Typed Static Analysis: Application to Groundness Analysis of Prolog and λProlog

Olivier Ridoux¹, Patrice Boizumault², and Frédéric Malésieux²*

¹ IRISA, Campus universitaire de Beaulieu
F-35042 RENNES cedex, FRANCE
Olivier.Ridoux@irisa.fr

² École des Mines de Nantes, 4, rue Alfred Kastler, BP 20722
F-44307 NANTES cedex03, FRANCE
Patrice.Boizumault@emn.fr

Abstract. We enrich the domain Pos by combining it with types. This makes static analysis more precise, since deduced properties concern both terms considered as a whole, and the details of their structure, as it is defined by types. We use this enriched domain to redefine first-order groundness analysis (Prolog terms) as it is formalized by Codish and Demoen [CD95] and higher-order groundness analysis (λProlog terms) as defined by the authors [MRB98].

1 Introduction

The works presented in this article as been stimulated by the study of static analysis for λProlog [Mal99], but applies equally well to typed versions of Prolog.

Our purpose is not to define a new type analysis for Prolog or for λProlog; there are already several proposals, and in facts, Prolog and λProlog do not diverge too much as far as prescriptive typing is considered [MO84,Han89,LR91,NP92,LR96]. The prescriptive point of view considers well-typing as a property of programs, and relates it to the semantics of program via a semantic soundness theorem which says roughly that “Well-typed programs cannot go wrong” [Mil78]. In the prescriptive view, one can consider as ill-typed programs and goals with a perfectly well-defined (but undesired) semantics. For instance, one can reject every program where a call to the classical predicate append has arguments that are not lists (though append([],3,3) is a logical consequence of the standard semantics). In a sense, the prescriptive view bridges the gap between the intended semantics of a program and its actual semantics.

Our purpose is to combine existing informations about types with informations that can be expressed in a domain like Pos [CFW91,MS93]. So doing, we expect a better precision as illustrated in the following example. In a non-typed groundness analysis using Pos, if an element of a list is non-ground, all the list is said to be non-ground (e.g., [A,2,3]). An incomplete list is deemed non-ground

* This work was done while the author was at École des Mines de Nantes.
as well (e.g., [1, 2, 3|Z]), so that groundness analysis with domain $\mathcal{Pos}$ does not make a difference between these two cases. However, predicates such as $\text{append}$ see the difference because they have only to do with the structure of lists. If the length of an input list is unknown, they try to complete it by unification and backtracking, but if it is known, these predicates are deterministic. So, it is important to formalize the difference between a proper list of ground elements (which we will write $\text{list}_a \text{true}$ or, equivalently, $\text{true}$), a proper list of not-all-ground elements (written $(\text{list}_a \text{false})$) and an improper list (written $\text{false}$).

Predicates like $\text{append}$ are especially interested in the structure of lists because they are generic; i.e., they are defined for lists in general, without considering the types of elements. This is what the type declarations of these predicates indicate formally; they are polymorphic, e.g., $\text{type append (list A) } \rightarrow (\text{list A}) \rightarrow (\text{list A}) \rightarrow o$. This suggests to model the expression of abstract properties on the expression of polymorphism. So, we will derive from the presence of type variables in declarations (e.g., $A$ in $(\text{list A})$) abstract domains which are more refined than the ordinary $\mathcal{Pos}$ domain (i.e., a two-value domain: $\text{true}$ and $\text{false}$). Thus, in the case of lists, the abstract domain contains values $\text{true}$, $(\text{list}_a \text{false})$, $(\text{list}_a (\text{list}_a \text{false}))$, etc.

In the sequel, we first analyze the case of a typed variant of PROLOG in Section 2, and then the case of $\lambda$PROLOG in Section 3. Section 2 also briefly presents the domain $\mathcal{Pos}$ and the abstract compilation of groundness for PROLOG, while Section 3 contains an introduction to $\lambda$PROLOG and to the abstract compilation of groundness for $\lambda$PROLOG. We discuss correctness and termination of our proposal in Section 4.

## 2 Typed Properties for PROLOG

We combine $\mathcal{Pos}$ with simple types in the static analysis of PROLOG programs. We first recall the principles of abstract compilation, and then we expose our technique.

### 2.1 Abstract Compilation

The principle of abstract compilation is to translate a source program into an abstract program whose denotation is computed according to its concrete semantics. This can be seen as a way of implementing abstract interpretation by partially evaluating it for a given program. This technique is called abstract compilation by Hermenegildo et al. [HWD92] and originated from works by Debray

---

1 Proper/partial/incomplete are used here as in [O’K90]. I.e., A “proper” Thing is a non-variable Thing each of whose Thing arguments is a proper Thing; A “partial” Thing is either a variable or a Thing at least one of whose Thing arguments is a partial Thing; Partial Things are sometimes called “incomplete” Things.

2 Everywhere in this article, when a program symbol is reused to express an abstraction, we write its abstract version with a subscripted $a$. 