Type inference in Prolog: a new approach

M.-M. Corsini and K. Musumbu

LaBRI, URA 1304 du C.N.R.S., Université de Bordeaux I, 351 cours de la Libération, 33405 Talence Cedex, France

Abstract


This paper presents a new approach to type inference of Prolog programs. The novelty is in the fact that we only require the existence of a type domain $\mathcal{T}$ with a few primitive operations such as the abstract unification of elements of $\mathcal{T}$, and operations allowing the construction and the extraction of types. We focus on the derivation of accurate sharing information that we prove correct. The derivation process is designed as an application of a recent method for global analysis for logic programs, formalized by an abstract interpretation framework. The framework ensures correctness and termination of the inferred properties if certain requirements are satisfied.

1. Introduction

This paper presents a new approach to infer types in Prolog programs and deals with the problem of deriving detailed information as sharing between variables, or type information, allowing improvements in performance as shown by Mülkers et al. in [14]. The information computed by a type inference system allows for several optimizations, such as specialized code generation, use of specific unification modules, suppression of some choice points, or even clause indexing not reduced to the first parameter. The type information is also very useful for debugging. Abstract interpretation, first defined in [5], is a general framework in which it is possible to define techniques allowing for the static computation of information about the run-time behavior of programs; this has been widely studied in logic programming [1, 3, 6, 8, 9–11, 15]. Following [4], we call T-Abint any method based on abstract interpretation. A T-Abint specifies an abstract domain $\mathcal{D}$, whose elements (called...
abstract states) approximate the substitutions. The task of any T-Abint is to analyze
any logic program $P$ with a goal (or set of goals) and associate with each clause $C$ of
$P$ a set $S(C)$ of abstract states such that during the execution of $P$, whenever $C$ is called
with a substitution $\sigma$, there exists a state in $S(C)$ that approximates $\sigma$. If so, we will say
that the T-Abint is correct (or safe).

In this paper we consider types as set of terms that are not necessarily ground
Prolog terms. The novelty of this approach, compared with [10, 11, 20], is that we
require the existence of a type domain with some operations and few general proper-
ties. Our framework is much more general than the others because it does not rely on
a rigid type system. It is a kind of “meta”-type inference framework. The T-Abint
developed here can be used in any top-down or bottom-up abstract interpretation
[1, 3, 4, 9, 12, 15, 16]. We focus on the formal justification of an analysis of groundness
and sharing information, and of the proof and safety of type inferencing. Our proposal
is very close to that of [2, 4].

Roughly speaking, an abstract state $\beta$ is defined as a 4-tuple $(sv, tp, frm, Ps)$, where $sv$
corresponds to the equality constraints between variables of the domain of substitu-
tion, $tp$ is their type definition, $frm$ restricts their forms pointing out the relation of
inclusion between subterms and $Ps$ (for possible sharing) restricts the sharing relation
of variables. Each component is described by ad hoc properties; then we prove the
derived operation which abstracts the unification of concrete terms to be correct. The
rest of the paper is divided as follows; Section 2 describes the notations and definitions
used throughout the paper. Section 3 specifies the abstract states for type inferencing
and Section 4 presents the derived algorithm of abstract unification and a short proof
of its correctness. Section 5 is a comparison with related works and Section 6 sum-
marizes our conclusions.

2. Preliminaries

2.1. Notations

- $C_{\text{sub}} D$ denotes the set of substitutions $\theta$ such that $\text{dom}(\theta) = D$.
- $\# S$ denotes the cardinality of the set $S$.
- $\text{var}(t)$ denotes the set of variables occurring in the term $t$.
- $F \rightarrow G$ denotes a partial mapping form $F$ to $G$.

2.2. Normalized logic programs

We assume that the reader is familiar with the principles of logic programing (see
e.g. [13]). We only consider normalized Prolog programs, that is those in which the
operations of unification are explicitly written. More formally, a clause in normalized
logic program contains only distinct variables in its head, while the literals in the body
satisfy one of the following.
• \( x_i = x_j \)
• \( x_i = f(x_{i_1}, \ldots, x_{i_n}) \)
• \( p(x_{i_1}, \ldots, x_{i_n}) \)
where \( x_i, x_j, x_{i_1}, \ldots, x_{i_n} \) are distinct variables.

2.3. Substitutions

Let \( \text{Var} \) denote an enumerable set of variables and \( \text{Term} \) the set of valid Prolog terms. A substitution \( \theta \) is a total function from \( \text{Var} \) to \( \text{Term} \), such that the set \( \text{dom}(\theta) = \{ x \in \text{Var}: x \theta \neq x \} \) is finite. A substitution \( \sigma \) can be depicted by a list of \( \text{variable/value} \) pairs \( \{ x/\sigma: x \in \text{dom}(\sigma) \} \); \( \text{codom}(\sigma) \) denotes the set of variables \( \{ y \in \text{Var}: y \in \text{var}(x) \text{ and } x \in \text{dom}(\sigma) \} \). The composition of two substitutions \( \sigma \) and \( \phi \) is denoted \( \sigma \phi \). In this paper we only consider idempotent substitutions, i.e. \( \sigma \sigma = \sigma \).

2.4. Most general unifier

A substitution \( \theta \) is more general than a substitution \( \sigma \), written \( \sigma \leq \theta \), if there exists a substitution \( \phi \) such that \( \sigma = \theta \phi \). A substitution \( \sigma \) is a unifier of two terms \( a \) and \( b \) if \( a \sigma = b \sigma \). We call \( \sigma \) most general unifier (mgu) if \( \sigma \) is more general than any other unifier of \( a \) and \( b \).

2.5. Preliminary domain

Let \( (\mathcal{T}, \leq) \) be a given domain of types with:

- a monotonic concretization function \( \text{Cc}: \mathcal{T} \rightarrow \mathcal{P}(\text{Term}) \)
- four primitives, which have to be monotonic and consistent, specified as follows:
  - Cons (construction): It takes a functor of arity \( n \) and \( n \) types \( T_i \) and returns a new type \( T' \) such that \( t_i \in \text{Cc}(T_i) \forall i \implies f(t_1, \ldots, t_n) \in \text{Cc}(T') \).
  - Extr (extraction): It takes a functor of arity \( n \) and a type \( T \) and returns a tuple \( (T_1, \ldots, T_n) \) such that \( f(t_1, \ldots, t_n) \in \text{Cc}(T) \implies t_i \in \text{Cc}(T_i) \).
  - UaT (abstract unification of types): It takes two types \( T_1 \) and \( T_2 \) and returns a new type \( T \) s.t. \( t_i \in \text{Cc}(T_i), \sigma \text{ mgu}(t_1, t_2) \implies t_1 \sigma, t_2 \sigma \in \text{Cc}(T) \)
  - IaT (abstract instantiation of type): It takes a type \( T \) and returns a new type \( T' \) s.t. \( t \in \text{Cc}(T) \) and \( \sigma \) a substitution \( \Rightarrow t \sigma \in \text{Cc}(T') \).

3. Abstract domain

3.1. Motivation

An abstract state represents a set of concrete substitutions. In our framework, the notion of abstract states \( \beta = (sv, tp, frm, Ps) \) is based upon a set of indices (consecutive positive integers) \( [1..p] \) where each index corresponds to a subterm of a term bound
to a variable of some domain. The type of each subterm is defined by a function $t_p$ from $[1..p]$ to the type domain of interest. A partial function $frm$ from $[1..p]$ to a set $Frm_p$, describes the inclusion between subterms. An element of $Frm_p$ looks like $f(i_1, \ldots, i_n)$, where $f$ is a functor and $1 \leq i_1, \ldots, i_n \leq p$. Whenever $frm(i) = f(i_1, \ldots, i_n)$, this means that the term $t_i$ represented by the index $i$ is something like $f(t_{i_1}, \ldots, t_{i_n})$, where the concrete term $t_{i_j}$ is represented by the index $i_j$. This approach is a powerful representation of the relationship between terms and subterms. Moreover, one index may occur many times within the same form, or even within different forms. Possible sharing is described by a binary relation $Ps$ on indices having undefined forms. Finally, the relation between terms and variables is given by a function $sv$. Notice that we associate the same index with two different variables whenever they are equal.

In the following, we define precisely each component of an abstract state together with a concretization function $Cc$ and, finally, the abstract domain itself.

3.2. Same value component

Let $\theta = \{\ldots, X_i/t_i, \ldots\} \in Csub_D$. The component $sv$ on $D$ is any onto mapping $D \rightarrow [1..m]$, where $m = \# \{t_i: \forall i, j, i \neq j, t_i \neq t_j\}$. $Sv_m$ is the set of all $sv$ and $Sv = \bigcup_m Sv_m$. Thus,

$$Cc: Sv \rightarrow Csub_D$$

$$sv \mapsto \{\theta: \text{dom } \theta = D \text{ and } \forall x, x' \in D \text{ s.t. } sv(x) = sv(x') \Rightarrow x\theta = x'\theta\}.$$ 

The function $sv^{-1}$ splits $D$ into equivalence classes of elements associated with equal terms.

3.3. Type component

**Definition 3.1.** Any mapping $t_p$ on $[1..p]$, $p \in \mathbb{N}$: $t_p: [1..p] \rightarrow T \cup \{\bot\}$ is called a type component. $Tp_p$ denotes the set of these mappings for some $p$, and $Tp = \bigcup_p Tp_p$ denotes the entire set of mappings, for any $p$.

For each $p \in \mathbb{N}$, we define:

$$Cc: Tp_p \rightarrow \text{Term}^p$$

$$t_p \mapsto \{(t_1, \ldots, t_p) \forall i, 1 \leq i \leq p: t_i \in t_p(i)\}.$$ 

3.4. Form component

**Definition 3.2.** A form is a term $f(i_1, \ldots, i_p)$, where $f$ is a functor of arity $p$, $i_1, \ldots, i_p$ are strictly positive integers not necessarily distinct. $Frm_p$ is the set of forms such that $1 \leq i_1, \ldots, i_p \leq p$. Moreover, any constant is itself a form. A form component is a partial
Type inference in Prolog: a new approach

21

mapping \( \text{frm} : [1..p] \rightarrow \text{Term} \) such that \( \forall i, 1 \leq i \leq p: \text{frm}(i) = f(i_1, \ldots, i_p) \). When \( \text{frm}(i) \) is undefined, we just write \( \text{frm}(i) = \text{ind} \).

For each \( p \in \mathbb{N} \), we define:

\[
\text{Cc} : \text{Term}^p \rightarrow \text{Term}^p
\]

\[
\text{frm} \mapsto \{ (t_1, \ldots, t_p) \mid \forall i, 1 \leq i \leq p: \text{frm}(i) = f(i_1, \ldots, i_p) \Rightarrow t_i = f(t_{i_1}, \ldots, t_{i_p}) \}
\]

The form component describes the relation between a term and its subterms.

3.5. Possible sharing component

**Definition 3.3.** Let \( p \in \mathbb{N} \), we call component \( \text{Ps} \) on \([1..p]\) any binary symmetric relation on \([1..p]\). We denote by \( \text{Psh}_p \) the set of these relations for some \( p \), and \( \text{Psh} = \bigcup_p \text{Psh}_p \). Let \( \text{frm} \) be a form component on \([1..p]\). \( \text{Ps} \) is said to be compatible with \( \text{frm} \) if

\[
\forall i, j: 1 \leq i, j \leq p: \text{Ps}(i, j) \Rightarrow \text{frm}(i) = \text{ind} = \text{frm}(j).
\]

For each \( \text{Ps} \) defined with \( \text{frm} \), we consider the closure of \( \text{Ps} \), denoted \( \text{Ps}^* \), which is defined as the smallest \( \text{Ps} \) on \([1..p]\) such that \( \forall i, j, k \) s.t. \( 1 \leq i, j, k \leq p \):

1. \( \text{Ps}(i, j) \Rightarrow \text{Ps}^*(i, j) \)
2. \( \text{frm}(k) = f(\ldots, j, \ldots) \) and \( \text{Ps}(i, j) \Rightarrow \text{Ps}^*(i, k) \).

In the sequel, we only consider possible sharing component compatible with the form component. For each \( p \in \mathbb{N} \) we define

\[
\text{Cc} : \text{Psh} \rightarrow \text{Term}^p
\]

\[
\text{Ps} \mapsto \{ (t_1, \ldots, t_p) : \forall i, j: 1 \leq i, j \leq p \text{ and } \text{frm}(i) = \text{frm}(j) = \text{ind} \text{ such that } \text{var}(t_i) \cap \text{var}(t_j) \neq \emptyset \Rightarrow \text{Ps}(i, j) \}
\]

This notion of possible sharing component compatible with the form component is fundamental to prevent abnormal information during the abstract computation. It has been proved in [18] that the abstract operations preserve this property. Notice that the closure of \( \text{Ps} \) contains all the information about sharing, because the form component contains all the relations between a term and its subterms, whilst \( \text{Ps} \) contains all the relations between uninstantiated variables.

3.6. Abstract domain

We first define a “preliminary domain” and an order on its elements, and then we restrict this domain to obtain the effective abstract domain.
Definition 3.4. Let $\theta$ be a substitution $\{x_1/t_1, \ldots, x_n/t_n\}$, and $D = \text{dom}(\theta)$. The preliminary domain $P_{\text{sub}}$ is the subset of $\text{sv} \times T_p \times \text{Frm} \times \text{Psh}$ which satisfies the following conditions:

$$\beta = (\text{sv}, t_p, \text{frm}, Ps) \in P_{\text{sub}} \text{ if and only if}$$

1. $\exists m, p \in \mathbb{N}: m \geq m, \text{sv} \in \text{Sc}_m, t_p \in T_p, \text{frm} \in \text{Frm}_p$ and $Ps \in \text{Psh}$,
2. $\forall i, 1 \leq i \leq p: \text{frm}(i) = f(i_1, \ldots, i_q), (tp(i_1), \ldots, tp(i_q)) \subseteq \text{Extr}(f, tp(i))$,
3. $\forall i, m < i \leq p$ such that $\exists j, 1 \leq j \leq p: \text{frm}(j) = f(\ldots, i, \ldots)$.

An abstract state $\beta$ is a 4-tuple $(\text{sv}, t_p, \text{frm}, Ps)$ which, intuitively, gives information about $\theta$. The meaning of $t_p$ is, first, to define the type of the terms $t_1, \ldots, t_n$. The function $t_p$ is not defined as a mapping from $D$ to $\mathcal{F}$ for two reasons. First, the $\text{sv}$ component specifies that some $t_i$'s are equal; so it is sufficient to define $t_p$ on the codomain $[1..m]$ of $\text{sv}$. In other words, $m$ is the number of distinct terms among $t_1, \ldots, t_n$. This choice restricts the domain of $t_p$ and, moreover, avoids problems of inconsistency. Second, it is possible to extend the domain $[1..m]$ to $[1..p]$ (where $m \leq p$) in order to specify the type of some subterm of $t_1, \ldots, t_m$, i.e. to extend the “expressive power” of $\beta$. The meaning of the $\text{frm}$ component is to point out these subterms within the $t_i$'s. It also gives the principal functor of the $t_i$'s. The $Ps$ component is used to indicate that some terms $t_i, t_j$, or even some subterms, might share variables; notice that “not $Ps(i, j)$” means that the terms associated with $i$ and $j$ surely do not share variables.

The concretization function $Cc$ associated with $P_{\text{sub}}$ is defined as follows:

$$Cc: P_{\text{sub}} \rightarrow C_{\text{sub}}$$

$$\beta \mapsto \{\theta: \text{dom } \theta = D \text{ and } \exists t_1, \ldots, t_p \in \text{C}(t_p) \cap \text{C}(\text{frm}) \cap \text{C}(Ps^*) \text{ s.t. } \forall x \in D: x\theta = t_{\text{sv}(x)}\}$$

We are now able to define a partial order, denoted $\leq$, among the elements of $P_{\text{sub}}$. Consider two abstract states $\beta$ and $\beta'$, we say that $\beta' \leq \beta$ whenever the information induced by $\beta'$ is more restrictive than that induced by $\beta$. Intuitively this means that whenever a concrete substitution can be depicted by $\beta'$, it can also be done by $\beta$. In other words, the set of concretization of $\beta'$ is smaller than the one of $\beta$.

Definition 3.5. Let $\beta, \beta' \in P_{\text{sub}}$, $\beta' \leq \beta$ if and only if there exists a function $ti: [1..p] \rightarrow [1..p']$ such that

1. $\forall x \in D: sv'(x) = ti(sv(x))$,
2. $\forall i, 1 \leq i \leq p, tp'(ti(i)) \leq tp(i)$,
3. $\forall i: 1 \leq i \leq p: \text{frm}(i) = f(i_1, \ldots, i_q) \Rightarrow \text{frm'}(ti(i)) = f(ti(i_1), \ldots, ti(i_q))$,
4. $\forall i, j: 1 \leq i, j \leq p: \text{frm}(i) = \text{frm}(j) \Rightarrow \text{ind: } Ps^*(ti(i), ti(j)) \Rightarrow Ps(i, j)$. 
The relation ≤ defined on $P_{asubD}$ is a preorder, i.e. there exists elements of $P_{asubD}$ which are distinct but equivalent in the sense that
\[ \beta \neq \beta', \quad \beta \leq \beta', \quad \beta' \leq \beta \quad \text{and} \quad \text{Cc}(\beta) = \text{Cc}(\beta'). \]
In fact these elements are deducible from each other by a permutation of the indices.

**Definition 3.6.** Let $\equiv$ be the equivalence relation between two elements of $P_{asubD}$ which are distinct but equivalent in the sense above. The abstract domain $A_{subD}$ is then defined as
\[ A_{subD} = (P_{asubD}; \bot) \cup \{1\}. \]

Let $\beta \in P_{asub}$, $\bar{\beta}$ denotes the class of $\beta$ in $A_{subD}$.

The concretization function: $\text{Cc} : A_{subD} \rightarrow C_{subD}$ is defined as follows:
\[ \text{Cc}(\bot) = \emptyset, \]
\[ \text{Cc}(\bar{\beta}) = \text{Cc}(\beta). \]

We can now define an order on the elements of $A_{subD}$:
\[ \bot \leq \bar{\beta}, \quad \forall \beta \in A_{subD}, \quad \beta \leq \beta' \iff \beta \equiv \beta' \quad \forall \beta, \beta' \in P_{asubD}. \]

**Theorem 3.7.** There are no infinite increasing sequences in $A_{subD}$.

**Proof.** A formal version can be found in [18]. Here we just give a sketch of the proof. With each abstract state $\bar{\beta} = (sv, tp, frm, Ps)$, one can associate an integer defined as follows:
- (a) if $frm(i) = \text{ind}$ then weight$(i, \beta) = 1$ else weight$(i, \beta) = 1 + \sum_{j=1}^{n} \text{weight}(i, \beta)$ with $frm(i) = f(i_1, \ldots, i_n)$.
- (b) weight$(\beta) = \sum_{x \in D} \text{weight}(sv(x), \beta)$.

Let $\beta_1$ and $\beta_2$ be abstract states. One can then establish that $\beta_1 \leq \beta_2 \Rightarrow \text{weight}(\beta_1) \geq \text{weight}(\beta_2)$. Together with the hypothesis that in the types domain there does not exist a nonstationary increasing sequence, it is possible to conclude the nonexistence of an infinite chain in $A_{subD}$.

This result ensures that the algorithm of the abstract interpretation will terminate whenever the abstract operations are monotonic and consistent.

**4. Abstract operations**

In any framework of abstract interpretation [1, 3, 4, 9, 11, 15–18], it is necessary to define processes which mimic the operations in the concrete domain. In fact, the extension called procedure-exit in [1] and the abstract interpretation of built-ins
(X = X' or X = f(X_1, \ldots, X_n)) can be easily deduced from a kind of "super-unification" (an abstract unification of a list of pairs of terms). The only remaining point is the definition of the LUB (least upper bound), which relies on the properties of monotonicity. Its consistency is deducible from the fact that any LUB is an upper bound, which is de facto consistent. The rest of the abstract operations are easy to define because they are exact.

**Notation.** Let \( \alpha = (t, \text{frm}, \text{Ps}) \) be a \( q \)-tuple of abstract terms, i.e. \( t \in T_{q}, \text{frm} \in \text{Frm}_{q}, \text{Ps} \in \text{Psh}_q \). We note \( \text{Uact}_1(i, j, \alpha) = \alpha' \) with \( 1 \leq i, j \leq q \).

This notation is only another way of saying that we consider a state \( \beta = (s, t, \text{frm}, \text{Ps}) \) such that the co-domain of \( s \) is the set of indices \( [1..q] \). Moreover, this allows us to focus on the crucial information, the type and sharing of variables, rather than on the technical point, consistency of the indices.

**4.1. Abstract unification of pair of terms**

**Specification of \( \text{Uact}_1 \):** It takes two integers and a \( q \)-tuple \( \alpha \) and returns a \( q \)-tuple \( \alpha' \) such that

\[
(t_1, \ldots, t_q) \in \text{Cc}(\alpha) \quad \text{and} \quad \sigma = \text{mgv}(t_i, t_j) \quad \Rightarrow \quad (t_1, \ldots, t_q)\sigma \in \text{Cc}(\alpha').
\]

**Definition 4.1.** The operation \( \text{Uact}_1(i, j, \alpha) \) is defined only if \( \text{frm}(i) = \text{ind} = \text{frm}(j) \).

Assume \( \alpha' = (t', \text{frm}', \text{Ps}') \), the computation of each component of \( \alpha' \) is as follows:

1. \( t'(1 \leq k \leq q) \)
   - case of
     - (a) not \( \text{Ps}^*(i, k) \) and not \( \text{Ps}^*(j, k) \) then \( t'(k) = t(p(k) \)
     - (b) \( \text{Ps}^*(i, k) \) or \( \text{Ps}^*(j, k) \)
       (i) \( i = k \) or \( j = k \) then \( t'(k) = \text{UaT}(tp(i), tp(j)) \)
       (ii) \( i \neq k \neq j \) and \( \text{frm}(k) = f(k_1, \ldots, k_n) \) then \( t'(k) = \text{UaT}(tp(k), \text{Cons}(f, t'(k_1), \ldots, t'(k_n))) \)
       (iii) \( i \neq k \neq j \) and \( \text{frm}(k) = \text{ind} \) then \( t'(k) = \text{IaT}(tp(k)) \).
2. \( \text{frm}' = \text{frm} \).
3. Let \( P_1 = \{(k, l) \text{ s.t. } Ps(k, l), \text{ and } ng(tp'(k)), \text{ and } ng(tp'(l)) \} \) and \( P_2 = \{(k, l) \} \) such that \( Ps(k, l) \) and \( \exists k', l' \in \{i, j\} : Ps(k, k') \text{ and } Ps(l, l') \). We have that if \( ng(tp'(i)) \) then \( P' = P_1 \cup P_2 \) otherwise \( P' = P_1 \).

The previous definition is easy to understand though hard to read. We first compute the type of terms, which depends on their sharing. Obviously, the form component has not changed. Then we have to compute the new sharing relations (note that some may have disappeared due to the instantiation to a ground term). So if the term \( i \) is ground (the function \( ng \) returns false) no new sharing has been created so the \( P' \) is \( P \) minus some terms (propagation of groundness), otherwise, there might be new relations which are computed in \( P_2 \).
We have to prove that Uact1 is consistent, i.e. that the definition above respects the specification. As Cc(\(x'\)) is the intersection of the concretization of each component, we have to establish that \(t \sigma\) belongs to each concretization. We just sketch the proof which can be found in [18].

Due to lack of place, we only establish that \(t \sigma \in \text{Cc}(tp')\). As \(t = (t_1, \ldots, t_q)\), we have to prove the following fact: \(t_k \sigma \in \text{Cc}(tp') \forall k, 1 \leq k \leq p'\).

- If \(k\) does not share with \(i\) or \(j\), from the definition of \(Ps^*\), we have \(\text{Cc}(tp(k)) = \text{Cc}(tp'(k))\).
- If \(k\) is either \(i\) or \(j\) then \(t_k \sigma = t_i \sigma = t_j \sigma\) by the definition of mgu; thus \(t_k \sigma\) belongs to \(\text{Cc}(\text{UaT}(tp(i), tp(j))) = \text{Cc}(tp'(k))\) (see the specification of UaT).
- If \(\text{frm}(k) = f(k_1, \ldots, k_n)\), from the definition of \(Ps^*\) there exists \(k_s\) such that: \(\text{var}(t_i) \cap \text{var}(t_{k_s}) \neq 0\) or \(\text{var}(t_j) \cap \text{var}(t_{k_s}) \neq 0\), \(\sigma\) has an effect on the term \(t_{k_s}\) and so on \(t_k\). From this we have that \(t_k \sigma\) belongs to \(\text{Cc}(\text{UaT}(tp(k), \text{Cons}(f, tp'(k_1), \ldots, tp'(k_n))))\).
- Otherwise, \(\text{frm}(k) = \text{ind} = t_k\) is a variable, as there is a sharing between \(i\) and \(k\) or \(j\) and \(k\), then we have \(t_k \sigma \in \text{Cc}(\text{IaT}(tp(k))) = \text{Cc}(tp'(k))\).

We now have to prove that Uact1 is monotonic. The property is established for each component of a \(q\)-tuple of the abstract term.

**Property 4.2.** Let \(x_1\) and \(x_2\) be two \(q\)-tuples. Let \(i\) and \(j\) such that \(1 \leq i, j \leq q\), then we have:

\[x'_1 = \text{Uact1}(i, j, x_1) \preceq \text{Uact1}(i, j, x_2) = x'_2\]

**Proof.** We consider some \(k\) such that \(1 \leq k \leq q\), and we compute the different components of \(x'_1\) and \(x'_2\) w.r.t. \(k\). The proof relies on the property of monotonicity of the four primitives UaT, IaT, Cons and Ext. See [18] for details.

**4.2. Abstract specialization of a term**

**Specification of Specat:** it takes two integers and a \(q\)-tuple \(x\) and returns a \(q\)-tuple \(x'\) such that

\[(t_1, \ldots, t_q) \in \text{Cc}(x) \text{ and } \sigma = \text{mgu}(t_i, t_j) \Rightarrow (t_1, \ldots, t_q) \sigma \in \text{Cc}(x')\]

The operation Specat\((i, j, x)\) is defined only if \(\text{frm}(i) = \text{ind}\) and \(\text{frm}(j) = f(j_1, \ldots, j_n)\). As the operation is straightforward, but the formal definition a bit tedious, we describe only the process.

To compute \(\sigma\), the mgu of \(t_i\) and \(t_j\), it is possible to first compute the mgu \(\sigma'\) of \(t_i\) and \(f(y_1, \ldots, y_n)\) where the \(y_k \not\in \text{var}(t_i) \forall k\), and then the mgu of \((y_1, \ldots, y_n)\sigma'\) and \((t_{j_1}, \ldots, t_{j_n})\). We illustrate the process by the following example.
Example. Let \( \theta = \{X_1/g(Z), X_2/Z, X_3/f(a, b), X_4/Y\} \). Let \( \alpha=(tp, frm, Ps) \) describe properties about the value bound to \( X_i \). For the sake of simplicity, we consider the generalized modes type domain. We then have:

\[
\begin{align*}
\text{tp} &= \{\langle 1, \text{var} \rangle, \langle 2, \text{var} \rangle, \langle 3, \text{ground} \rangle, \langle 4, \text{var} \rangle, \langle 5, \text{ground} \rangle, \langle 6, \text{ground} \rangle\} \\
\text{frm} &= \{\langle 1, g(2) \rangle, \langle 3, f(5, 6) \rangle, \langle 5, a \rangle, \langle 6, b \rangle\} \\
Ps &= \emptyset
\end{align*}
\]

Suppose that we compute \( \text{Specat}(2, 3, \alpha) = (tp', frm', Ps') \), then

\[
\begin{align*}
\text{tp}'(1) &= t_2T(tp(1), tp(2)) = \text{ground} \\
\text{tp}'(2) &= t_2 \cup t_3T(tp(2), \text{Cons}(f, tp(5), tp(6))) = \text{ground} \\
\text{tp}'(i) &= \text{tp}(i) \quad \text{for} \quad i = 3, \ldots, 6 \\
\text{frm}' &= \text{frm} \cup \{\langle 2, f(5, 6) \rangle\} \\
Ps' &= Ps
\end{align*}
\]

4.3. Abstract unification of a pair of terms

Specification of Uact: It takes two integers and a \( q \)-tuple \( \alpha \) and returns a \( q \)-tuple \( \alpha' \) such that

\[
(t_1, \ldots, t_q) \in Cc(\alpha) \quad \text{and} \quad \sigma = \text{mgu}(t_i, t_j) \Rightarrow \exists (t'_1, \ldots, t'_q) \in Cc(\alpha') \quad \text{and} \quad \\
\exists \theta \text{ s.t. } (t_1, \ldots, t_q)\sigma\theta = (t'_1, \ldots, t'_q)
\]

Definition 4.3. \( \text{Uact}(i, j, \alpha) = \alpha' \) is defined as follows:

(a) \( i = j \Rightarrow \alpha = \alpha' \)
(b) \( i \neq j \) and \( \text{frm}(i) \neq \text{ind} - \text{frm}(j) \Rightarrow \alpha' = \text{Uact1}(i, j, \alpha) \)
(c) \( i \neq j \) and \( \text{frm}(i) \neq \text{ind} \) or \( \text{frm}(j) \neq \text{ind} \) if \( \text{frm}(i) = \text{ind} \) then \( \alpha' = \text{Specat}(i, j, \alpha) \) else \( \alpha' = \text{Specat}(i, j, \alpha) \)
(d) otherwise, there must exist a functor \( f \) such that \( \text{frm}(i) = f(i_1, \ldots, i_n) \) and \( \text{frm}(j) = f(j_1, \ldots, j_n) \). Assume that \( \alpha_k = \text{Uact}(i_k, j_k, \alpha_{k-1}) \) with \( k = 1, \ldots, n \). Then \( \alpha' = \text{Fct}(i, j, \alpha_n) \) where \( \text{Fct} \) is some process that permits to merge two indices \( i \) and \( j \) having the same property.

The definition of Uact is recursive, so we have to give some argument to justify its noncircularity. Let \( h(i, \alpha) \) the height of some index \( i \) in \( \alpha \) defined below.

Definition. If \( \text{frm}(i) = \text{ind} \) then \( h(i, \alpha) = 0 \) else \( h(i, \alpha) = 1 + \max(h(i_1, \alpha), \ldots, h(i_n, \alpha)) \) with \( \text{frm}(i) = f(i_1, \ldots, i_n) \).

As \( \text{Uact}(i, j, \alpha) \) does not modify the value of \( \max(h(i, \alpha), h(j, \alpha)) \), one can establish that \( \max(h(i_k, \alpha_{k-1}), h(j_k, \alpha_{k-1})) \leq \max(h(i, \alpha), h(j, \alpha)) \).

We now have to demonstrate that Uact is monotonic and consistent. As the proofs are really tedious and hard to read, we just sketch them.
Property 4.4. Uact is consistent in the sense of its specification.

Proof (Musumbu [18]). We focus only on the point $d$ of the Definition 4.3. Since there exists a unifier of $t_i$ and $t_j$, the terms have the same functor with the same arity $n$. There exist substitutions $\sigma_0, \ldots, \sigma_n, \sigma'_1, \ldots, \sigma'_n$ such that

- $\sigma_0$ is the empty substitution,
- $\sigma'_k = \text{mgu}(t_k, \sigma_{k-1}, t_j, \sigma_{k-1})$, $\forall k, 1 \leq k \leq n$,
- $\sigma_k = \sigma_{k-1} \sigma'_k$, $\forall k, 1 \leq k \leq n$

Let $\sigma = \sigma_n \tau$ where $\tau$ is a renaming substitution. We prove by induction on $k$ that there exists a $q$-tuple of abstract terms $s_k$ and a substitution $\theta_k$ such that $s_k = t \sigma_k \theta_k$ where $s_k \in \text{Cc}(a_k)$.

Property 4.5. Let $\alpha_1$ and $\alpha_2$ be two $q$-tuples. Let $i, j$ such that $1 \leq i, j \leq q$. We have

$$\text{Uact}(i, j, \alpha_1) \preceq \text{Uact}(i, j, \alpha_2).$$

Proof. This proof is very lengthy. It is by an induction on the structure of the definition of Uact.

4.4. Abstract unification for a list of pairs of terms

This operation, called Ualct, is a tail-recursive generalization of Uact. Let $l = ((i_1, j_1), \ldots, (i_r, j_r))$ such that $1 \leq i_1, j_1, \ldots, i_r, j_r \leq q$. Let $u \in \text{Cc}(\alpha)$ and $\sigma$ be the mgu of $(u_{i_1}, \ldots, u_{i_r})$ and $(u_{j_1}, \ldots, u_{j_r})$.

Specification of Ualct: It takes a list of pairs of terms and a $q$-tuple and returns a $q$-tuple such that

$$\text{Ualct}(l, \alpha) = \alpha' \Rightarrow \exists v \in \text{Cc}(\alpha') \text{ and } \exists \theta \text{ s.t. } u \sigma \theta = v.$$

Theorem 4.6. Let $\alpha, i, j$ s.t. $1 \leq i, j \leq q$ and $l$ a list of pairs of indices belonging to $[1..p]$. Then we have

$$\text{Ualct}((i, j), l, \alpha) = \text{Ualct}(l, \text{Uact}(i, j, \alpha))$$

with the convention that $\text{Ualct}(l, \bot) = \bot$.

Proof. Immediate, by induction on the structure of the definition of Uact.

4.5. Least upper bound

In this section we define the notion of least upper bound in our abstract domain. This operation permits the representation of a set of abstract states by a unique state.

Specification of LUB: Let $\beta_1$ and $\beta_2$ be abstract states on the same domain $D$. Then $\text{LUB}(\beta_1, \beta_2) = \beta'$ such that

- $\theta \in \text{Cc}(\beta_1)$ or $\theta \in \text{Cc}(\beta_2) \Rightarrow \theta \in \text{Cc}(\beta')$,
- $\beta_1, \beta_2 \preceq \beta'$ and $\forall \beta, \beta_1, \beta_2 \preceq \beta \Rightarrow \beta' \preceq \beta$. 
Definition 4.7. Let $D$ be the domain of the abstract states $\beta_1, \beta_2$. As the index associated with a variable can be different in $\beta_1$ and in $\beta_2$, we have to define two sets:

$$E = \{(i, j): \exists X \in D \text{ s.t. } i = sv_1(X) \text{ and } j = sv_2(X)\},$$

$$F = \{(i, j): (i, j) \in E, \text{ or } \exists i', j', k \text{ such that } (i', j') \in F \text{ and } frm_1(i') = f(i_1, \ldots, i_n) \text{ and } (i', j') = (i_k, j_k) \text{ and } frm_2(j') = f(j_l, \ldots, j_m)\},$$

where $p' = \# F$ and $fc$ is a 1-to-1 mapping from $F \rightarrow [1..p']$. Each component of $\beta'$ is computed as follows:

- $tp'(k) = \text{Lub}(tp_1(i), tp_2(j))$ where $k = fc(i, j)$ and Lub is a least upper bound defined on the given type domain.
- $frm'(k) = f(k_1, \ldots, k_n)$ where $k = fc(i, j)$, $frm_1(i) = f(i_1, \ldots, i_n)$, $frm_2(j) = f(j_1, \ldots, j_m)$ and $k_l = fc(i_l, j_l)$
- $Ps' = \{(k, k') \text{ such that } frm'(k) = frm'(k') = \text{id} \text{ and } 3i, j, i', j', fc(i, j) = fc(i', j') = k' \text{ and } Ps_1(i, i') \text{ or } Ps_2(j, j')\}$.
- $sv'(X) = fc(sv_1(X), sv_2(X))$, $\forall X \in D$.

Example. Let $\beta_1 = (sv_1, \ldots, Ps_1)$ and $\beta_2 = (sv_2, \ldots, Ps_2)$ where $sv_1 = \{\langle X_1, 1\rangle, \langle X_2, 2\rangle, \langle X_3, 2\rangle, \ldots\}$ and $sv_2 = \{\langle X_1, 2\rangle, \langle X_2, 1\rangle, \langle X_3, 1\rangle, \ldots\}$. We leave the type and the possible sharing components undescribed. Let $frm_1 = \{\langle 2, f(4, 5)\rangle\}$ and $frm_2$ be $\{\langle 1, f(4, 6)\rangle\}$. Then we have

$$E = \{(1, 2), (2, 1), \ldots\} \text{ and } F = \{(1, 2), (2, 1), \ldots, (4, 4), (5, 6), \ldots\}.$$ Moreover let $fc: F \rightarrow [1..p']$ s.t. $fc(1, 2) = 1, fc(2, 1) = 2, fc(4, 4) = 7, fc(5, 6) = 8$; then it is possible to compute the different components as, for example:

$$tp'(1) = \text{Lub}(tp_1(1), tp_2(2)), \ldots$$

$$frm'(2) = f(7, 8), \ldots$$

4.6. Some more examples

We assume the existence of an abstract interpretation algorithm with memorization, see for instance that of [4, 9, 11, 12]. None of them will be described. The only interesting point is that we perform the analysis for each module, with an initial query wherein all the arguments are unbound variables. As the abstract states contain information such as possible sharing, form description, equality constraint (the $sv$ component) and type description, the abstract state associated with each clause is very accurate as illustrated in the following example.

Example. Let us consider the well-known append program which looks like:

$$\langle 1 \rangle \text{ app}(X, Y, Z) : - X = [], Y = Z.$$

$$\langle 2 \rangle \text{ app}(X, Y, Z) : - X = [A|B], Z = [A|C], \text{ app}(B, Y, C).$$
Assume that the domain of type consists of lists and prolog terms, where List can be depicted as follows:

\[
\text{List ::= } [] | .(\text{Term, List}).
\]

Then the analysis returns the following abstract states:

\[
\begin{align*}
\langle 1 \rangle & : \{\langle \text{A}, 1 \rangle, \langle \text{L}, 2 \rangle, \langle \text{Z}, 3 \rangle, \langle \text{A}, 4 \rangle, \langle \text{B}, 5 \rangle, \langle \text{C}, 6 \rangle\}, \{\langle 1, \text{List}\rangle, \langle 2, \text{var}\rangle\}, \{\langle 1, []\rangle\} \\
\langle 2 \rangle & : \{\langle \text{X}, 1 \rangle, \langle \text{Y}, 2 \rangle, \langle \text{Z}, 3 \rangle, \langle \text{A}, 4 \rangle, \langle \text{B}, 5 \rangle, \langle \text{C}, 6 \rangle\}, \{\langle 1, \text{List}\rangle, \langle 2, \text{List}\rangle, \langle 3, \text{List}\rangle, \langle 4, \text{var}\rangle, \langle 5, \text{List}\rangle, \langle 6, \text{List}\rangle\}, \{\langle 1, (4, 5)\rangle, \langle 3, (4, 6)\rangle\}
\end{align*}
\]

Note that these states correspond to success patterns. Moreover, whenever the type of some variable is \text{var}, which stands for unbounded variable, any value of this parameter is acceptable in any query. It is well known that the call \text{app}([], 9, \text{X}) succeeds (unfortunately) and bind \text{X} to 9.

It is possible to decrease the complexity of this step by ordering the different predicates of the program \text{P}, and first analyzing the predicates that do not call any others. Whenever there is a cycle, select one of the predicates in the cycle and perform the ordering on the rest of the predicates.

**Example.** The predicate \text{ordo} orders the elements of a list

\[
\text{ordo}([], L, L):-!.
\]
\[
\text{ordo}(L, [], L):-!.
\]
\[
\text{ordo}([X|L1], [X|L2], L):-!, \text{ordo}(L1, L2, L).
\]
\[
\text{ordo}([X|L], [Y|L1], [Z|L2]) :- \text{sup}(X, Y), !, X=Z, \text{ordo}(L, [Y|L1], L2).
\]
\[
\text{ordo}([X|L], [Y|L1], [Z|L2]) :- Z=Y, \text{ordo}([X|L], L1, L2).
\]
\[
\text{sup}(s(X), 0).
\]
\[
\text{sup}(s(X), s(Y)) :- \text{sup}(X, Y).
\]

On this example, the ordering gives \text{sup, ordo}. And the static analysis can be performed by:

1. analyze \text{sup}.
2. analyze \text{ordo} (use the states computed at step 1.)

Assume that the domain of type contains prolog terms, lists defined as above and naturals \text{Nat ::= 0 | s(Nat)}. The abstract interpretation finds that the arguments of predicate \text{sup} have type \text{Nat}. The analysis of \text{ordo} raises to the following states (only the \text{sv} and \text{tp} components are given)

\[
\begin{align*}
\langle 1 \rangle & : \{\langle \text{A}, 1 \rangle, \langle \text{L}, 2 \rangle\}, \{\langle 1, \text{List}\rangle, \langle 2, \text{var}\rangle\} \ldots \\
\langle 2 \rangle & : \{\langle \text{L}, 1 \rangle, \langle \text{A}, 2 \rangle\}, \{\langle 1, \text{var}\rangle, \langle 2, \text{List}\rangle\} \ldots
\end{align*}
\]
\[ \langle 3 \rangle \left( \left\{ \langle A, 1 \rangle, \langle B, 2 \rangle, \langle L, 3 \rangle, \langle X, 4 \rangle, \langle L1, 5 \rangle, \langle L2, 6 \rangle \right\}, \{1, 2, 3, 5, 6 \text{ are List and } \langle 4, \text{var} \rangle \} \ldots \right) \]

\[ \langle 4 \rangle \left( \left\{ \langle A, 1 \rangle, \langle B, 2 \rangle, \langle C, 3 \rangle, \langle X, 4 \rangle, \langle L, 5 \rangle, \langle Y, 6 \rangle, \langle L1, 7 \rangle, \langle Z, 4 \rangle, \langle L2, 8 \rangle \right\}, \{4, 6 \text{ are Nat, the rest of variables are List} \} \ldots \right) \]

\[ \langle 5 \rangle \text{ same as } \langle 4 \rangle \text{ except that } X, Y \text{ and } Z \text{ are var} \]

The variables, \( A, B, C \) appearing in the abstract states are created by the normalization of the program.

5. Related work

In this section we consider some other frameworks and point out either the differences or our methods of simulation.

First of all we mention the approach of De Boeck and Le Charlier \([8]\) which was independently developed and is very close to ours. The differences reside in two points: first, it is possible for us to find (deduce) the type of a term which has a known form, and second, we believe that our presentation provides easier proofs of correctness and monotonicity.

In \([11]\) Kanamori and Kawamura present a framework for analyzing Prolog programs based on OLDT Resolution \([19]\), a top-down Prolog interpreter with memorization. They consider a type definition as set of definite clauses satisfying the following two conditions:

(1) The head of each clause is a unary predicate \( p \) called type predicate. The argument of \( p \) is either a constant \( b \) called bottom element of \( p \), or a term \( t \) of the form \( c(X_1, \ldots , X_n) \) where \( c \) is said to be a constructor of \( p \).

(2) The body of each clause consists of literals whose predicate is a type predicate and whose arguments \( X_j \) are in the head. The type of a type predicate \( p \) is the set of terms \( t \) such that \( p(t) \) succeeds without instantiating variables of \( t \).

Our framework is as efficient as theirs. For example, if we consider disjoint types:

- \( \text{any} \) is the set of all terms,
- \( \text{p}_i \) is the set of terms of type \( p_i \), and
- \( \emptyset \) is the empty set.

(1) \( \text{UaT} \) can be defined as follows: Let \( A \) and \( B \) be literals, \( \nu \) the type substitution associated with \( A \), and \( \tau \) the type substitution associated with \( B \). First, unify the two literals, and let \( \eta \) (if it exists) be the mgu. The information types (\( \nu \) and \( \tau \)) are then propagated in two steps: an inward propagation from terms to subterms and an outward propagation from subterms to terms. See \([11]\) for the formal definitions.

(2) \( \text{IaT} \) is only the outward propagation.

In \([10]\), the authors introduce type graphs which allow high levels of precision, and permit the representation of more type values. Such type graphs give not only information about the degree of instantiation, but also about the names and the
positions of functors occurring in the terms. By means of the form component, we can obtain such information easily, as long as the type domain is restricted to finite terms.

6. Conclusions

We have presented an abstract domain which deals with type inference and derivation of accurate sharing information. Moreover, we have proved its correctness. Notice that our technique is much more efficient than any other well-known framework dealing with mode inference. The novelty of our approach resides in a “meta”-type inference framework. That is to say that on the contrary to [10, 11, 20], our framework is independent of the type domain. We have given some examples highlighting how we can handle the techniques of type inference of others.

Acknowledgment

This work was partially supported by the GRECO de Programmation (METHEOL project).

References