract interpretation applications we have chosen to rely on the framework of [KKa90], that is based on the ÖLDT-resolution of [TSa86], refined according to the ideas introduced in [CoF91].

The paper is organized as follows. Section 2 contains all definitions concerning types. Section 3 is devoted to the description of the general abstract interpretation framework that will be the basis for the definition of the abstract interpretation applications presented in Sections 4 and 5. A short conclusion closes the paper.

2. Types

The notions related to logic programming are assumed to be known, see for instance [Llo87]. Let Pred and Func be disjoint ranked alphabets of predicate and functor symbols. Var is the infinite set of variable symbols. All considered programs use only symbols from these alphabets. If t is a term, an atom or a clause, Var(t) is the set of the variables of t.

For the definitions about types we follow [MO'K84], for a semantical approach see [Han89]. Let Tcons and Tvar be disjoint alphabets of type constructors and type variables, respectively. The variables of Tvar are $\alpha,\beta,...$ whereas the usual variables of Var are $x,y,z,...$. The set of types corresponding to Tcons and Tvar is defined by the following grammar:

\text{TYPE} ::= \text{Tvar I Tcons(TYPE*)}

The elements of TYPE are generally denoted with $\rho$, $\tau$,....and are called types. As usual, $\rho \in \text{TYPE}$ is a monotype if it does not contain any type variable, otherwise it is a polytype.