Typed Static Analysis: Application to the Groundness Analysis of Typed Prolog

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

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

13 July 2001

Abstract

We enrich the domain $\mathcal{Pos}$ used for the static analysis of Prolog programs by combining it with types. We adopt the prescriptive view on typing, and we assume that programs are well-typed in an already existing type system. Typed static analysis of Typed Prolog programs gives access to more refined properties than untyped analysis because types give information on the inductive structure of terms that untyped static analysis does not discover. The increased refinement is not in variables assigned to true (e.g., variable recognized as bound to ground terms), but rather in variables not assigned to true; they can be assigned a more informative value than false. For instance, the proposed analysis can show that a variable is bound to a nil-terminated list whose elements are not necessarily ground. We contend that this kind of property is sometimes more useful than groundness. Because
of constructors of compound types, e.g., list, the typed abstract domain can be infinite, but we show that if the so-called head-condition is satisfied by the analyzed program, then only a finite part of the domain is used.

1 Introduction

This article is a version of [RBM99] in which formal details are treated more deeply, but that is focused on Typed Prolog. The original article treated the static analysis of both Typed Prolog and λProlog. Note that though λProlog is absent from the technical contribution of this article, it is present in the formalisms used for defining types and for describing concrete and abstract semantics.

Our purpose is to benefit from an existing type analysis for enriching truth-value-based static analysis. In this flavour of static analysis, properties are expressed as truth-values and boolean functions. Combining properties and types requires a common format, which in this article is the sequent calculus. It happens to be a convenient framework for type and property specification, which is well-known concerning types [How80, HS86, Bar91], but is less well-known for static properties.

In the sequel, we first present in Section 2 an overview of our proposal in which we recall a few intuitions on types and on static analysis; then, we recall in Section 3 the static analysis of the groundness property of Prolog programs via domain Pos. In the latter section, we also rephrase static analysis of Prolog in a deduction system framework. Then, we present Typed Prolog in Section 4. In Section 5 we present the typed abstract interpretation of Typed Prolog programs for groundness. We study termination in Section 6. Finally, we compare our method with other works in Section 7, and we conclude in Section 8.

2 Overview

We present informally the main ingredients of our proposal.

Notation 1 In the following, we use the Standard Prolog notation for lists; i.e., [] is the empty list, [t] is a singleton list whose unique element
is $t$, $[t|u]$ is a cons of $t$ and $u$, and more generally, $[t_1, t_2, \ldots, t_n]$ is a nil-terminated list whose elements are $t_1, t_2, \ldots, t_n$, and $[t_1, t_2, \ldots, t_n|u]$ is a u-terminated list whose first elements are $t_1, t_2, \ldots, t_n$.

We also use the **Standard Prolog** lexical convention on names; i.e., variable names begin with an upper-case letter or an underscore (e.g., X or _x). Other names begin with a lower-case letter. We use the same convention for distinguishing type constants from type variables.

We use a curried notation for non-list terms; e.g., $(f\ ab)$ is a compound term whose constructor is $f$ and subterms are $a$ and $b$. Note that $(f\ ab)$ is equivalent to $((f\\ a)\ b)$. The reason for using this notation is that it facilitates the expression of typing rules. It is the standard notation in λProlog [MN86], and more generally, this is the preferred notation for expressing formal type systems (e.g., [Bar91]).

We use the usual arrow notation for types; e.g., $t_1 \rightarrow t_2 \rightarrow u$ is the type of something that takes two parameters of types $t_1$ and $t_2$ and returns something of type $u$. Note that $t_1 \rightarrow t_2 \rightarrow u$ is equivalent to $t_1 \rightarrow (t_2 \rightarrow u)$. Types of term constructors and of predicate constructors differ only in the type of what is returned; predicate constructors are recognized because they return a truth-value, whose type is written $\mathbf{o}$. This usage can be traced back to Church [Chu40].

In logical formula, we write the scope of operators with parentheses, and the scope of quantifiers with square brackets. This makes nested formulas easier to read.

In logic programs, we adopt the notation for connectives that is used in logic (i.e., $\wedge, \vee$, etc) instead of the Prolog notation (i.e., $\rightarrow, \lnot$, etc). This is because we will use deduction rules that are close to standard logical deduction rules. This notation makes them still closer.

## 2.1 Prescriptive typing

Typing of logic programs has traditionally been studied from two points of view. The prescriptive point of view [MO84, Han89, LR91, NP92, LR96] 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] (e.g., well-typed logic programs cannot produce ill-typed goals).

In the prescriptive view, programs and queries with a perfectly well-defined semantics can be judged ill-typed. For instance, one can reject every
PROLOG program where a call to the classical predicate \texttt{append/3} has arguments that are not lists, though \texttt{append([], 3, 3)} is a logical consequence of the standard semantics of PROLOG. In a sense, the prescriptive view bridges the gap between the intended semantics of a program and its actual semantics.

This is opposed to the \textit{descriptive} point of view, where typing is an abstraction of the semantics. From the descriptive point of view, one aims at recognizing, for instance, that the first argument of predicate \texttt{append/3} is a list in all its answers.

Several systems implement the prescriptive view, e.g., G"odel, Mercury, \texttt{\lambda PROLOG} [HL94, SHC+96, MN86, MNPS91], and in fact, they do not diverge too much as far as prescriptive typing is considered. \texttt{\lambda PROLOG} is more closely related to a logical vision of types “à la Church”, and we will follow it in the technical sections.

\section{2.2 Static analysis}

Static analysis aims at computing semantic properties of programs without executing them.

Abstract interpretation [CC77, CC92] is a method for static analysis in which the computation domain of programs is replaced by an abstract domain that is chosen as a vehicle for a target property, and that forgets about other properties.

For instance, assuming PROLOG terms as a computation domain, the fact that they contain no free variable (i.e., the groundness property) might be of interest. It might be so because one knows an optimization that applies to ground terms (e.g., one can avoid the consistency check of negation-as-failure for ground goals). In this context, an abstract domain may consist in the two boolean values, \texttt{true} and \texttt{false}, for \texttt{ground} and \texttt{non-ground}, endowed with the corresponding abstraction of unification, etc. For instance, abstract unification will represent the fact that unifying two \texttt{ground} terms will result in another \texttt{ground} term. The abstract unification gives an accurate abstraction in this case, but cannot tell anything about unifying two \texttt{non-ground} terms, so that one must add a third value, \texttt{don't know}, or consider that \texttt{false} means \texttt{non-ground} or \texttt{don't know}.

So, an abstract domain is not chosen only for expressing properties; it must also be chosen for how accurately it propagates information. Thus, it is frequent that an abstract domain contains more values than is strictly necessary for expressing a property. For instance, an early domain for groundness
analysis of logic programs [Mel86] contained such values as, e.g., “non-ground terms with a fixed functor” or “terms made of a functor applied to free variables”.

A popular abstract domain for properties like groundness is \( \mathcal{P}os \) [CFW91, MS93]. In this domain, predicates are abstracted in propositional formulas (e.g., “the first and second arguments of an answer to append/3 are ground if and only if its third argument is also ground”).

2.3 Typed static analysis

Our purpose is to incorporate existing type information in the computation of properties that can be expressed in a domain like \( \mathcal{P}os \). So doing, we expect that more refined properties can be analyzed as illustrated in the following example. We insist that our intent is not to define a new type system, but rather to benefit from an existing one.

Example 1 (Typed static analysis) In a non-typed groundness analysis using \( \mathcal{P}os \), if an element of a list is non-ground, all the list is said to be non-ground (e.g., \([A, 2, 3]\) is non-ground). An incomplete list is deemed non-ground as well (e.g., \([1, 2, 3|Z]\) is also non-ground), so that groundness analysis with a boolean domain does not make any difference between these two cases.

However, predicates such as append/3 see the difference because they have only to deal with the structure of lists. If the tail of an input list is unknown, these predicates try to complete it by unification and backtracking, but if the tail is known, they are deterministic.

So, it is important to formalize the difference between a proper\(^1\) list of ground elements (abstracted into true by groundness analysis) a proper list of not-all-ground elements, and an improper list (both abstracted into false by groundness analysis).

We believe that groundness is sometimes not the property of interest in static program analysis, but a useful approximation for a more precise property. For instance, the safety condition for negation-as-failure is that

---

\(^1\)Proper/partial/incomplete are used here as in O’Keefe’s *Craft of Prolog* [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.
a negated goal must be ground when it is called [Llo84]. However, in that
case groundness is only a safe approximation for “a goal whose execution
does not bind its free variables”. This example shows the role of groundness
as an approximation of another property, but our proposal does not help in
analyzing the desired property either. Another example is the compilation of
predicates like append/3, and our proposal brings a solution to it. If the first
argument to a goal append is not a variable then a PROLOG engine can select
one clause and need not create a choice point. This motivates a static analysis
for determining such calls to append, and the compilation of a “functional”
version of append which is to be used by these calls. The desired property is
to be bound to either nil or cons, recursively, but groundness also asks that
list elements are ground. Our method computes the desired property.

Since ill-typed programs have no semantics from the prescriptive point
of view, our method tells nothing about terms like [1|2] simply because they
make a program ill-typed. Note also that usual prescriptive type check-
ing/inference methods cannot tell if a term of type list is a proper list.

We observe that combining truth-values with types introduces significant
values between true and false. We will see that truth-value-based groundness
analysis can be recovered by mapping non-true values to false, but properness
analysis is obtained by mapping non-false values to true.

2.4 Typed static analysis vs. Typing + untyped static
analysis

In the following, we continue using the prescriptive view on types. In particu-
lar we assume that the program has been type-checked before any property
is being analyzed. On the contrary, the property itself, properness in exam-
ple 1, can be viewed either as prescriptive or as descriptive. If it is viewed
as prescriptive, programs may be rejected for not satisfying the property,
though they are well-typed and they may have a well-defined semantics. If it
is viewed as descriptive, the property can be considered as an automatically
computed comment on a program, which can be used in the processing of
the program (e.g., at compile-time).

Predicates like append/3 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., in APROLOG,
Table 1: Overview of typed analysis

<table>
<thead>
<tr>
<th>term</th>
<th>type</th>
<th>typed abstract value</th>
<th>properness</th>
<th>groundness</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1, 2]</td>
<td>(list int)</td>
<td>(list&lt;sub&gt;a&lt;/sub&gt; true) ≡ true</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>[1, X]</td>
<td>well-typed iff X : int</td>
<td>(list&lt;sub&gt;a&lt;/sub&gt; false)</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>[1</td>
<td>Y]</td>
<td>well-typed iff Y : (list int)</td>
<td>false</td>
<td>no</td>
</tr>
<tr>
<td>[1,&quot;1&quot;]</td>
<td>ill-typed</td>
<td>irrelevant&lt;sup&gt;3&lt;/sup&gt;</td>
<td>irrelevant&lt;sup&gt;3&lt;/sup&gt;</td>
<td>irrelevant&lt;sup&gt;3&lt;/sup&gt;</td>
</tr>
</tbody>
</table>

1. As inferred/checked by a type analyzer before typed analysis is done.
2. Useful, e.g., for compiling append/3.
3. Irrelevant, because of the prescriptive view.

one writes type append (list A) → (list A) → (list A) → 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 (list A)) abstract domains which are more refined than the ordinary P<sub>os</sub> domain (i.e., a two-value domain: true and false). Thus, in the case of lists, the abstract domain contains values<sup>2</sup> true, false, (list<sub>a</sub> false), (list<sub>a</sub> (list<sub>a</sub> false)), etc.

The intended meaning of an expression like (list<sub>a</sub> false) is to represent proper lists whose elements are not known to be ground or proper. Similarly, (list<sub>a</sub> (list<sub>a</sub> false)) represents proper lists whose elements are themselves proper lists whose elements are not known to be ground or proper. Note that a proper list of ground elements being itself a ground term, (list<sub>a</sub> true) must be equivalent to true in the abstract domain.

This overview can be summarized as in Table 1.

Typed static analysis is more than a combination of the results of a type analysis and of an untyped static analysis; e.g., terms [X|Y] and [X] have

<sup>2</sup>Everywhere in this article, when a program symbol is reused to express an abstraction, we write its abstract version with a subscripted a.
both type (\textit{list A}) and non-typed property \textit{false}, but the former will have typed property \textit{false} (because it is not even a proper list) and the latter typed property (\textit{list} \textit{a} \textit{false}) (because it is a proper list of non-ground elements). So, there is no functional dependency from untyped properties and types on one side, to typed properties on the other side.

3 \textbf{PROLOG groundness analysis}

Groundness is one of the most interesting and extensively studied properties of \textsc{prolog} programs. We present it rapidly via its implementation by abstract compilation. Abstract compilation is a simple and effective way of doing abstract interpretation. In particular, it allows for describing the concrete semantics and its abstract counterpart in the same language. Our proposal is not bound to abstract compilation but we think that abstract compilation is a convenient framework for explaining our proposal.

We first recall the principles of abstract compilation, then we expose its application to the analysis of groundness, and finally, we revisit it in a sequent-based framework.

3.1 Abstract compilation

The principle of abstract compilation is to translate a source program into an abstract program in the same language whose denotation is computed according to the concrete semantics. This can be considered as a way of implementing abstract interpretation by partially evaluating it for a given program. This technique is called abstract compilation by Hermenegildo \textit{et al.} [HWD92] and originated from works by Debray and Warren [DW86]. Codish and Demoen apply it to the analysis of groundness for \textsc{prolog} programs [CD95].

The analysis proceeds in several phases, but we only focus on the translation of a program $P$ into a program $P_a$ whose denotation approximates the non-ground minimal model of $P$ (e.g., as defined in the S-semantics [FLMP89]). Another important phase is to compute the denotation of $P_a$, which gives information on all possible answers. We will make an allusion to this in Section 6. A final phase extracts information on predicate calls rather than on predicate answers.

All programs must be normalized before being translated. Normalization
is independent from the analyzed property. The purpose of normalization is to make every unification step explicit, and to make sure that every goal corresponds to a single unification step. So, there must be at most one constant in every goal, and all variables of a goal must be different. More precisely, PROLOG programs are normalized in a way such that every atom occurring in a program clause is either of the form \( X = Y \), \( p(X_1, \ldots, X_k) \), or \( X_0 = f(X_1, \ldots, X_k) \) where all the \( X_i \) are distinct variables. So doing, atoms can be considered as made of one relational symbol (i.e., \( = \), \( p \), and \( =f \)) and no function symbol. The role of normalization is mostly to make the abstract compiler deal with only one symbol at a time.

The abstraction of a normalized program is obtained by replacing every atom by an abstract atom that depends on the analyzed property because it describes the relation that exists between the arguments of a successful call to the atom. So, the unique symbol of every atom is replaced by its abstract counterpart. In this article, we only consider the groundness property and its typed variant.

### 3.2 Abstract compilation for groundness

Atoms of the form \( p(X_1, \ldots, X_k) \) are abstracted into \( p_a(X_1, \ldots, X_k) \). Atoms of the form \( X_0 = f(X_1, \ldots, X_k) \) are abstracted into \( \text{iff}(X_0, [X_1, \ldots, X_k]) \), which\(^3\) is true if and only if \( X_0 \leftrightarrow (X_1 \land \ldots \land X_k) \). In particular, \( \text{iff}(X_0, []) \) is logically equivalent to \( X_0 = \text{true} \). For groundness analysis, \( \text{iff}(X_0, [X_1, \ldots, X_k]) \) can be read as “\( X_0 \) is bound to a ground term if and only if \( X_1, \ldots, X_k \) are bound to ground terms”. Atoms of the form \( X = Y \) are abstracted into \( \text{iff}(X, [Y]) \). To summarize, relational symbols \( = \), \( p \), and \( =f \) (for all \( f \)) are abstracted in \( \text{iff} \), \( p \), and \( \text{iff} \), respectively.

The abstract program is a DATALOG program [RU95] with propositional constants \( \text{true} \) and \( \text{false} \) as computation domain. Computing all the answers of a DATALOG program is a decidable problem. This property is crucial for the abstract compilation technique; the language of the abstract programs must be a decidable language. The computation domain of the abstract

\(^3\)Actually, other authors do not use a list as second argument of \( \text{iff} \). They rely on the ability of concrete Prolog systems to use the same identifier for functors with different arities: e.g., \( \text{iff} /1 \), \( \text{iff} /2 \), \( \text{iff} /3 \), etc. Since this form of overloading is not generally handled in typed variants of Prolog, we aggregate the variized sequence of arguments in a list. Note that these lists do not change the computation domain. In particular, there is no variable of type \( \text{list} \).
The abstract program can be considered as defining formulas in $\mathcal{P}_{\text{os}}$ [CFW91, MS93, AMSS98]. $\mathcal{P}_{\text{os}}$ consists of positive propositional formulas. Here, *positive* means that when all variables of a propositional formula are instantiated to $true$, the formula is equivalent to $true$. An alternative definition is that positive propositional formulas are built with connectives $\land$, $\lor$, and $\Rightarrow$, and no other connective.

Since boolean formulas can be represented as truth-tables, one can consider that predicates are abstracted as truth-tables. This fits well with the extensional reading of predicates as sets of true atoms. In fact, every line of a truth-table whose result value is $true$ (a true-line) corresponds to an atom in the extension of the predicate, and vice-versa. Every goal in the body of a predicate clause contributes to true-lines of the predicate by conjunction of true-lines of the goal predicates. All true-lines contributed by some clause of a predicate are combined in a disjunction to form the abstract meaning of the predicate. Boolean functions that can represent the abstract meaning of predicates are closed by disjunction and conjunction. Moreover, since formulas corresponding to goals are equivalences (relational symbol $iff$) the set of their true-lines contains at least a line where all positions are $true$. It is easy to verify that this property is preserved by disjunction and conjunction. This property is the extensional counterpart of the positiveness of $\mathcal{P}_{\text{os}}$.

**Example 2 (Abstract compilation of $\text{append}/3$ and $\text{reverse}/2$)** Let $P$ be the following Prolog program:

\[
\begin{align*}
\text{append}([], L, L) & . \\
\text{append}([X|Xs1], Ys, [X|Zs1]) & \iff \text{append}(Xs1, Ys, Zs1) . \\
\text{reverse}([], []) & . \\
\text{reverse}([X|Xs1], Ys) & \iff \text{reverse}(Xs1, Ys1) \land \text{append}(Ys1, [X], Ys) .
\end{align*}
\]

$P$ is normalized (left column) and abstracted in $P_a$ (right column).

\[
\begin{align*}
\text{append}(Xs, Ys, Zs) & \iff \text{append}_a(Xs, Ys, Zs) \iff \\
Xs & = [] \land Ys = Zs . \\
\text{append}(Xs, Ys, Zs) & \iff \text{append}_a(Xs, Ys, Zs) \iff \\
Xs & = [X|Xs1] \land Ys = Zs . \\
\text{append}(Xs1, Ys, Zs1) & \iff \text{append}_a(Xs1, Ys, Zs1) \iff \\
Zs & = [X|Zs1] \land \text{append}(Xs1, Ys, Zs1) . \\
\text{append}(Xs1, Ys, Zs1) & \iff \text{append}_a(Xs1, Ys, Zs1) .
\end{align*}
\]
\[
\text{reverse}(Xs, Ys) \iff \text{iff}(Xs, \[]) \land \text{iff}(Ys, \[]).
\]
\[
\text{reverse}(Xs, Ys) \iff \\
Xs = \[] \land Ys = \[].
\]
\[
\text{reverse}(Xs, Ys) \iff \\
\text{iff}(Xs, [X, Xs1]) \land \text{iff}(Ys, [Xs1, Ys1]) \land \\
A = [X, A1] \land A1 = \[] \land \\
\text{iff}(A, [X, A1]) \land \text{iff}(A1, \[]) \land \\
\text{append}(Ys1, A, Ys).
\]

The minimal model of \( P_a \) (augmented with definitions for \( \text{iff} \)) gives the following set of facts, \( M(P_a) \):

\[
\begin{align*}
\text{append}_a(\text{true}, \text{true}, \text{true}), & \quad \text{append}_a(\text{true}, \text{false}, \text{false}), \\
\text{append}_a(\text{false}, \text{true}, \text{false}), & \quad \text{append}_a(\text{false}, \text{false}, \text{false}), \\
\text{reverse}_a(\text{true}, \text{true}), & \quad \text{reverse}_a(\text{false}, \text{false}),
\end{align*}
\]

and corresponds to the following truth-tables:

<table>
<thead>
<tr>
<th>( X_1 )</th>
<th>( X_2 )</th>
<th>( X_3 )</th>
<th>( \text{append}_a \equiv (X_1 \land X_2) \iff X_3 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( X_1 )</th>
<th>( X_2 )</th>
<th>( \text{reverse}_a \equiv X_1 \iff X_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Let \( B \) be the goal \( \text{reverse}([1], \text{L}) \). Its normal form is

\[
X = [Y | Z] \land Y = 1 \land Z = \[] \land \text{reverse}_a(X, \text{L})
\]

and its abstraction \( B_a \) is

\[
\text{iff}(X, [Y, Z]) \land \text{iff}(Y, \[]) \land \text{iff}(Z, \[]) \land \text{reverse}_a(X, \text{L}).
\]
An approximation of the answer set results from executing \( B_a \) on \( M(P_a) \): \( \{ \text{reverse}_a(\text{true},\text{true}) \} \). It says that all answers to goal \( \text{reverse}([1], L) \) are ground.

This analysis says in particular that the first argument of an answer to \( \text{append}/3 \) is either ground or not, which is a trivial fact, but is of little use in applications. More informative is the fact that the first argument of an answer to \( \text{append}/3 \) is always a proper list, which is not true for the other arguments. However, this more informative fact does not belong to groundness analysis.

Note that though prescriptive typing forces all three arguments of \( \text{append}/3 \) to be of type list, it does not force all of them to be proper lists in all answers. This is why typed analysis is not a trivial combination of typing and static analysis.

### 3.3 A sequent-style presentation of abstract interpretation

For the sake of the combination with types, we give a slightly different view on the previous abstraction.

The abstraction of terms and goals is expressed as sequents \( \text{term} :_{a} \text{property} \), and deduction rules, which are a variant of a simple typing theory. They can also be considered as a variant of the deduction rules used for the syntactic equality theory.

The abstraction of compound formulas (i.e., goals, clauses, predicates, and programs) are expressed using sequents \( \text{program} \vdash_{a} \text{abstract\_goal} \) and deduction rules which are a variant of classical (in fact intuitionistic is enough) sequent calculus deduction rules. The deduction rules we propose are fully standard except for the formation of sequents; their antecedent is a program formula (i.e., a combination of clauses), while their consequent is an abstract goal formula.

Using sequents for describing the semantics of logic programming languages is not frequent, except for \( \lambda \text{Prolog} \) where this is the standard [NM88]. We prefer this formalism to the usual Herbrand-domain semantics because it avoids problems with substitutions by avoiding substitutions. Indeed, substitutions are an operational way of doing quantifier elimination, but sequent calculus deals directly with quantifier elimination.
Universal quantifiers at the clause level are eliminated by simply replacing the quantified variables by values of the abstract domain. However, abstract values cannot be plugged in a concrete term without caution. So, abstract values are wrapped in an uninterpreted symbol written \( \gamma \). Symbol \( \gamma \) can be seen as having type \( \text{abstract-domain} \rightarrow A \).

**Definition 1 (Abstraction as sequent calculus)** The property of a compound term is the conjunction of the properties of its subterms:

\[
\frac{X_1 : a \ P_1 \ \ldots \ \ X_k : a \ P_k}{f(X_1, \ldots, X_k) : a \ \land_{i \in [1,k]} P_i} \quad \text{Abstr}_f \quad \text{for all } f
\]

The property of an atomic formula is an abstract atomic formula:

\[
\frac{X_1 : a \ P_1 \ \ldots \ \ X_k : a \ P_k}{p(X_1, \ldots, X_k) : a \ p(a(P_1, \ldots, P_k))} \quad \text{Abstr}_p \quad \text{for all } p
\]

The property of a conjunction of goals is the conjunction of their properties:

\[
\frac{G_1 : a \ P_1 \ G_2 : a \ P_2}{G_1 \land G_2 : a \ P_1 \land P_2} \quad \text{Abstr}_\text{goal}
\]

The property of any concrete term of property \( X \) is \( X \):

\[
\frac{()}{(\gamma X) : a \ X} \quad \text{Abstr}_\gamma
\]

An abstract goal is provable from a program that contains a fact \( G \) if it is the property of \( G \):

\[
\frac{G : a \ A}{\mathcal{P}, G \vdash a \ A} \quad \text{Axiom}
\]

A conjunction of abstract goals is provable if both are provable:

\[
\frac{\mathcal{P} \vdash a \ A_1 \ \mathcal{P} \vdash a \ A_2}{\mathcal{P} \vdash a \ A_1 \land A_2} \quad \land_{\text{goal}}
\]

An abstract goal is provable from a program that contains a universally quantified clause, if it is provable from an instance of this clause:

\[
\frac{\mathcal{P}, C[x \leftarrow (\gamma v)] \vdash a \ A}{\mathcal{P}, \forall x[C] \vdash a \ A} \quad \forall_{\text{clause}}
\]
where \( v \) is in the abstract domain (e.g., either true or false if the abstract domain is \( \text{Bool} \)). An abstract goal is provable from a program that contains a conjunction of clauses if it is provable from any of them:

\[
\frac{\mathcal{P}, C_1 \vdash_a A}{\mathcal{P}, C_1 \land C_2 \vdash_a A} \quad \land_{\text{clause}}
\]

where \( i \) is either 1 or 2. An abstract goal is provable from a program that contains some clause, if it is provable from the program augmented with the head of the clause, and if the abstraction of the body of the clause is provable from the program:

\[
\frac{B \vdash_a B_a \quad \mathcal{P} \vdash_a B_a \quad \mathcal{P}, H \vdash_a A}{\mathcal{P}, H \Leftarrow B \vdash_a A} \quad \Leftarrow_{\text{clause}}
\]

The first rule (\( \text{Abstr}_f \)) must be instantiated for every term constructor \( f \). Similarly, the second rule (\( \text{Abstr}_p \)) must be instantiated for every predicate constructor \( p \). For instance, in the case of lists we have the two following rules:

\[
\frac{X_1 ; a ; P_1 \quad X_2 ; a ; P_2}{[X_1 \mid X_2] ; a ; P_1 \land P_2} \quad \text{Abstr}_{\text{cons}}
\]

\[
\frac{a \vdash_a \text{true}}{\hat{a} ; a ; \text{true}} \quad \text{Abstr}_{\text{nil}}
\]

Note that nullary term constructors (e.g., \( \text{nil} \)) give rise to axiom rules (i.e., rules with no premises). Similarly, rule \( \text{Abstr}_p \) gives rise to axiom rules when it is instantiated to nullary \( p \)'s.

As usual, deduction rules are combined in trees to form proofs, and one calls theorem the conclusion of the root of some proof. The success set of a program \( \mathcal{P} \) is the set of all theorems whose antecedent is the program (i.e., theorems \( \mathcal{T} \) such that \( \mathcal{P} \vdash_a \mathcal{T} \)). Among the possible proofs of a theorem, so called uniform proofs [MNS87] are preferred for their operational reading: only apply clause rules (i.e., \( \land_{\text{clause}}, \Leftarrow_{\text{clause}}, \forall_{\text{clause}} \)) when no other rule applies. So, uniform proofs are almost completely directed by the goals, except for the selection of clauses.

**Example 3 (Sequent proof)** The following is a proof of sequent

\[
\forall X[\text{app} \, \hat{a} \, X , X] \vdash_a \text{app} \, \text{true} \, \text{false} \, \text{false}
\]
using the above deduction rules.

\[
\begin{array}{c}
\frac{[]} :_a \text{true} \quad \text{Abstr}_\text{nil} \\
\frac{\gamma \text{false} :_a \text{false}}{\text{Abstr}_\gamma} \\
\frac{\gamma \text{false} :_a \text{false}}{\text{Abstr}_\gamma}
\end{array}
\]

\[
\frac{\text{app} [] (\gamma \text{false}) (\gamma \text{false}) :_a \text{app} \text{true} \text{false} \text{false}}{\text{Abstr}_\text{app}} \quad \text{Axiom}
\]

\[
\frac{\text{app} [] (\gamma \text{false}) (\gamma \text{false}) \vdash_a \text{app} \text{true} \text{false} \text{false}}{\forall \text{clause}}
\]

Note that rule \(\forall_{\text{clause}}\) is the place in which substitutions usually enter. Instead of replacing a quantified variable by an arbitrary term, the usual PROLOG operational semantics tends to replace quantified variables by new free variables (i.e., renaming) that will be instantiated later on through unification. This amounts to choosing lazily the term by which a quantified variable is replaced, but it is unnecessarily operational for our purpose.

Note also that \(H \leftarrow B\) in the conclusion sequent of rule \(\leftarrow_{\text{clause}}\) is usually not a clause of the actual program, but a clause in which quantifications have been eliminated using rule \(\forall_{\text{clause}}\).

**Lemma 1** \(f(X_1, \ldots, X_k) :_a P_0\) if and only if \(X_1 :_a P_1, \ldots, X_k :_a P_k\) and \(\text{iff}(P_0, [P_1, \ldots, P_k])\).

**Proof** By induction on the structure of terms, and because there are never two rules with the same conclusion. \(\square\)

As a corollary, \(X_0 = f(X_1, \ldots, X_k) :_a =_a(f_0(P_0, P_1, \ldots, P_k))\) if and only if \(\text{iff}(P_0, [P_1, \ldots, P_k])\).

**Theorem 1** An abstract atom \(A\) is in the denotation of an abstract program \(P_a\) if and only if the sequent \(P \vdash_a A\) is a theorem of the above sequent calculus: i.e., \(A \in M(P_a)\) if and only if \(P \vdash_a A\).

**Proof** By correctness and completeness of the classical sequent calculus, an atom \(A\) is in the denotation of a program \(P\) if and only if the sequent \(P \vdash A\) is a theorem of the classical sequent calculus. So, \(A \in M(P_a)\) if and only if \(P_a \vdash A\). So, it remains to prove that \(P_a \vdash A\) if and only if \(P \vdash_a A\).

The classical sequent calculus and the above sequent calculus differ only in rules \(\text{Axiom}\) and \(\leftarrow_{\text{clause}}\) (the use of sequents \(\text{term} :_a \text{property}\)). Note that
the restriction on the domain of values in rule $\forall_{\text{clause}}$ does not count as a
difference, because we are comparing with classical proofs of abstract goals
in abstract programs (i.e., whose domain is already restricted).

Programs $\mathcal{P}$ and $\mathcal{P}_a$ have the same structure, so uniform proofs in both
sequent calculus will match. Thus, it remains to show that (1)

$$
\mathcal{P}_a, p_a(P_1, \ldots, P_n) \vdash p_a(P_1, \ldots, P_n)
$$

(in the classical system) if and only if

$$
\mathcal{P}, p(X_1, \ldots, X_n) \vdash p_a(P_1, \ldots, P_n)
$$

(in the above system), and (2)

$$
\begin{array}{c}
\mathcal{P}_a \vdash B_a \\
\mathcal{P}_a, H_a \vdash A
\end{array}
\quad \quad
\begin{array}{c}
\mathcal{P}_a, H_a \Leftarrow B_a \vdash A
\end{array}
$$

(in the classical system) if and only if

$$
\begin{array}{c}
B : a B_a \\
\mathcal{P} \vdash B_a \\
\mathcal{P}, H \vdash A
\end{array}
\quad \quad
\begin{array}{c}
\mathcal{P}, H \Leftarrow B \vdash_a A
\end{array}
$$

(in the above system). (1) comes from Definition 1 (rules $\text{Abstr}_f$ and $\text{Abstr}_p$)
and the corollary of Lemma 1. (2) comes by induction on the structure of
proofs ((1) is the base case), and from the definition of abstract compilation
which is mimicked by Definition 1 (rules $\text{Abstr}_f$ and $\text{Abstr}_p$).

In fact, it is as if the above abstract semantics performs just-in-time
abstract compilation (i.e., in rule $\Leftarrow_{\text{clause}}$). Note that this semantics is just
as evaluable as a program if one takes care of loops, which is possible if the
abstract domain is finite. E.g., memoing techniques can do the job [War92].

4  TYPED PROLOG

4.1  Types and terms

Types exist from the beginning in $\lambda$PROLOG but not in PROLOG, though
typed variants of PROLOG have been defined [MO84, Han89, LR91], and
some PROLOG systems have a type-checker.
We describe $\lambda$PROLOG types and a part of $\lambda$PROLOG term formation rules as a way of building a typed version of PROLOG.

Simple types [Chu40] are generated by the following grammar:

$$ $T ::= \mathcal{W} | (\mathcal{K}_i \overbrace{T \ldots T}^i) | (T \rightarrow T) $$

where $\mathcal{W}$ and $\mathcal{K}_i$ are respectively type variables and type constants of arity $i$. Types constants are sometimes called kinds, hence the $\mathcal{K}$. We suppose that $\mathcal{K}_0$ contains the constant $o$ for the type of truth values. The arrow associates to the right. The rightmost type is called the result type. Type constants are declared as follows: kind $\kappa$ type $\rightarrow \ldots \rightarrow$ type $\rightarrow$ type. The number of arrows defines the arity of the type constants.

Simply typed first-order terms are defined as follows:

$$ $\mathcal{FOT}_t ::= \mathcal{C}_t \mid \mathcal{U}_t $$

$$ $\mathcal{FOT}_t ::= (\mathcal{FOT}_v \rightarrow_t \mathcal{FOT}_v) $$

where $\mathcal{C}_t$ and $\mathcal{U}_t$ are respectively term constants and logical variables, all being of type $t$. The second rule generates applications. Application is considered left-associative. Every logical variable $X$ has a type $\tau(X) \in T$ inferred by a typing system. Every constant $f$ has a type $\tau(f) \in T$ defined by a type declaration: e.g., type $\text{nil} (\text{list} A)$. A constant is called a predicate constant if its result type is $o$: e.g., type $\text{nilp} (\text{list} A) \rightarrow o$.

The types in this definition do not differ essentially from types in the literature on PROLOG and types. It is only the external presentation that differs. For instance,_TYPED PROLOG systems usually distinguish types of predicates from types of functions as follows:

**predicate** nilp (list A).

**function** nil (list A).

with the obvious disadvantages of giving the same notation to different types, and to distinguish things that are essentially the same. The second disadvantage shows itself when dealing with higher-order; it is difficult with this presentation, but it is just natural in the presentation we have chosen.

---

4This is not the standard notation for Prolog. This Lisp-like notation has been chosen because the expression of the discipline of simple types is easier with this syntax. Indeed, only one application (i.e., one type arrow) is considered at a time.
4.2 A note on the head-condition

The inference of types of variables is decidable when types of all constants (especially predicate constants) are known [MO84]. The inference of types of predicate constants is also decidable if clauses are not subject to the head-condition (i.e., *definitional genericity*) [LR91]. According to this condition every predicate constant must occur under the same type in the head of every clause that defines it, and this type must be a renaming of the type of the predicate.

**Example 4 (The head-condition)** Let the predicates \( p, q, \) and \( r \) be

\[
p X \leftarrow p \[X\].
\]

\[
q \[X\] \leftarrow q \ X.
\]

\[
r [\ ] X \ X . \quad r [A|X] Y [A|Z] \leftarrow r X Y Z . \quad r [1 \ 2 \ [1,2]].
\]

If the declared type of \( r \) is \((\text{list } A) \rightarrow (\text{list } A) \rightarrow o\), then \( r \) is not definitionally generic because the occurrence of \( r \) in the head of its third clause has a type that is a strict instance of the declared type. If the declared type of \( r \) is \((\text{list int}) \rightarrow (\text{list int}) \rightarrow (\text{list int}) \rightarrow o\), then the head-condition is satisfied, but \( r \) is no longer polymorphic. \( p \) is definitionally generic (e.g., give type \( A \rightarrow o \) to \( p \)), but \( q \) is not. Indeed, whichever type is given to \( q \), the type of the head occurrence is either not compatible or is strictly more instantiated than the type of the body occurrence.

Under the head-condition, the inference of the types of predicates is undecidable in general, because the general semi-unification problem can be reduced to it [Tiu90]. Lakshman and Reddy propose a restriction on recursive calls to predicates that restores decidability. Their restriction resembles the ML restriction that there is no polymorphism in recursive definitions.

Whether to adopt the head-condition for **Typed Prolog** or **λProlog** is not settled definitely. The basic literature on **Prolog** gives no hint because **Prolog** is usually not typed, and the literature on **λProlog** gives no hint either. The literature on **Prolog** and types also does not help. E.g., Mycroft and O’Keefe adopt this convention silently (it is in their definition, but it is not explained [MO84]), whereas Hanus argues it is unnatural [Han89]. However, several real-size logic programming systems, e.g., Gödel [HT92, HL94],
Mercury [SHC+96], implement the head-condition. Recently, Deransart and Smaus [DS00] have proposed a semantic presentation of this property. They also have proposed a less restrictive variant of it. On the \( \lambda \text{PROLOG} \) side, Brisset and Ridoux adopt these conventions in their \( \lambda \text{PROLOG} \) compiler [BR93], though Nadathur and Pfenning propose a typing scheme that rejects the head-condition [NP92].

We believe the head-condition is a convenient property because it makes it possible to do separate compilation without redoing type checking/inference when linking separate program files. It is also a safe condition for not representing types at run-time. Note that in general, types must be represented at run-time when the head-condition is not satisfied.

Example 5 (Run-time type-checking) Consider the following program:

\[
\begin{align*}
\text{p } & \text{"1" . } \\
\text{p } & \text{[1] . } \\
\text{p } & \text{1 .}
\end{align*}
\]

query \( Y \leftarrow p \ X \land \text{append } X \ X \ Y \).

Goal \( p \ X \) must select the right clause for \( p \) according to the type of \( X \). Hence, the type of \( X \) must be represented at run-time.

Finally, there are program equivalences that do not preserve well-typing if the head-condition is not satisfied. E.g., programs \((p \ 1) \land (p \text{ "1"})\) and \( p \ X \leftarrow (X = 1 \lor X = \text{"1"})\) are logically equivalent in the untyped setting, but the first program is well-typed if and only if the head-condition is rejected, whereas the second program is always ill-typed. Note that transforming a program \((p \ 1) \land (p \text{ "1"})\) into a program \( p \ X \leftarrow (X = 1 \lor X = \text{"1"})\) forms the first step of the Clark completion of a program [Cla78].

In this article, we assume that all types are available, either via inference or via declaration, and we also assume that definitional genericity is enforced; this will help in proving the termination of the analysis (see Section 6).

4.3 A note on rational terms

In the following, we consider only well-typed programs with the usual PROLOG semantics. Though most PROLOG systems do not perform the occurrence-check, we will not consider that the computation domain is made of rational terms. This is because most PROLOG systems do not really implement them;
they let programs build rational terms, but usually loop or abort when unifying or printing them, and very often rational terms do not belong to the source syntax. Note that typing prevents the formation of rational terms with infinite type; e.g., $X = [X]$ is ill-typed (it is essential that type checking performs the occurrence-check). Since our proposal consists essentially in computing in the domain of the types of a program that is never executed, we believe that it applies as well to programs that compute rational terms. However, checking this conjecture deserves some more works that we do not have done yet. So, we restrict the application of our method to TYPED PROLOG programs that are not-subject-to-occur-check [DFT91].

5 Typed static analysis of the groundness property

5.1 Typed $\mathcal{P}os$

We combine $\mathcal{P}os$ with simple types in the static analysis of TYPED PROLOG programs. We call this new domain $\mathcal{P}os_T$.

Firstly, we build an abstract domain for terms that combines $\{true, false\}$ with types. We call this domain $\mathcal{B}ool_T$. The constants of this domain are built with values $true$ and $false$ and with type constants (defined in the program). We define an equivalence relation, $=_{true}$, on $\mathcal{B}ool_T$.

Definition 2 (Domain $\mathcal{B}ool_T$)

$true \in \mathcal{B}ool_T,$

$false \in \mathcal{B}ool_T,$

and for every type constant $\kappa \in \mathcal{K}_i$, if all $x_j \in \mathcal{B}ool_T$, $(\kappa_a x_1 \ldots x_i) \in \mathcal{B}ool_T.$

(Equivalence relation, $=_{true}$)

$(\kappa_a true \ldots true) =_{true} true.$

$(\kappa_a P_1 \ldots P_n) =_{true} (\kappa_a Q_1 \ldots Q_n)$ if and only if $P_i =_{true} Q_i$ for every $i$.

We will use the first axiom oriented left-to-right as a rewriting rule, $\rightarrow_{true}$, to compute normal-forms.
For every subset $S$ of $\text{Bool}_T$ we write $S / =_{\text{true}}$ the set of equivalence classes of $S$ modulo $S / =_{\text{true}}$.

For every $t$, we write $\parallel t \parallel_{\text{true}}$ the equivalence class that contains $t$. By convention, we represent equivalence classes by their normal-form element with respect to rule $\rightarrow_{\text{true}}$.

In the simply typed variant of Typed Prolog, $\text{Bool}_T$ can be declared as follows:

- $\text{kind Bool}_T$ type .
- $\text{type} (\text{true, false}) \text{Bool}_T$ . % Basic values
- $\text{type} \kappa_a \underbrace{\text{Bool}_T \rightarrow \ldots \rightarrow \text{Bool}_T}_n$ . % $\kappa$ is a type constant of arity $n$

**Example 6 (Abstract values)** If there is a declaration

- $\text{kind} \text{pair type} \rightarrow \text{type} \rightarrow \text{type}$

in the program, values $(\text{pair}_a \text{true true})$, $(\text{pair}_a \text{true false})$, etc, are in $\text{Bool}_T$. Likewise, if a type constant list is declared, values $(\text{list}_a \text{false})$, $(\text{list}_a (\text{list}_a \text{false}))$, etc, are in $\text{Bool}_T$. And if both pair and list are declared, then $(\text{list}_a (\text{pair}_a \text{true false}))$ and $(\text{pair}_a \text{true (list}_a \text{false}))$, etc, are in $\text{Bool}_T$ as well.

Given a type $\tau$ (possibly containing variables), we write $\tau_a$ the term of $\text{Bool}_T$ obtained by replacing in $\tau$ every $n$-ary ($n \geq 1$) type constant $\kappa$ with $\kappa_a$, and every nullary type constant with $\text{true}$.

We have supposed that programs are type-checked before analyzing them. So, every symbol in a program, constant or variable symbol, is given a type $\tau$ which may contain type variables as traces of polymorphism. In fact, not all elements of $\text{Bool}_T$ can be the abstraction of a term of type $\tau$. E.g., a term of type $(\text{list} A)$ cannot be abstracted in $(\text{pair}_a \text{false false})$. This is important because binary operations and relations that are to be defined on $\text{Bool}_T$ are only used between abstract values that correspond to the same type. We call these values *compatible abstract instances*. We write $\text{Bool}_T(\tau)$ the maximal subset of $\text{Bool}_T$ that is compatible with $\tau$.

**Definition 3 (Abstract instances of a type)**

$$\tau(\text{Bool}_T) = \{ p \in \text{Bool}_T \mid p \text{ is an instance of } \tau_a \}$$
(Partial order relation, <false)

\[ \text{false} <_{\text{false}} (\kappa_a P_1 \ldots P_n) \text{ for all } \kappa_a, P_1, \ldots, P_n. \]
\[ (\kappa_a P_1 \ldots P_n) <_{\text{false}} (\kappa_a Q_1 \ldots Q_n) \text{ if and only if } P_i <_{\text{false}} Q_i \text{ for at least one } i, \text{ and } P_i \leq_{\text{false}} Q_i \text{ for others (product order).} \]

Relation <false is extended naturally to equivalence classes modulo =true:

\[ \|t\|_{\text{true}} <_{\text{false}} \|t'\|_{\text{true}} \text{ if } t <_{\text{false}} t'. \]

(Compatible abstract instances)

\[ \text{Bool}_T(\tau) = \{ p \in \text{Bool}_T \mid \exists p' \in \tau(\text{Bool}_T)[p \leq_{\text{false}} p'] \} \]

Example 7 (Abstract instances) If pairs and lists are defined as in Example 6, (list (pair A A))(Bool_T) is the set

\[ \{ (\text{list}_a (\text{pair}_a \text{false false})), (\text{list}_a (\text{pair}_a (\text{list}_a \text{false})) (\text{list}_a \text{false})), \ldots, (\text{list}_a (\text{pair}_a (\text{pair}_a \text{false} (\text{list}_a \text{false})) (\text{pair}_a \text{false} (\text{list}_a \text{false}))), \ldots, \text{true} \} \]

and Bool_T((list (pair A A))) is the set

\[ \{ \text{false}, (\text{list}_a \text{false}), (\text{list}_a (\text{pair}_a \text{false false})), (\text{list}_a (\text{pair}_a \text{false} (\text{list}_a \text{false}))), (\text{list}_a (\text{pair}_a \text{false} (\text{pair}_a \text{false}))), \ldots, (\text{list}_a (\text{pair}_a (\text{pair}_a \text{false} (\text{pair}_a \text{false}))) (\text{pair}_a \text{false} (\text{list}_a \text{false})))), \ldots, (\text{list}_a (\text{pair}_a (\text{pair}_a \text{false} (\text{list}_a \text{false}))) (\text{pair}_a \text{false} (\text{list}_a \text{false}))), \ldots, \text{true} \}. \]

Note that \[ \|\text{false}\|_{\text{true}} <_{\text{false}} \|\text{true}\|_{\text{true}}, \text{ though } \text{false} \not<_{\text{false}} \text{true}. \] This is because \[ \text{false} <_{\text{false}} (\text{list}_a \text{true}) =_{\text{true}} \text{true}. \]

For every type \( \tau \), (Bool_T(\tau)/=true, <false) forms a lattice whose least value is \[ \|\text{false}\|_{\text{true}} \] and greatest value is \[ \|\text{true}\|_{\text{true}}. \]

Definition 4 (Lattice operations in Bool_T(\tau)/=true)

Let \( v_1 \) and \( v_2 \) in Bool_T(\tau)/=true.

\( v_1 \cap v_2 \) is the greatest \( v \) such that \( v \leq_{\text{false}} v_1 \) \( \land \) \( v \leq_{\text{false}} v_2 \).

\( v_1 \cup v_2 \) is the least \( v \) such that \( v_1 \leq_{\text{false}} v \) \( \land \) \( v_2 \leq_{\text{false}} v \).
In particular, when applied to compatible abstract values operation \( \sqcap \) never yields \( false \) unless one of its arguments is already \( false \). This property is reminiscent of the positiveness of \( Pos \).

Every time there is a type constant of arity greater than 0, \( \langle \mathsf{Bool}_T(\tau) / \mathsf{=} \mathsf{true}, <_\mathsf{false} \rangle \) is a lattice with an infinite height. We will see in section 6 that it does not impede termination. More precisely, there are infinite increasing chains in \( \langle \mathsf{Bool}_T(\tau) / \mathsf{=} \mathsf{true}, <_\mathsf{false} \rangle \): e.g.,

\[
false, \ (\text{list} \mathsf{a} \ false), \ (\text{list}_a (\text{list} \mathsf{a} \ false)), \ ... \ (\text{list}_a^n \ false), ...
\]

and also infinite decreasing chains: e.g.,

\[
true, \ (\text{pair} \mathsf{a} \ false \ true), \ (\text{pair}_a \ false \ (\text{pair} \mathsf{a} \ false \ true)), \ ... \ ((\text{pair}_a \ false)^n \ true), ...
\]

but no such infinite decreasing chains exists in \( \langle \mathsf{Bool}_T(\tau), <_\mathsf{false} \rangle \).

**Lemma 2 (Decreasing chains)** All decreasing chains of \( \mathsf{Bool}_T \) in order \( <_\mathsf{false} \) are finite.

**Proof** The axiom of relation \( <_\mathsf{false} \) shows that the left member has at least one constructor less than the right member. There is only a finite number of such constructors in any value. Hence, decreasing chains are finite. \( \square \)

The problem with order \( <_\mathsf{false} \) in \( \mathsf{Bool}_T / \mathsf{=} \mathsf{true} \) is that occurrences of \( true \) are equivalent to compound values with arbitrary many constructors. In fact, decreasing chains of \( \mathsf{Bool}_T / \mathsf{=} \mathsf{true} \) in order \( <_\mathsf{false} \) are finite if their starting element does not have any occurrences of \( true \).

So, termination of static analysis will come from a cautious usage of relation \( =_\mathsf{true} \). Essentially, it must be used only for normalizing results.

Predicate \( \mathsf{iff} \) must be redefined to take into account the domain \( \mathsf{Bool}_T \).

**Definition 5** \( \mathsf{iff}_T \ X_0 [X_1, \ldots, X_k] \) is defined by \( X_0 = \bigcap_{i \in [1,k]} X_i \)

Now, we can define the abstract domain for predicate, that is the typed counterpart of \( Pos \).
Definition 6 (Domain $\mathcal{P}_{\text{os}_T}$) A predicate $p$ with type $\tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow o$ is abstracted as a function $p_a$ with type $\text{Bool}_T(\tau_1) \rightarrow \ldots \rightarrow \text{Bool}_T(\tau_n) \rightarrow \text{Bool}$.

In the same way as boolean formulas can be represented as finite truth-tables, abstract predicate functions can be represented as infinite decision-tables. We consider only functions that can be built using $\text{iff}_T$, join, and union.

(Union of functions) The union, $\sqcup$, of two functions $f$ and $g$ of same type on $\text{Bool}$, is defined as follows:
$$(f \sqcup g)(x) = f(x) \lor g(x).$$

(Join of functions) The join, $\sqcap$, of two functions $f$ and $g$ of variables $X_1, \ldots, X_n, Y_1, \ldots, Y_p$ and $X_1, \ldots, X_n, Z_1, \ldots, Z_q$ is defined as follows:
$$(f \sqcap g)(X,Y,Z) = f(X,Y) \land g(X,Z).$$

5.2 First-order typed groundness analysis

We present the analysis of the typed groundness property for Typed Prolog. Let us first study the case of lists in an exploratory phase.

The data type $\text{list}$ is defined as follows:

```
kind list type → type .
type [] (list _).
type '!' A → (list A) → (list A).
```

These declarations represent the following type deduction rules:

$\frac{}{[]} : (\text{list } \_)}$

$\frac{X_1 : A \quad X_2 : (\text{list } A)}{[X_1 | X_2] : (\text{list } A)}$

Note that proofs are built by assembling instances of such deduction rules. The multiple occurrences of the same variable in a deduction rule express that several things must be identical. E.g., in the above rule, the type of $X_1$, and the type of the elements of $X_2$ must be identical. We call this a coreference constraint.

Now, we combine typing rules with deduction rules for groundness (see Section 3.3) by relaxing type constraints in premises of typing rules, and propagating the relaxation in conclusions. The first kind of relaxation is to
replace multiple occurrences of the same variable with as many occurrences of different variables: this relaxes coreference constraints. The second kind of relaxation is to replace compound type expressions by lower types in the \(<\text{false}\) order.

\[
\begin{array}{c}
\text{[]} : \text{a} (\text{list}\text{a} \text{true}) \\
X_1 : \text{a} A_1 \\
X_2 : \text{a} (\text{list}\text{a} A_2) \\
\text{[X_1|X_2]} : \text{a} (\text{list}\text{a} A_1 \sqcap A_2)
\end{array}
\]

\[
\begin{array}{c}
X_1 : \text{a} P_1 \\
X_2 : \text{a} P_2 \\
\text{[X_1|X_2]} : \text{a} \text{false}
\end{array}
\quad \text{if } \forall P \in \text{Bool}_T[P_1 <_{\text{false}} P] \\
\quad \text{or } \forall P \in \text{Bool}_T[P_2 <_{\text{false}} (\text{list}\text{a} P)]
\]

Note that \(P_1 <_{\text{false}} P\) implies \(P_1 \in \text{Bool}_T(P)\), and \(P_2 <_{\text{false}} (\text{list}\text{a} P)\) implies \(P_2 \in \text{Bool}_T((\text{list}\text{a} P))\) (see Definition 3). This generalizes in that the abstract value corresponding to a type \(\tau\) is in \(\text{Bool}_T(\tau)\).

Since \(\text{false}\) is the least value of \(\text{Bool}_T\), we can replace the last rule with:

\[
\begin{array}{c}
X_1 : \text{a} P_1 \\
X_2 : \text{a} P_2 \\
\text{[X_1|X_2]} : \text{a} \text{false}
\quad \text{if } P_1 <_{\text{false}} \text{false} \lor P_2 <_{\text{false}} (\text{list}\text{a} \text{false})
\end{array}
\]

Constraint \(P_1 <_{\text{false}} \text{false}\) cannot be satisfied, and the second one simply says \(P_2 = \text{false}\). However, in the sequel, we prefer to use forms similar to \(P_1 <_{\text{false}} \text{false}\) and \(P_2 <_{\text{false}} (\text{list}\text{a} \text{false})\) for the sake of generalization.

This rule says that the only way a \(\text{cons}\) can have property \(\text{false}\) is that its \(\text{cdr}\) has also property \(\text{false}\). Note the parallel with the informal definition of \(\text{partial}\) in Example 1.

We define new predicates for coding the preceding propagation rules:

\[
\begin{array}{c}
\text{type} \iff \text{list}\text{nil} \text{Bool}_T \to (\text{list} \text{Bool}_T) \to o. \\
\quad \iff \text{list}\text{nil} (\text{list}\text{a} V) [] \Leftarrow \iff V \[].
\end{array}
\]

\[
\begin{array}{c}
\text{type} \iff \text{list}\text{cons} \text{Bool}_T \to (\text{list} \text{Bool}_T) \to o. \\
\quad \iff \text{list}\text{cons} (\text{list}\text{a} X) [A_1, (\text{list}\text{a} A_2)] \Leftarrow \iff X [A_1, A_2]. \\
\quad \iff \text{list}\text{cons} \text{false} [\_ P_2] \Leftarrow P_2 <_{\text{false}} (\text{list}\text{a} \text{false}).
\end{array}
\]

We can easily generalize these deduction rules to all the first-order constants, \textbf{type} \(c \tau_1 \to \ldots \to \tau_k \to (\kappa T_1 \ldots T_n)\), where \(\tau_1, \ldots, \tau_k\) and \((\kappa T_1 \ldots T_n)\) are either basic types \((\text{int}, \text{float}, \text{string}, \text{etc})\), or compound types \((\text{list} A), (\text{pair} A B), \text{etc})\). Every declaration of a term constant corresponds to three deduction rules: the usual typing rule, a static analysis rule that propagates groundness, and a static analysis rule that propagates non-groundness.
Definition 7 (Deduction rules)

Typing

\[
\begin{align*}
X_1 : \tau_1 & \quad \cdots \quad X_k : \tau_k \\
(c \ X_1 \ \cdots \ X_k) : (\kappa \ T_1 \ \cdots \ T_n)
\end{align*}
\]

Groundness

\[
\begin{align*}
X_1 : a \ \tau_1' & \quad \cdots \quad X_k : a \ \tau_k' \\
(c \ X_1 \ \cdots \ X_k) : a (\kappa_a \ (\cap R_1) \ \cdots \ (\cap R_n))
\end{align*}
\]

where \(\tau_j'\) (\(j\) from 1 to \(k\)) are copies of types \(\tau_j\) where all occurrences of all variables are renamed apart. \(R_i\) is a list of all the renaming variables of \(T_i\) in the \(\tau_j'\)'s (\(j\) from 1 to \(k\)). Hence, \((\cap R_i)\) is the greatest lower bound of the renaming variables of a given variable \(T_i\).

In the example of lists, the list of renaming variables of \(A\) is \([A_1, A_2]\).

Non-groundness

\[
\begin{align*}
X_1 : a \ P_1 & \quad \cdots \quad X_k : a \ P_k \\
(c \ X_1 \ \cdots \ X_k) : a \ \text{false} & \quad \text{if} \ \bigvee_{i \in [1,k]} (P_i <_{\text{false}} \tau_i'')
\end{align*}
\]

where \(P_i\) are new variables, one per argument of \(c\), and the \(\tau_i''\)'s are instances of \(\tau_j\) where all variables are replaced with false.

The \(\tau_i''\)'s are minimal properties that arguments to a term constructor must satisfy so that the whole term is a proper term. As soon as one of the arguments does not satisfy the property, the whole term is deemed partial.

The general scheme can produce unsatisfiable constraints \(P_i <_{\text{false}} \text{false}\); and this allows in practice to forget rules if none of their premises can be satisfied.

Abstraction as sequent calculus The complete system of deduction rules for the typed analysis of groundness for Typed Prolog is made of the above rules plus \(\land_{\text{goal}}, \land_{\text{clause}}, \forall_{\text{clause}}, \leq_{\text{clause}}, \text{Axiom}, \text{and Abstr}_{\text{goal}}\) (see Definition 1).

Note that concrete programs are supposed to be well-typed, so all occurrences of a constant have types which are instances of its type scheme. So, properties are always compatible with the type of the term they are attached to (see Definition 3). So, the reason why a constraint like \(P_i <_{\text{false}} \tau_i''\) fails cannot be that \(P_i\) is not comparable with \(\tau_i''\). The only reason \(P_i <_{\text{false}} \tau_i''\) can fail is that \(P_i \geq_{\text{false}} \tau_i''\). So, in this occurrence, we have \(\neg(P_i <_{\text{false}} \tau_i'') \Rightarrow P_i \geq_{\text{false}} \tau_i''\), even if \(<_{\text{false}}\) is only a partial order.
Theorem 2 Let \( \mathcal{P} \) be a program and \( A \) an abstract atom, then \( \mathcal{P} \vdash_{a} A \) if and only if there is a goal \( G \) such that \( \mathcal{P} \vdash G \) and \( G :_{a} A \).

**Proof** By induction on the height of sequent proofs.

- The base case is the application of rule *Axiom*.
  - (if) From \( \mathcal{P} \vdash G \) we have \( G \in \mathcal{P} \).
    Hence, from \( G :_{a} A \) and rule *Axiom* we have \( \mathcal{P} \vdash_{a} A \).
  - (only if) From \( \mathcal{P} \vdash_{a} A \) we have \( G \in \mathcal{P} \) for some \( G \) and \( G :_{a} A \) because rule *Axiom* is the only one that can prove a sequent \( \mathcal{P} \vdash_{a} A \).
- Induction: by inspection of all rules. \( \Box \)

5.3 Typed abstract compilation of groundness

Abstraction procedures for the abstract compilation of declarations of type and term constants are as follows:

**Definition 8**

*(Abstraction of type constants)* For each type constant \( \kappa \):

\[
\text{Abstr}_{\text{kind}}[\text{kind } \kappa \text{ type } \rightarrow \ldots \text{ type}] \equiv \text{type } \kappa_{a} \text{ Bool }_{T} \rightarrow \ldots \text{ Bool }_{T}.
\]

*(Abstraction of term constants)* For each term constant \( c \):

\[
\text{Abstr}_{\text{type}}[\text{type } c \tau_{1} \rightarrow \ldots \rightarrow \tau_{k} \rightarrow (\kappa \ T_{1} \ldots T_{n})] \equiv \text{type } \text{iff}_{\kappa_{a} c} \text{ Bool }_{T} \rightarrow (\text{list } \text{Bool }_{T}) \rightarrow o.
\]

\[
\text{iff}_{\kappa_{a} c} (\kappa_{a} T_{1} \ldots T_{n}) [\tau_{1}', \ldots, \tau_{k}'] \Leftarrow \text{iff } T_{1} \ R_{1} \land \ldots \land \text{iff } T_{n} \ R_{n}.
\]

\[
\text{iff}_{\kappa_{a} c} \text{ false } [P_{1}, \ldots, P_{k}] \Leftarrow P_{1} \ <_{\text{false } \tau_{1}''} \lor \ldots \lor P_{k} <_{\text{false } \tau_{k}''}.
\]

It is easy to see how a deduction rule

\[
\begin{array}{ll}
\text{Premisse} & \\
\text{Conclusion} & \\
\end{array}
\]
can be written as a Horn clause Conclusion $\iff$ Premisse. The first clause of predicate $\text{iff}_{\kappa \cdot c}$ implements rule $\text{Abstr}_{\text{term}^+}$, whereas the second clause implements rule $\text{Abstr}_{\text{term}^-}$ (see Definition 7).

The following example shows one application of this method to a more complex type (trees with varisized nodes).

**Example 8 (Abstraction of constants)** Concrete declarations:

- **kind** $\text{ntree type}$ $\rightarrow$ $\text{type}$ $\rightarrow$ $\text{type}$.
- **type** $\text{leaf} L \rightarrow (\text{ntree} N L)$.
- **type** $\text{node} (\text{list} (\text{ntree} N L)) \rightarrow N \rightarrow (\text{ntree} N L)$.

Their abstractions (where unsatisfiable premises are underlined, and correspond to useless clauses):

- $\text{iff}_{\text{ntree} \cdot \text{leaf}} (\text{ntree} a N_0 L_0) [L_1] \iff L_0 [L_1]$.
- $\text{iff}_{\text{ntree} \cdot \text{leaf}} false [P_1] \iff P_1 <false \cdot false$.

- $\text{iff}_{\text{ntree} \cdot \text{node}} (\text{ntree} a N_0 L_0) [(\text{list}_a (\text{ntree} N_1 L_1)), N_2] \iff L_0 [L_1] \land N_0 [N_1, N_2]$.
- $\text{iff}_{\text{ntree} \cdot \text{node}} false [P_1, P_2] \iff (P_1 <false (\text{list}_a (\text{ntree} false false)) \lor P_2 <false false)$.

**Lemma 3** $f(X_1, \ldots, X_k):_a P_0$ if and only if $\text{iff}_T (P_0, [P_1, \ldots, P_k])$ and $X_1 :_a P_1, \ldots, X_k :_a P_k$.

**Proof** By induction on the structure of terms. $\square$

The abstraction procedure for first-order terms is given in the following definition.

**Definition 9 (Abstraction of equalities with term constants)**

$$\text{Abstr}_{\text{goal}}[F = (c \cdot X_1 \ldots X_k)] \equiv \text{iff}_{\kappa \cdot c} F [X_1, \ldots, X_k]$$

where $c$ is a term constant of result type $(\kappa T_1 \ldots T_n)$.

**Definition 9 (Abstraction of equalities without term constants)**

$$\text{Abstr}_{\text{goal}}[F = X] \equiv \text{iff}_T F [X]$$

where $X$ is a variable.

**Definition 9 (Abstraction of predicate calls)**

$$\text{Abstr}_{\text{goal}}[(p \cdot X_1 \ldots X_k)] \equiv (p_a X_1 \ldots X_k).$$
Again, deduction rules can be written as Horn clauses. The definition of \( \text{Abstr}_{\text{goal}} \) must be completed to handle connectives, and an \( \text{Abstr}_{\text{clause}} \) procedure must also be defined. Their definitions are straightforward, and correspond to abstract deduction rules of Section 1.

**Definition 10 (Abstraction of conjunctions in goals)**

\[
\text{Abstr}_{\text{goal}}[G_1 \land G_2] \equiv \text{Abstr}_{\text{goal}}[G_1] \land \text{Abstr}_{\text{goal}}[G_2].
\]

**Abstraction of universal quantifications in clauses**

\[
\text{Abstr}_{\text{clause}}[\forall X[C]] \equiv \forall X[\text{Abstr}_{\text{clause}}[C]].
\]

**Abstraction of implications in clauses**

\[
\text{Abstr}_{\text{clause}}[H \iff B] \equiv \text{Abstr}_{\text{goal}}[H] \iff \text{Abstr}_{\text{goal}}[B].
\]

**Abstraction of conjunctions in clauses**

\[
\text{Abstr}_{\text{clause}}[C_1 \land C_2] \equiv \text{Abstr}_{\text{clause}}[C_1] \land \text{Abstr}_{\text{clause}}[C_2].
\]

Let us apply this method to program \( \text{append} \) (see definition and normal form in Example 2). Since abstraction is only applied on normalized programs, the only possible forms for atomic formulas using list constants are \( L = \square \) and \( L = [A|Ls] \).

**Example 9 (Abstraction of append)** The abstraction of program \( \text{append} \) gives the following result:

\[
\text{type} \ \text{append}_{a} \ 	ext{Bool}_{T} \rightarrow 	ext{Bool}_{T} \rightarrow 	ext{Bool}_{T} \rightarrow o.
\]

\[
\text{append}_{a} \ Xs \ Ys \ Zs \iff \text{list\_nil} \ Xs \square \land \text{iff}_{T} \ Ys \ Zs.
\]

\[
\text{append}_{a} \ Xs \ Ys \ Zs \iff
\text{iff}_{\text{list\_cons}} \ Xs \ [X|Xs1] \land \text{iff}_{\text{list\_cons}} \ Zs \ [X|Zs1] \land
\text{append}_{a} \ Xs1 \ Ys \ Zs1.
\]

The set of success patterns of program \( \text{append}_{a}/3 \) is as follows:

\[
\text{append}_{a} \ 	ext{true \ true \ true},
\text{append}_{a} \ (\text{list}_{a} \ \text{false}) \ (\text{list}_{a} \ \text{false}) \ (\text{list}_{a} \ \text{false}),
\text{append}_{a} \ \text{true} \ (\text{list}_{a} \ \text{false}) \ (\text{list}_{a} \ \text{false}),
\text{append}_{a} \ (\text{list}_{a} \ \text{false}) \ \text{true} \ (\text{list}_{a} \ \text{false}),
\text{append}_{a} \ (\text{list}_{a} \ \text{false}) \ \text{false} \ \text{false},
\text{append}_{a} \ \text{true} \ \text{false} \ \text{false}.
\]

29
Note that this set does not contain atoms like (append\(_a\) false \ldots \ldots). This is because predicate append/3 either goes through or constructs an entire list in its first argument. Hence, the first argument of every answer to append/3 is a proper list.

This example shows that typed analysis is not merely a combination of type inference/checking and of static analysis in Pos. For instance, an argument can be a list, and still have property false (e.g., second and third arguments of append/3); to have property (list\(_a\) |textitsomething), an argument must be a proper list.

Note that if one substitutes true for properties (list\(_a\) something) (i.e., not false), one is left with a purely boolean truth-table, hence a purely boolean formula. In the case of append/3, the boolean formula is \(X_1 \land (X_2 \Leftrightarrow X_3)\). Similarly, if one substitutes false for all non-true properties (e.g., (list\(_a\) false)), one is left with another purely boolean formula. In the case of append/3, it is \((X_1 \land X_2) \Leftrightarrow X_3\), which expresses the groundness property. In fact, the property expressed by the first formula is the properness property.

Furthermore, the second and third arguments of append/3 have the same type ((list\(_a\) A)), and are equivalent with respect to the groundness property \((X_2 \Leftrightarrow X_3)\). In particular, typing and groundness are invariant with respect to exchanging the second and third arguments. However, properness is not invariant for this operation. This shows again that typed analysis is more than merging the results of a type analysis and of an untyped groundness analysis.

6 Termination

The lattice \((\text{Bool}_T(\tau) / =_{\text{true}, \textless_{\text{false}}} \langle, \textless_{\text{false}} \rangle)\) has an infinite height in all non-degenerated cases (see Section 5.1). In this section, we show that it does not impede the termination of the analysis.

The reason comes from the hypothesis that programs are type-checked before being analyzed, that they follow the discipline of simple types and the polymorphism discipline of definitional genericity [LR91] (related to the so-called head-condition [HT92]), and from the observation that all decreasing chains of \(\text{Bool}_T\) in order \(<_{\text{false}}\) are finite (see Lemma 2).

The first hypothesis essentially implies that a type is associated to every component (constants or variables) of a clause. The second hypothesis implies that all occurrences in clause heads of a predicate constant
have equivalent types, and that all other occurrences (i.e., in clause bodies) have types that are either equivalent to or more instantiated than the types of head occurrences. This condition implies among other things that no type information goes from a called predicate to a calling goal. We are not concerned here with means for enforcing/checking definitional genericity [MO84, Han89, LR91, HT92]. However, we can say that practical logic programming systems exist that satisfy these hypotheses, e.g., Gödel, Mercury, and λPROLOG [HL94, SHC+96, MN86].

Thus, every component of a clause is equipped with a simple type that may contain type variables as a trace of polymorphic declarations. The head-condition insures that the types of most-general answers, like those computed by the S-semantics [FLMP89] for PROLOG, are not more instantiated than the type of their predicate. In other words, such a semantics applied to well-typed programs does not compute any new type.

Finally, observe that conclusions of typed analysis rules are either true, or lower than or equal (in the \( <_{\text{false}} \) order of Definition 2) to conclusions of the type rules from which they are derived. The typed analysis rules being applied to the same terms as the typing rules, they associate to terms properties that are either true, or are lower than or equal to their types. So, the answers computed by the S-semantics applied to the abstract program are on chains descending from the types of the clause heads. So, they can be searched among finite subsets of \( \text{Bool}_T \). So, only bounded-size decision-table must be considered for functions in \( \text{Pos}_T \).

So, the effective domain of typed analysis is the set of type instances obtained by substituting true or false to type variables in types of all program components, plus all lower properties in \( \text{Bool}_T \). It is a finite set, that could be represented as a set of constants, though this would not be suitable for comparing properties with respect to ordering \( <_{\text{false}} \). Then, the domain of abstract programs can be assimilated with DATALOG.

7 Related works

We know of several other works with similar-looking objects: Codish and Demoen’s, and Codish and Lagoon’s type dependencies [CD94, CL96], Smaus, Hill and King’s typed analysis [SHK00], and Bossi et al.’s semi-linear norms [BCF91].

The first work and ours combine types and \( \text{Pos} \), but the main difference is that the first work adopts the descriptive view of types, whereas we
stick to the prescriptive view. In Codish and Demoen’s work types are not known \textit{a priori} but are inferred. In fact, they are the objectives of the inference, whereas our objective is to infer typed properties knowing types \textit{a priori}. On a more technical side, the prescriptive view plus the choice of the head-condition gives us the rational for termination even if the abstract domain seems infinite. In their works, Codish and Demoen gain finiteness by bounding the nesting of types in abstract expression (2 in the examples of their article). Moreover, the prescriptive point of view gives us a rational for inventing typed properties systematically, though Codish and Demoen have to invent every type of interest by hand. Finally, the description of their implementation is much more advanced than ours. To give an idea of some subtle difference between this works and ours, consider what it infers for a polymorphic \texttt{append/3} predicate,

\begin{align*}
\text{append}(\text{list}(\text{bot}), A, A) . \\
\text{append}(\text{list}(A), \text{list}(B), \text{list}(C)) &\iff \text{least-upper-bound}(A, B, C) . \\
\text{append}(\text{list}(\_), B, \text{any}) &\iff \text{non_equal}(B, \text{list}(\_)) .
\end{align*}

and compare with the result in Example 9. Our 4 first lines correspond to their second line; our first and last lines correspond to their first line; but our fifth line and their third line have no correspondent. This is due in particular to the fact that considering prescriptive types \textit{a priori} forces $B$ to be a list. In fact the readings of expressions like \texttt{list(something)} are not the same in the two works. Note however, that our work and Codish and Demoen’s are very parallel, and differ mainly in point of views. Our work considers types as prescriptive, and polymorphism as the hook for attaching static properties to types.

Codish and Lagoon keep the descriptive point of view, but change the domain. The domain is essentially made of sets of types. Set construction is represented by an associative, commutative, and idempotent operator, $\oplus$, which introduces the need for ACI-unification. E.g., a list $[X]$ is abstracted in $\text{list}(A) \oplus \text{nil}$, a list $[1|L]$ in $\text{list}(\text{int}) \oplus L$, a list $[X, Y]$ in $\text{list}(X) \oplus \text{list}(Y) \oplus \text{nil}$, and a list $[1, "1"]$ in $\text{list}(\text{int}) \oplus \text{list}(\text{string}) \oplus \text{nil}$. The last example is ill-typed from the prescriptive view, and would not be treated in our method. The third example ($[X, Y]$) must not make one believe that there are as many \texttt{list(...)} in the abstraction of a list as there are elements in the list. In fact, ACI-unification will recognize (thanks to the idempotency axiom) when $X$ and $Y$ are identical. However, if it is actually the case that the types of
all the elements of a list are different then termination is endangered. The following program constructs such a list:

\[
\begin{align*}
p & : \mathbb{N} \\
p \in \mathbb{N} & \mid L \\
p E \mid [E] \mid L & \iff p \mid [E] \mid L 
\end{align*}
\]

So, Codish and Lagoon propose to bound the depth of the computation for polymorphic types. In our proposal, the head-condition insures that this cannot happen, so that termination is guaranteed even for polymorphic types. Note also that Codish and Lagoon’s proposal is restricted to unary type constants; e.g., it does not handle our ntree example (see Example 8 in Section 5.3).

Smaus, Hill, and King propose a framework for typed static analysis, but they do not try to build a Pos-like abstract domain for their analysis. So, they write \texttt{list(Any, Open)} for partial lists (false in our notation), and \texttt{list(Any, Ter)} for proper lists ((\texttt{list Any}) in our notation), but they do not try to identify \texttt{true} and \texttt{list(int(Ter), Ter)}.

The relation with Bossi et al.’s work is different: the inferred property is the same (called rigidity in their work) but the formal means is different. Bossi et al. use term metrics, and recognize terms whose measure is invariant under substitution; e.g., the length of rigid (proper) list is invariant under substitution. We believe that our approach to analyzing rigidity is simpler than in this work, but one must notice that in this work the measure itself (beside its invariance) is useful for proving termination, which is the main objective of the work.

8 Conclusion and further works

In this article, we refine the information obtained with groundness analysis by combining the types of terms with the expression of the analyzed property.

As types are only a special kind of properties, it may seem vain to separate type inference and static analysis. However, a clear difference arises when one considers prescriptive types (which are not consequences of a program semantics), and semantic properties. This is what we are doing in supposing that polymorphic types are given, either via declarations or via inference, and we use them to build a property domain that is more refined than Pos. It is also important for the termination of evaluation of the abstract program that definitional genericity is enforced (see Section 6).
To summarize, let us say that groundness analysis has to do with ground-
ness, type inference/checking has to do with well-typing, and typed ground-
ness analysis has to do with typed terms forming proper structures [O’K90].
Typed groundness analysis meets plain groundness analysis when a term and
all its subterms are proper structures, but being a proper structure is to be
“ground enough” for many purpose, e.g., compile-time optimization.

The result of abstracting a Typed Prolog program with respect to the
typed groundness property is a Datalog program in which a finite set of
constants is represented by complex terms. This language is decidable, so
it makes sense to use it as the target of an abstract compilation method.
Indeed, our typed propagation rules infer only properties which are either
true, or lower than types given by inference and verification rules of type in
Typed Prolog. In fact, the inferred/verified types form an envelop for the
properties which can be reached in typed static analysis.

The present work can be continued on different ways. First, one can
apply it to other properties of Prolog: e.g., sharing [JL89] (see [CSS99]
for the relation between Pos and sharing), finiteness [BDM92], and func-
tional dependency [ZD90]. E.g., a list whose elements share a subterm could
be distinguished from a list that shares with a sublist. These experiments
could lead to a generic tool to combine non-typed properties with types. We
think that the presentation with deduction rules is a very convenient way for
studying this generic combination. The second direction is to implement the
method entirely. At the present time, a prototype implements the normal-
ization and the abstraction for Typed Prolog, and the evaluation of the
resulting abstract program.

References

Two classes of boolean functions for dependency analysis. Science

[Bar91] H. Barendregt. Introduction to generalized type systems. J. Func-

[BCF91] A. Bossi, N. Cocco, and M. Fabis. Proving termination of logic
programs by exploiting term properties. In S. Abramsky and


