A bottom-up polymorphic type inference in logic programming

Roberto Barbuti and Roberto Giacobazzi

Dipartimento di Informatica, Università di Pisa, Corso Italia 40, 56125 Pisa, Italy

Communicated by M. Sintzoff
Received February 1990
Revised April 1992

Abstract


We present a type inference system for Horn clause logic programs, based on a bottom-up abstract interpretation technique. Through the definition of suitable abstract operators, we obtain an abstract immediate consequence operator map associated with the program to analyze. The least fixpoint of such an operator gives an approximated description, by means of types, of the success set of the program. By changing the abstract domain of types, we easily obtain different type inference systems. This is useful to make the inference appropriate for different purposes. Due to the semantic basis, the system declaratively handles type union and parametric polymorphism.

1. Introduction

Logic programming has been introduced as an untyped language, without any discipline on types. Such flexibility is often paid in terms of difficult debugging and not efficient computations. On the other hand, a strong type discipline as in Pascal precludes the flexibility which is a key point of the logic programming style. Given the importance of the topic, many studies were devoted to type schemes for logic programming. The proposed approaches can be partitioned in two main classes, type checking and type inference. Type checking consists in verifying whether the use of types in a program is consistent with some type declarations. On the other hand, type inference is the process of determining the type of program units, given a program with incomplete type declarations. Type inference was first studied within the functional paradigm [23, 28]. Its application to logic programming has been
done with a conceptually different meaning. In functional programming, all the rules (or branches) defining a function must be potentially applicable to values in the same domain. Thus, if a function is defined by rules involving different (comparable) types, the overall type is the greatest lower bound of them. On the other hand, type inference in logic programming respects the untyped nature of the paradigm. If the clauses defining a predicate generate results with different types, the inferred type is simply the union of them, without imposing restrictions. Thus, type inference in logic programming results in the approximated description, by means of types, of the success set. This result can be used as a powerful tool for debugging logic programs.

A number of different semantics for logic programs are possible as a basis for abstract interpretation. The distinction between top-down and bottom-up data-flow analyses in imperative and functional languages helps also in logic programming to distinguish between two different classes of analyses [25]. Top-down analyses propagate the information in the same direction as an SLD-refutation, whereas bottom-up analyses propagate information in the opposite direction, thus returning an approximation of the whole success set. A number of studies have been devoted to show the equivalence of the two approaches in logic programming [3, 7]. However, since the bottom-up one returns an approximation of the success set of the program, it is more adequate for goal-independent type inference.

The aim of this paper is to show how polymorphic type inference can be seen as an instance of bottom-up abstract interpretation. Moreover different type inferences can be obtained as instances of the framework with different abstract domains. This approach can be used to study the relationships among various type inference systems.

Logic programming is not extended with complex type structures, but types are only used to describe approximations of success sets in a concise way. We show how polymorphic typing can be handled in a very natural way by abstracting an underlying (concrete) semantics defined in terms of non-ground semantic objects. The advantages of this approach come from the use of a new declarative semantics for logic programs via a bottom-up abstract interpretation, namely:

- the correctness proof can be carried out in a standard way,
- the inference process is goal-independent (we generate a success set on the domain of “polymorphic types” with respect to which any goal can be analyzed),
- type inference can be easily enriched simply by changing the domain of types.

The above features characterize this approach with respect to the ones in the literature.

The paper is organized as follows. Section 2 discusses other approaches to the problem. Section 3 briefly reviews the notion of abstract interpretation, the semantics basis of our approach and a bottom-up abstract interpretation scheme. Section 4 presents a type inference system, together with some examples. In Section 5, different
definitions of the abstract domain and operators lead to a discussion about the role of bottom-up abstract interpretation for type inference in logic programming. Finally, Section 6 concludes.

2. Related work

In this section we discuss works about the introduction of type information in logic programming. Such studies are conceptually partitioned in three classes: type checking, type inference, and abstract interpretation-based type inference. One of the first contributions on a polymorphic type system in logic programming is the one of Mycroft and O'Keefe [30], inspired by the type system of ML [28]. Polymorphism results from the possibility to have type variables in the type structures. In this type system for Prolog, the programmer has to declare types for predicates and functions. The approach is to consider type specifications as restrictions on the arguments of predicates and terms, and a static type checker is devoted to check that these restrictions are respected. In [15], Gang and Zhiliang give a type system for Prolog with type declarations. It is mainly intended to gain efficiencies at compile-time and run-time. In [17], Hanus describes a type system for HCL, where predicates and functions are declared with polymorphic types. The obtained polymorphic type system is an extension of the one in [30]. A model theoretic and an operational semantics are given. A specialized unification algorithm extends the refutation method to a typed world. One of the first works about type inference is presented by Mishra in [29]. The type of a predicate describes all terms for which the predicate may succeed, a predicate $p$ has type $\tau$ if $p(t)$ fails for all $t$ "outside" $\tau$. The inferred types are symbolic descriptions of terms by means of ground regular trees (only monomorphic types are inferred). Type inference is performed by a set of inference rules on a restricted class of Prolog predicates. The extension to a polymorphic type scheme presents some problem due to difficulties in handling parameterized regular trees. Zobel [36] presents an approach similar to the previous one. A syntactic type inference based on a specialization of the classical unification algorithm is presented. It does not require any type declaration: The inference process returns a set of type rules: $\alpha \leftarrow \{f(\beta)\}$ for each predicate in the program. The type of a predicate has the form $p(\eta_1, \ldots, \eta_n)$ together with a set of type rules $S = \{\alpha_j \leftarrow \omega_j\}$, in which each $\eta_i$ may be defined. For the sake of finiteness, the clauses are explored only once, thus recursive clauses can give rise to highly incomplete type information. Pyo and Reddy follow this line by presenting a type inference algorithm based on the notion of type as a set of values [31]. An approach based on a top-down abstract interpretation technique can be found in Bruynooghe et al. [6]. To obtain a finite lattice of types (the abstract domain), a normalized type graph (rational tree) is associated with each type. The meaning of a graph is the set of ground terms which can be folded onto it. The presence of restrictions ensures that all type graphs are finite and that only a finite number of such graphs can be constructed from a given finite set of
functor symbols. Type inference is performed by specializing the abstract interpretation scheme, which generate an AND/OR graph associated with the refutation process for a given goal. The main problem in this approach is the strong approximation induced by the restrictions on type graphs: for example, to limit the depth of the graph, any acyclic path is not allowed to contain the same functor symbol twice. An extension of this approach is given in [5]. Kluzniak [21] uses top-down abstract interpretation to derive a description of the solutions of a Ground Prolog goal, starting from mode declarations. The above approaches do not use type names, types are described by different representations of a set of values. The following two approaches are closer to ours. The Herbrand Universe is partitioned in classes, and each class is described by a type term. Then, for each predicate an approximation of its success set is inferred and its representation by means of type terms is given. In [19], Kanomori and Horiuchi present a type inference based on an abstract interpretation technique. The polymorphic inference derives from a variation (at the abstract level) of the OLDT-resolution by Sato and Tamaki (this is a hybrid interpretation for logic programs, based on both a top-down and a bottom-up analysis of the program itself). Since the algorithm might generate an infinite number of solutions, a depth-k cutting prevents the analysis from infinite loops [32]. Our type structure and inference is similar to this one. One of the main differences is the use of variables in our type patterns (instead of the any value), thus preserving the connection among arguments of polymorphic types. Moreover, the use of bottom-up abstract semantics allow us to give a view of predicate types in the whole program (not only those involved in a specific goal). In [34], a paper by Xu and Warren, the typing is a consequence of a partition of the Herbrand Universe based on type declarations. The inference procedure is performed by a program transformation which returns a type inference logic program, and then by querying this new program. A call to PROVE (the type inference procedure) is performed for each predicate defined in the original program. The answer obtained by PROVE, which represents the associated type pattern, can contain uninstantiated variables. These variables are considered as globally quantified on types, thus handling polymorphism. Because of the groundness of the Herbrand Universe, the treatment of parametric polymorphism does not result uniform with respect to the monomorphic case. The semantic foundation of polymorphism cannot be given in a declarative style, through the notion of Herbrand model. The same problem is present in [35] where types are seen as regular sets of terms. In this paper type inference and type checking are studied using abstract interpretation. However, the approach does not allow to infer parametric types. Moreover the type of a predicate is considered to be unique.

We define a polymorphic type inference system for logic programs, based on a bottom-up abstract interpretation scheme. This approach overcomes the problems of the previous ones, mainly for two points: (i) the use of bottom-up abstract interpretation allows a goal-independent analysis, and (ii) the use of a suitable semantics gives a natural (semantics-based) treatment of polymorphism. Through the definition of a suitable set of abstract operators we obtain the abstract (finitely
convergent) counterpart of the immediate consequence operator $T_P$. The least fixpoint of it is the approximation of the success set of the program by means of type patterns describing the well-typed atoms in it.

3. Preliminaries

In this section we present the bottom-up abstract interpretation framework together with a $T_P$-based concrete semantics for logic programs. Any concept not formally defined can be found in [2, 3, 11, 12].

3.1. Abstract interpretation

We formalize the notion of abstract interpretation according to the ideas given by Cousot and Cousot [9]. An introduction to the subject of abstract interpretation in the field of logic and functional languages can be found in [1]. Let us consider a concrete domain $D_P$ (a set of computational state representations associated with the generic program $P$), which is a complete lattice with respect to a partial order relation $\leq_{D_P}$, and a state transition function $E_P$, defined on $D_P$. Let us assume that the program's computational behavior is fully characterized by a solution of the recursive equation $x = E_P(x)$. Thus we can state that $(D_P, \leq_{D_P}, E_P)$ defines a standard fixpoint semantics for the program $P$. Since computations can be infinite, the above definition cannot be used for static program analysis. Therefore we define a non-standard fixpoint semantics based on an abstract (usually finite or even Noetherian) domain $G_P$, called abstract semantics $(G_P, \leq_{G_P}, E_P)$, to guarantee the effectiveness of the analysis. Obviously, this kind of analysis is less precise than the one given by the complete semantics definition.

The relation between the standard and the abstract semantics was established in [9] by defining a pair of functions $\alpha$ and $\gamma$ (abstraction and concretization), which form a Galois insertion between $(D_P, \leq_{D_P})$ and $(G_P, \leq_{G_P})$ [9, 27]. This notion allows us to formally handle the correctness condition on the abstract interpretation framework. Formally, an abstract interpretation scheme [9, 25] is a tuple $(\langle D_P, \leq_{D_P} \rangle, E_P, \langle G_P, \leq_{G_P} \rangle, \alpha, \gamma)$ such that:

1. $(D_P, \leq_{D_P})$ and $(G_P, \leq_{G_P})$ are complete lattices,
2. $E_P : D_P \to D_P$ and $E_P : G_P \to G_P$ are monotonic functions,
3. $\alpha : D_P \to G_P$ (abstraction) and $\gamma : G_P \to D_P$ (concretization) are monotonic functions,
4. $\forall d^* \in G_P, \alpha(\gamma(d^*)) = d^*$,
5. $\forall d \in D_P, d \leq_{D_P} \gamma(\alpha(d))$.

The last three conditions define a Galois insertion between the abstract domain and the concrete one. Galois insertions give us the formal framework to prove the correctness of an abstract interpretation. They define a strong relationship between the concrete domain of computations and the abstract, usually simpler, domain
associated with the analysis. Their usefulness was outlined in [9, 10] as a base to
develop soundness conditions for semantics-based analysis of flowchart programs.
They also ensure that the domain $\mathcal{D}_P$ (the abstract domain of computation) does
not contain redundant elements. According to conditions (4) and (5), the concretiza-
tion cannot cause any loss of information, while the abstraction of a concrete
object may cause some loss of information [9]. Given a Galois insertion $(\alpha, \gamma)$ on
the lattices $(D_P, \preceq_{DP})$ and $(\mathcal{D}_P, \preceq_{\mathcal{D}P})$, the following statements hold [10, 27]:

- abstraction and concretization uniquely determine each other,
- $\gamma$ preserves arbitrary meets, that is $\forall \mathcal{B} \subseteq \mathcal{D}_P, \quad glb_{\mathcal{D}_P}\{\gamma(d^{\alpha})|d^{\alpha} \in \mathcal{B}\} = \gamma(glb_{\mathcal{D}_P}\mathcal{B})$,
- $\alpha$ preserves arbitrary joins (i.e. $\alpha$ is additive), that is $\forall B \subseteq D_P, \quad lub_{\mathcal{D}_P}\{\alpha(d)|d \in B\} = \alpha(lub_{\mathcal{D}_P}B)$.

The program analysis is done by computing a (finite) Kleene sequence
$\{\mathcal{E}_P^n(\bot_{\mathcal{D}_P})\}_{n \in \{1, \ldots, k\}}$ of finitely computable approximations of the semantic solution
of the recursive equation $x = E_P(x)$. The correctness of such an approximation can
be given by conditions relating $E_P$ with $\mathcal{E}_P$: for example, $\forall d \in D_P, \quad \alpha(E_P(d)) \preceq_{\mathcal{D}_P} \mathcal{E}_P(\alpha(d))$ [9, 10]. In our case, we will consider a weaker notion of
correctness (see Theorem 4.23).

In the following sections we introduce a declarative semantics for logic programs
adequate for semantics-based program analyses and the main ideas of a framework
for bottom-up logic program analysis based on such a semantics.

3.2. Bottom-up semantics of logic programs

The following results about the semantics of logic programs are defined in [11, 12].
This semantic definition is closer than the one in [24, 33] to the operational behavior
of logic programs, which was shown to require a more complex and appropriate
notion of model, in general different from the minimal Herbrand model. The model
theoretic and fixpoint semantics in [12] capture possibly non-ground computed
answer substitutions. Such a semantic definition is then the right one in order to
develop a bottom-up abstract interpretation framework, without requiring to extend
the standard semantics with a somewhat artificial collecting semantics. A standard
semantics able to characterize the operational behavior of logic languages can
therefore be found inside the first-order logic semantics (model theoretic and
fixpoint), by considering possibly non-initial models, having a richer information
structure, like those giving a representation (usually finite by means of variables)
of infinite sets of ground atoms. We have to consider a different notion of program
equivalence, rich enough to model the useful information for program analysis, i.e.
the observable properties of the operational semantics.

We consider an extended Herbrand Universe containing also non-ground terms.
Thus, it is possible to define a non-ground success set and to declaratively character-
ize the ability of logic programs to compute answer substitutions (which are non-ground in general).

Let us consider the set $\Sigma$ of constructors, with typical elements $a$, $b$, $c$, ..., (constructors with 0 arguments $\Sigma_0$) and $f$, $g$, $h$, ... (constructors with one or more arguments $\Sigma_n$, $n > 0$), and a denumerable set $Var$ of variables. The free $\Sigma$-algebra on $Var$, $T_\Sigma(Var)$, is inductively defined as the least family such that:

- $\forall c \in \Sigma_0$, $c \in T_\Sigma(Var)$;
- $\forall v \in Var$, $v \in T_\Sigma(Var)$;
- $\forall t_1 \in T_\Sigma(Var), \ldots, \forall t_n \in T_\Sigma(Var), \forall d \in \Sigma_n$, $d(t_1, \ldots, t_n) \in T_\Sigma(Var)$.

Notice that the standard Herbrand Universe is defined as $T_\Sigma(\emptyset)$, that is the set of all ground terms in the language. In the following, we denote by $\equiv$ the syntactic equivalence of objects.

The set of substitutions, with typical elements $\sigma$, $\vartheta$, ... consists of all the mappings $\vartheta$ from $Var$ into $T_\Sigma(Var)$, such that $\{x \in Var \mid \vartheta(x) \neq x\}$ is finite. $e$ denotes the empty substitution.

The application of a substitution $\vartheta$ to a term $t$ (denoted $t\vartheta$) is defined as the term obtained by replacing each variable $x$ in $t$ by $\vartheta(x)$. The composition $\vartheta\sigma$ of $\vartheta$ and $\sigma$ is defined as function composition. We recall that the composition is associative and for each term $t$, $t(\vartheta\sigma) = (t\vartheta)\sigma$. A renaming is a variable bijection. A substitution $\vartheta'$ is more general than $\vartheta$ ($\vartheta \subseteq \vartheta'$), iff there exists $\vartheta$ such that $\vartheta = \vartheta'\sigma$. The notion of unification can be given with respect to a set of equations. An equation is an expression of the form $t = u$, where $t, u \in T_\Sigma(Var)$. Given a set $E$ of equations, $E$ is unifiable iff there exists $\vartheta$ such that $\forall (t = u) \in E$, $(t\vartheta) = (u\vartheta)$. We denote by $\text{Unif}(E)$ the (possibly empty) set of unifiers of $E$. We denote by $\text{mgu}(E)$ the set

$$\{\vartheta \in \text{Unif}(E) \mid \forall \vartheta' \in \text{Unif}(E), \vartheta \subseteq \vartheta'\}.$$ 

It is well known that all the most general unifiers of a set $E$ are equivalent [22].

Consider a program $P$ and the set $\Sigma_P$ of constructors of the first-order language underlying $P$. The extended Herbrand Universe $U_P$ is defined as $T_{\Sigma_P}(Var)/\sim$, i.e. the set of equivalence classes with respect to the variance relation $\sim$ ($t_1 \sim t_2$ iff $\exists \vartheta_1, \vartheta_2$ such that $t_1\vartheta_1 = t_2 \land t_2\vartheta_2 = t_1$). Together with an added least element, it is a complete lattice with respect to the usual preorder on terms $\leq$, such that $t_1 \leq t_2$ iff $\exists \vartheta \mid t_1 = t_2\vartheta$.

Let $\Pi_P$ be the set of predicate symbols in the program $P$. An atom is an object of the form $p(t_1, \ldots, t_n)$ where $t_1, \ldots, t_n \in T_{\Sigma_P}(Var)$ and $p$ is an $n$-ary predicate symbol (i.e. $p \in \Pi_P$). We denote the set of atoms as $Atoms$. In the following, given two atoms $A = p(t_1, \ldots, t_n)$ and $A' = p(t'_1, \ldots, t'_n)$, we denote by $\text{mgu}(A, A')$ an element in $\text{mgu}\{\{t_i = t'_i \mid i = 1, \ldots, n\}\}$. We extend this notation to any tuple of atoms.

A clause is a formula of the form $H :- B_1, \ldots, B_n$ with $n \geq 0$ where $H$ (the head) and $B_1, \ldots, B_n$ (the body) are atoms and ":-" and "\," denote logic implication and conjunction respectively. The variables are assumed to be universally quantified. If the body is empty, the clause is a unit clause. A goal is a formula $B_1, \ldots, B_n$ (denoted
A logic program is a finite set of clauses.

Given a term \( t \in T_2(Var) \), the set of variables occurring in \( t \) is denoted by \( \text{vars}(t) \) (in the following, with abuse of notation, we apply the \( \text{vars} \) operator to atoms and clauses as well). We extend also the notion of variance \( \sim \) to any syntactic object (i.e. terms, atoms, clauses, etc.). For a syntactic object \( s \) and a set of equivalence classes modulo renaming of objects \( I \), we denote by \( \langle c_1, \ldots, c_n \rangle \prec_I I \) that \( c_1, \ldots, c_n \) are representatives of elements of \( I \) renamed apart from \( s \) and from each other. Namely, that \( [c_i]_E \in I \), \( \text{vars}(c_i) \cap \text{vars}(s) = \emptyset \) (\( i = 1..n \)); and that for \( i \neq j \), \( \text{vars}(c_i) \cap \text{vars}(c_j) = \emptyset \) (\( i = 1..n, j = 1..n \)). The powerset of a set \( X \) is denoted by \( \mathcal{P}(X) \). If \( X \) is a set, the corresponding set of \( n \)-tuples of elements in \( X \) is denoted \( X^n \).

If the following, an extended notion of interpretation is given \([13, 14]\). The extended interpretations are more expressive than Herbrand interpretations because of the use of more syntactic objects (like variables) in the semantic domains.

Analogously to the standard semantics, the base of interpretations \( B_P \) is defined as \( \text{Atoms} / \sim \). An extended interpretation (interpretation) \( I \) is a subset of \( B_P \) (it represents the set of atoms which are true in \( I \)).

Definition 3.1 \([13]\). An interpretation \( I \) is a model for the logic program \( P \) iff every clause of \( P \) is true in the Herbrand interpretation \( G(I) \) (i.e. \( G(I) \) is a Herbrand model for \( P \)), where \( G(I) \) represents the set of ground instances of atoms in \( I \).

In positive logic programs, the operational semantics is a set of possibly non-ground atoms. The notion of model is general enough to capture the observable operational behavior of a logic program as a model. Moreover, every Herbrand model is still a model in the extended context.

We introduce now an immediate consequences operator \( T_P \) on interpretations whose least fixpoint is an extended model (model) which is able to capture computed answer substitutions.

Definition 3.2 (Immediate consequences operator) \([11, 12]\). Given a logic program \( P \), the operator \( T_P \) on the set of interpretations associated with \( P \) is defined as follows

\[
T_P(I) = \left\{ A' \in B_P \mid \begin{array}{l}
C \equiv A:=-B_1, \ldots, B_n \in P \\
\langle B'_1, \ldots, B'_n \rangle \prec_c I \\
\theta = \text{mgu}((B_1, \ldots, B_n), (B'_1, \ldots, B'_n)) \\
A' = A \theta
\end{array} \right\}.
\]

This definition is different with respect to the standard ground \( T_P \) operator \([33]\). It derives possibly non-ground atoms by means of a bottom-up inference rule which is based on unification, as in the top-down SLD resolution. As usual, \( T_P \) is a continuous function on the complete lattice of interpretations ordered by set-inclusion \([12]\). In the following we use \( T_P^\alpha \uparrow \) as a notation for \( \bigcup_{\alpha \in \omega} T_P^\alpha(\theta) \), where \( \alpha \) is a set of finite ordinals and \( \omega \) denotes the set of all the finite ordinals. A fixpoint
characterization of the program semantics is $lfp(T_P) = T_P \uparrow \omega$. We observe that each interpretation $I$ such that $T_P(I) \subseteq I$ is a model for $P$, thus $lfp(T_P)$ is a model as well.

The two most important results which characterize the relation between the new fixpoint semantics and the operational one (the equivalence between fixpoint and operational semantics) are the strong soundness and the strong completeness theorems given below. They show that $lfp(T_P)$ is the fully abstract semantics with respect to computed answer substitutions. Previous attempts of defining bottom-up abstract interpretations failed on nontrivial analyses (like mode analysis and polymorphic type inference) since they were based on an immediate consequences operator leading to a fixpoint semantics which did not model computed answer substitutions.

In the following $\overset{\Delta}{\rightarrow}^* \Box$ denotes a refutation for a given goal $G$ in $P$, with answer substitution $\vartheta$, while $\vartheta|_G$ is the restriction of the substitution $\vartheta$ to the variables occurring in $G$, extended as an identity for each variable $x \in vars(G)$ such that $\vartheta(x)$ is undefined. We extend the notation $\vartheta|_s$ to any syntactic objects $s$.

**Theorem 3.3** (Strong Soundness) [11, 12]. Let $P$ be a logic program and let $G$ be a goal $:-B_1, \ldots, B_n$. Assume that $G \overset{\Delta}{\rightarrow}^* \Box$, then

$$\exists (B'_1, \ldots, B'_n) \prec_G lfp(T_P) \quad \text{and} \quad \exists \vartheta' = \text{mgu}((B_1, \ldots, B_n), (B'_1, \ldots, B'_n))$$

such that $\vartheta'|_G = \vartheta|_G$.

**Theorem 3.4** (Strong Completeness) [11, 12]. Let $P$ be a logic program and let $G$ be a goal $:-B_1, \ldots, B_n$. If

$$\exists (B'_1, \ldots, B'_n) \prec_G lfp(T_P) \quad \text{and} \quad \exists \vartheta' = \text{mgu}((B_1, \ldots, B_n), (B'_1, \ldots, B'_n)),$$

then $\exists \vartheta$ such that $G \overset{\Delta}{\rightarrow}^* \Box$ and $\vartheta|_G = \vartheta|_G$.

This semantics, due to its relation with the operational one, has been used in some bottom-up abstract interpretations frameworks [3, 7, 20]. Since we will introduce the notion of well-typed derivation for a goal, the correctness of our analysis will be proved with respect to a variant of the previous concrete semantics.

### 3.3. Bottom-up abstract interpretation

In the following we recall some of the basic concepts developed in [3]. This framework is particularly suitable for semantics-based polymorphic type inference in logic programming, due to the use of possibly non-ground semantic objects.

The basic idea is to abstract the concrete immediate consequences operator $T_P$ to obtain an abstract operator whose fixpoint is finitely computable and is a correct approximation of the concrete bottom-up semantics of the program. This is achieved by defining an abstract domain of interpretations, based on the notion of abstract atom. The construction of the domain of abstract interpretations follows a step-by-step approach. The main problem is what we want to observe from the concrete data behaviour and how the resulting abstract data objects are represented.
in order to have an efficient (concise) data-flow analysis. In order to suitably define an abstract domain for a given program analysis, three abstraction levels are presented, each one related with a different data-property. In the first one, the abstraction is performed on terms in order to summarize all and only those properties which are useful for the “term analysis”. The second step deals with how abstract terms are collected together (in abstract atoms) and which properties of their union the analysis is interested in. Finally, in the third step, the notion of abstract interpretation is introduced. It corresponds to choosing which properties we want to observe, collecting together different abstract atoms [3].

In type inference the first abstraction level will correspond to abstracting a given term returning its type. A corresponding domain of abstract atoms will be considered for the second one. As for the third abstraction level, we will consider abstract interpretations as sets of abstract atoms. In the following we will consider a simplified instance of the framework in [3], which is general enough to define our type inference system.

Let \( \mathcal{H} \) be a domain of abstract terms and \( \mathcal{B}_P \) the set of (abstract) atoms defined on \( \mathcal{H} \). Let \( (\mathcal{P}(\mathcal{B}_P), \subseteq) \) be the resulting abstract domain of interpretations, specified by means of a Galois insertion \((\alpha, \gamma)\) of \( (\mathcal{P}(\mathcal{B}_P), \subseteq) \) into \( (\mathcal{P}(\mathcal{B}_P), \subseteq) \). We assume that \( (\mathcal{P}(\mathcal{B}_P), \subseteq) \) is a finite lattice.

The definition of an abstract immediate consequences operator \( T_P : \mathcal{B}_P \rightarrow \mathcal{B}_P \), abstracting \( T_P \) is given in terms of a set of abstract operators, namely:

- **Abstract unification**, \( \alpha\text{-mgu} : B^p_\alpha \times B^p_\alpha \rightarrow (\text{Var} \rightarrow \mathcal{H}) \), plays the fundamental role of passing the information from the abstract to the concrete context, by returning an abstract substitution (i.e. a binding of concrete variables to abstract terms) defined on the set of variables of the concrete atoms.
- **Abstract substitution application**, \( \alpha\text{-apply} : B_P \times (\text{Var} \rightarrow \mathcal{H}) \rightarrow \mathcal{B}_P \), applies an abstract substitution to a concrete atom returning an abstract atom.

The abstract immediate consequences operator is then:

\[
T_P(I^\alpha) = \begin{cases} 
\alpha\text{-apply}(A, \theta^\alpha) & C \equiv A \vdash B_1, \ldots, B_n \in P \\
\langle B^\alpha_1, \ldots, B^\alpha_n \rangle \ll c I^\alpha & \theta^\alpha = \alpha\text{-mgu}((B_1, \ldots, B_n), (B^\alpha_1, \ldots, B^\alpha_n)) \\
\theta^\alpha \neq \text{Fail} & \end{cases}
\]

where \( I^\alpha \in \mathcal{P}(\mathcal{B}_P) \) and \( \ll \) extends in the obvious way on abstract variables (if any) as well as concrete ones.

4. Type inference

In this section we present a polymorphic type inference based on the above abstract interpretation scheme. In the first subsection we introduce type declarations. In the second one we present the domain of types. In the third subsection we present
an approximation of the domain of types. The fourth subsection introduces a set
of abstract operators which can be used to define a correct abstract immediate
consequences operator for polymorphic type inference. The proofs of the basic
properties of the abstract semantics are in Appendix A. Finally, in the last subsection
we show some examples.

4.1. Type declarations

In order to build the domain of types we use type declarations to reduce each
term to its type. The choice of having type declarations is motivated both by the
need to have a concise description of the success set (as suggested in [34]) and by
the advantages given by type variables [28]. Unlike other approaches, multiple
occurrences of the same type variable allow to specify that the type of components,
although not specified, must be the same. A type declaration is a set of rules grouped
in two possible constructs, simple type declaration and parametric type declaration:

- simple type declaration:

  type \( tname : \)
  
  \[
  c \rightarrow tname \\
  \ldots \\
  f(\tau_1, \ldots, \tau_m) \rightarrow tname
  \]
  
  end

- parametric type declaration:

  type \( tname(\xi_1, \ldots, \xi_n) : \)
  
  \[
  c \rightarrow tname(\perp, \ldots, \perp) \\
  \ldots \\
  f(\tau_1, \ldots, \tau_m) \rightarrow tname(\xi_1, \ldots, \xi_n)
  \]
  
  end

Each type declaration begins with the type name it defines; a parametric type is a
type name applied to variables \( (\xi_i) \) ranging on types.

The left part of each rule is either a constant or the application of a function
symbol to types \( \tau_i \). A type is a simple type name, a variable, or a parametric type
name applied to types. A parametric type declaration cannot consist of constants
only.

The right part of each rule defining a simple type is the type name itself, while
the right part of a rule defining a parametric type is the application of the type
name either to a sequence of variables or to a sequence of \( \perp \). \( \perp \) stands for the
undefined type, and \( tname(\perp, \ldots, \perp) \) denotes the type associated with any constant
in a parametric type declaration. Intuitively, the meaning of \( tname(\perp, \ldots, \perp) \) is
that the type associated with a constant of a parametric type is partially undefined:
it will be completely known only when used in a more structured object. The reason
of this choice will be clear in the following, let us only mention that a similar choice
is adopted in [19] (\( \emptyset \) replaces \( \perp \)) although with a different underlying intuition.
Type declarations can be used as a set of reduction rules to reduce terms to types. In order to assign only one type to a term, there are no two rules with unifiable left parts.

Reduction rules induce a set of equivalence classes on the universe of terms: two terms belong to the same class if and only if they are reducible to the same type or they are (both) not reducible to a type (i.e. they are not well-typed).

Finally, to be useful, type declarations must be complete for the logic program we want to study, i.e. each constant or function symbol occurring in the program must occur in a rule left part.

Example 1. As an example consider the definition of the type nat and the type list, a simple and a parametric type declaration respectively:

```
type nat :  
  0 ⇒ nat  
  s(nat) ⇒ nat  
end

type list(α) :  
  nil ⇒ list(⊥)  
  cons(α, list(α)) ⇒ list(α)  
end
```

Type declarations are defined on a set of type constructors denoted \(Ω\) and on a denumerable set of type variables \(Var\), such that \(Var \cap Var_r = \emptyset\). Let \(T_Ω(Var_r)\) denote the set of type terms inductively defined on \(Ω\) and \(Var_r\). In the following we denote (also inside programs) variables ranging on types (in \(Var_r\)) by Greek letters \(α, β, ζ, \ldots\) and arbitrary types (in \(T_Ω(Var_r)\)) by \(τ, τ', τ''\), \ldots Moreover, we denote by \(RR'\) the set of reduction rules of simple and parametric type declarations:

\[RR' = RR'_{simple} \cup RR'_{parametric} \]

4.2. The domain of types

Starting from type declarations, we define a standard methodology to construct the domains of types for a type inference system. The domain of types is built by typing the terms of the concrete one. Usually typing is performed by means of inference rules; in our case this role is played by the reduction rules obtained from type declarations. Thus, the domain of types is obtained by reducing terms to types by means of \(RR'\). An important point is that our system, given the presence of \(⊥\), and type variables, can infer different types for terms which are supposed to have the same type structure. The following two examples clarify the problem:

- Consider the term \(\text{cons}(0, \text{nil})\). The reduction process leads to \(\text{cons}(\text{nat, list}(⊥))\) which is not further reducible. To force the reduction we normalize two different types to the most descriptive of them when possible. In this case since \(\text{nat}\) is more descriptive than \(⊥\), \(\text{cons}(\text{nat, list}(⊥))\) is forced to \(\text{cons}(\text{nat, list(nat)})\) and then reduced.
Consider now \( \text{cons}(0, \text{cons}(X, Y)) \). This term can be reduced to \( \text{cons}({\text{nat}}, \text{list}(\alpha)) \) and further reduced to \( \text{list}(\text{nat}) \) by substituting \( \alpha \) with \( \text{nat} \). This means that a type name is more descriptive than a variable, or, in different words that we can know the type of a list by knowing the type of at least one element (a discussion on this assumption is presented in Section 5).

These examples show that typing is performed by both a reduction and a normalization process. These two processes can be performed by means of a simple deterministic program. Since the application of reduction rules needs unification, we use a Prolog program \( P' \) to perform reductions. This program derives directly from type declarations and type descriptivity and it is defined on the universe of terms extended with type terms. The definition is given by two predicates: Type, which is responsible of the reduction of terms to types, and Norm, which normalizes two types to the most descriptive, when possible. Notice that only type-correct terms can be reduced to their types. Let \( P' \) denote the following Prolog program:

\[
\{ \text{Type} (X, X) :- \text{Var}(X), !. \} \cup \\
\{ \text{Type}(\text{type}, \text{type}) :- !., \text{type is a simple or parametric type} \} \cup \\
\{ \text{Type}(\text{c}, \text{tname}). \mid c \rightarrow \text{tname} \in \text{RR}\_\text{simple}\} \cup \\
\{ \text{Type}(f(X_1, \ldots, X_m), \text{tname}) :- \text{Type}(X_1, \tau_1), \ldots, \text{Type}(X_m, \tau_m). \mid f(\tau_1, \ldots, \tau_m) \rightarrow \text{tname} \in \text{RR}\_\text{simple} \} \cup \\
\{ \text{Type}(\text{c}, \text{tname}(\bot, \ldots, \bot)). \mid c \rightarrow \text{tname}(\bot, \ldots, \bot) \in \text{RR}\_\text{parametric} \} \cup \\
\{ \text{Type}(f(X_1, \ldots, X_m), \text{tname}(\xi_1, \ldots, \xi_n)) : \}
\quad \text{Type}(X_1, \tau_1), \ldots, \text{Type}(X_m, \tau_m), \ldots, \\
\quad \text{Norm}(W_1, W_2, Z_1, \ldots, \text{Norm}(Z_k, W_k, \xi), \ldots, \}
\quad f(\tau_1, \ldots, \tau_m) \rightarrow \text{tname}(\xi_1, \ldots, \xi_n) \in \text{RR}\_\text{parametric}, \text{ and the terms } \tau_1', \ldots, \tau_m' \}
\quad \text{are obtained from } \tau_1, \ldots, \tau_m \text{ by replacing all the } k_i \text{ occurrences of each variable } \xi_i \text{ by } k_i \text{ fresh variables } W_s, s = 1..k_i.
\]

The first two sets of program clauses handle type variables and declared types (the type is given by themselves). The clauses associated with simple type declarations either give type to constants or recursively apply the predicate Type to the components of a term. The clauses associated with parametric type declaration do the same, the only difference is that in a clause corresponding to a rule \( f(\tau_1, \ldots, \tau_m) \rightarrow \text{tname}(\xi_1, \ldots, \xi_n) \), all the occurrences of a type variable \( \xi_i \) \((i = 1, \ldots, m) \) in \( \tau_1, \ldots, \tau_m \) are replaced by fresh variables \( W \). The predicate Norm returns the most descriptive type among the answers computed for the \( W \)'s (that is among the ones computed for the same variable \( \xi \)). The definition of the predicate Norm, computing in its third argument the most descriptive type between the first two, is
given as follows:

$\{ \text{Norm}(X, X, X) := \text{Var}(X), !. \} \cup$

$\{ \text{Norm}(X, Y, X) := Y = = \perp, !. \} \cup$

$\{ \text{Norm}(Y, X, X) := Y = = \perp, !. \} \cup$

$\{ \text{Norm}(\text{name}(x_1, \ldots, x_n), \text{name}(z_1, \ldots, z_m), \text{name}(\xi_1, \ldots, \xi_m)) := \}$

$\{ \text{Norm}(\xi_1, \xi_2, \xi_3, \ldots, \text{Norm}(\xi_{1,n}, \xi_{2,n}, \xi_{3,n}), !. \}$

$\text{name}(\xi_1, \ldots, \xi_m) \text{ is a parametric type.}$

$\{ \text{Norm}(X, X, X) \}.$

The first clause says that the most descriptive type between two variables is a variable, while the next two state that $\perp$ is less descriptive than any other type. A recursive definition of $\text{Norm}$ is associated with each parametric declared type name. In this case the computation is recursively performed on the type structure. The last clause states that the most descriptive type between a type name and a variable is the type name itself, and that two different type names have not a most descriptive type.

**Example 2.** Let us consider the type declarations given in Example 1. The associated logic program $P'$ is:

$\text{Type}(X, X) := \text{Var}(X), !.$

$\text{Type}(\text{nat}, \text{nat}) := !.$

$\text{Type}(\text{list}(X), \text{list}(X)) := !.$

$\text{Type}(0, \text{nat}).$

$\text{Type}(s(X), \text{nat}) := \text{Type}(X, \text{nat}).$

$\text{Type}(\text{nil}, \text{list}(\perp)).$

$\text{Type}(\text{cons}(X, Y), \text{list}(W)) := \text{Type}(X, Z), \text{Type}(Y, \text{list}(T)), \text{Norm}(Z, T, W).$

$\text{Norm}(X, X, X) := \text{Var}(X), !.$

$\text{Norm}(X, Y, X) := Y = = \perp, !.$

$\text{Norm}(Y, X, X) := Y = = \perp, !.$

$\text{Norm}(\text{list}(X), \text{list}(Y), \text{list}(Z)) := \text{Norm}(X, Y, Z), !.$

$\text{Norm}(X, X, X).$

Using $P'$ the type of the term $\text{cons}(\text{nil}, \text{cons}(\text{cons}(s(0), \text{nil}), \text{nil}))$ can be obtained as the computed answer substitution for the goal $\text{Type}(\text{cons}(\text{nil}, \text{cons}(\text{cons}(s(0), \text{nil}), \text{nil})), X')$ that is $\{ \text{list(list(nat))}/X' \}.$

Thus, the Prolog program $P'$, associated with a set of type declarations for a logic program $P$, gives a procedural method to compute the type for any term belonging to the universe of terms of $P$.

As in the concrete case, the notion of "equivalence up to renaming" can be also applied to type terms.
Definition 4.1. A term $t \in T_2(\text{Var})$ is reducible to the type $\tau$ ($t$ has type $\tau$) iff

$$ :- \text{Type}(t, X) \xrightarrow{\theta} \tau, \Box \quad \text{and} \quad \vartheta(X) \leftarrow \tau. $$

If $:- \text{Type}(t, X) \rightarrow \Box \text{Fail}$, then $t$ has type ERR.

Thus, given a term $t$ belonging to the universe of terms associated with the program $P$, if $t$ is reducible by means of $P^r$ to the type symbol $\tau$, the term is well-typed and its type is denoted by $\tau$. Note also that if the given term is not reducible to any type name, its type is denoted by ERR. In the following we denote $T_\Omega(\text{Var}_r) \cup \{\text{ERR}\}$ as $T_\Omega(\text{Var}_r)$.

Theorem 4.2. Consider a goal of the form $:- \text{Type}(t, t')$ and the Prolog program $P^r$. The computation of the goal terminates and, in case of success, it gives a unique computer answer substitution.

Proof. It follows from the definition of $P^r$. $\square$

Thus, $P^r$ completely characterizes the type of terms. Let us consider the equivalence relation $\equiv_{\tau}$ over the universe $T_2(\text{Var})$ induced by the program $P^r$, such that $\forall t_1, t_2 \in T_2(\text{Var})$, $t_1 \equiv_{\tau} t_2$ iff

$$ :- \text{Type}(t_1, \alpha_1) \xrightarrow{\theta} \tau, \Box \land :- \text{Type}(t_2, \alpha_2) \xrightarrow{\theta} \tau, \Box \land \vartheta_{|\alpha_1} \leftarrow \sigma_{|\alpha_2}, $$

or

$$ :- \text{Type}(t_1, \alpha_1) \rightarrow \Box \text{Fail} \land :- \text{Type}(t_2, \alpha_2) \rightarrow \Box \text{Fail}. $$

The quotient set $T_2(\text{Var})/\equiv_{\tau}$ represents the set of all equivalence classes each of them containing terms having the same type (in the following the equivalence class of a term $t$ (the type of $t$) will be denoted by $[t]_{\equiv_{\tau}}$).

Definition 4.3. A type substitution (denoted by $\theta^r$) is a mapping from a finite set of type variables to $T_\Omega(\text{Var}_r)$.

Definition 4.4. A $\perp_r$-substitution (denoted by $\eta^{+r}$) is a set of subterm descriptions $\{d_1^r, \ldots, d_n^r\}$. Each subterm description is a finite sequence of natural numbers identifying a specific subterm.

For example the sequence $(3, 2)$ means "the subterm which is the second argument of the third argument" of the original term.

Definition 4.5. The application of a $\perp_r$-substitution $\eta^{+r}$ to a type term $\tau$ results in a term $\tau'$ in which all the subterms described in $\eta^{+r}$ (and occurring in $\tau$) are replaced by $\perp_r$. 
For example the application of $\eta^+=\{1,1\}$ to the type term $\text{list}$(list(list(nat))) results in $\text{list}$(list($\bot$)).

**Definition 4.6** (Preordering relation among type terms). Given two type terms $\tau_1$ and $\tau_2$, $\tau_1<\tau_2$ if either $\tau_2=\text{ERR}$ or there exist $\varphi$, (type substitution) and $\eta^+$ (substitution of $\bot$, for terms) such that $\tau_1\sim(\tau_2\varphi)\eta^+$.

For example the type term $\tau_1=\text{list}(\bot)$ is smaller than $\tau_2=\text{list}$(list(nat)) because there exist $\varphi=\{}$ and $\eta^+=\{1\}$ such that $(\tau_2\varphi)\eta^+=\text{list}$(list($\bot$)). In the following, with abuse of notation, we use $<\tau$ as an ordering relation on $\tau_{\Omega}$(Var). We denote by $\sim$, the equivalence relation induced by the symmetric closure of $<\tau$. Notice that $\tau_1\sim\tau_2$ iff $\tau_1\sim\tau_2$. The least upper bound with respect to $<\tau$ is denoted $\sqcup$. 

**Theorem 4.7.** There exists $\mathfrak{D}\subseteq\tau_{\Omega}$(Var) such that $\mathfrak{D}/\sim$, and $\tau_{\Sigma}$(Var)/$\sim$, are isomorphic.

**Proof.** It follows by defining $\mathfrak{D}/\sim$, as:

$$\left\{ [\tau] | \begin{array}{l} t\in\tau_{\Sigma}$(Var) \\ \vdash\text{Type}(t,\tau)\rightarrow^*\mathfrak{D} \end{array} \right\} \cup \{\text{ERR}\}.$$

The mapping from $\tau_{\Sigma}$(Var)/$\sim$, to $\mathfrak{D}/\sim$, given by the program, is injective by Theorem 4.2 and by definition of $=\mathfrak{D}$. Surjectivity follows by definition.  

Thus, any equivalence class $[t]_{\sim}$, can be represented as a term in $\tau_{\Omega}$(Var)/$\sim$, denoting the type of the term. $\text{ERR}$ denotes the equivalence class of terms with a wrong type structure with respect to the type declarations. Since all the definitions we consider are independent on any particular choice of the elements in the equivalence classes modulo $\sim$, we will denote by $\tau$ the equivalence class $[\tau]_{\sim}$.

**Theorem 4.8.** For each $\tau_1,\tau_2\in\mathfrak{D}/\sim$, $\tau_1\sqcup\tau_2\in\mathfrak{D}/\sim$.

**Proof.** We need the following recursive definition of terms corresponding to a type $\tau$ in a term $t$: denoted $\{t\}$:

- $\{t\} = \{t\}$ if $[t]_{\sim} = \tau$ (t has type $\tau$);
- $\{t\} = \emptyset$ if t has simple type $\tau$ and $\tau \neq \tau$, or t has type $\phi(\bot)$ and $\tau \neq \phi(\bot)$;
- $\{f(t_1,\ldots,t_m)\}_\tau = \bigcup^n_{i=1} \{t_i\}_\tau$ otherwise.

The theorem is proved by constructing a term $t$ with type $\tau = \tau_1\sqcup\tau_2$, given $t_1$ and $t_2$ with type $\tau_1$ and $\tau_2$.

By definition let $i=1,2$ and let $\varphi_i$ and $\eta^+_i$ be such that $\tau_1 = (\tau\varphi_i)\eta^+_i$ and $\tau_2 = (\tau\varphi_i)\eta^+_i$. Let us consider the terms $\bar{t}_i$ and $\bar{t}_i$ obtained by substituting each term in $\{t_i\}$ with a fresh variable $X$, where $\{\bar{t}_i\}_\tau$ $\in\varphi_i$. Each $\eta^+_i$ identifies a class of subtypes in $\tau$, such that the corresponding type in $\tau_i$ is a constant in a parametric type definition: $\text{name}(\tau_\bot)$. The term $t$ is constructed by taking either $\bar{t}_1$ or $\bar{t}_2$ (consider $\bar{t}_i$) and by substituting each term in $\{\bar{t}_i\}_\text{name}(\tau_\bot)$ with a term such that the type of the resulting term is $\tau$. These terms exist since $\bar{t}_2$ does contain them by definition of lub. \qed
Definition 4.9 (Type patterns). An $n$-type pattern is an $n$-tuple $(\tau_1, \ldots, \tau_n)$ of type terms.

The previous notions of type reduction and ordering extend naturally on any tuples of (concrete) terms and on type patterns respectively. With abuse of notation, the domain of $n$-type patterns $\mathcal{D}^n/\perp$ is denoted $\mathcal{D}^n$ and by $\perp^n$ we denote the $n$-type pattern $(\perp, \ldots, \perp)$.

Proposition 4.10. Let $n$ be a finite natural number. $\mathcal{D}^n \cup \{\perp^n\}$ is an infinite complete lattice with respect to the $\leq_e$ ordering relation.

Proof. It is a straightforward consequence of the previous theorem. $\square$

Example 3. Let us consider the type definitions given in Example 1, the complete lattice of types is given in Fig. 1.

4.3. Abstract domain

The above domain of type patterns is characterized by an infinite number of elements. In order to obtain an abstract domain usable in abstract interpretation, we have to define a finite approximation of this infinite lattice of types.

Let us consider a recursive function $\text{depth}: T_\Omega(\text{Var},) \to \omega$ such that $\forall \tau \in T_\Omega(\text{Var})$:

1. $\text{depth}(\tau) = 0$ iff $\tau$ is a type variable, $\perp$, or a simple (non-parametric) type,
2. $\text{depth}(\tau) = 1 + \max\{\text{depth}(\tau_1), \ldots, \text{depth}(\tau_n)\}$ iff $\tau = \phi(\tau_1, \ldots, \tau_n)$, where $\phi$ is a parametric type name.

\[ \text{ERR} \]
\[ \perp \]
\[ \text{nat} \]
\[ \text{list(\text{nat})} \]
\[ \text{list(\text{list(\text{nat})})} \]
\[ \ldots \ldots \]
\[ \text{list(\perp)} \]

Fig. 1. The lattice of types: $\mathcal{D}$. 
A finite domain of type patterns is obtainable by extending the depth notion on type patterns and by cutting at depth $k$ the corresponding infinite lattice of types. Thus, given a positive integer $k$, let us define $\mathcal{D}_k^n \subseteq \mathcal{D}^n$ as follows:

**Definition 4.11.** $\mathcal{D}_k^n = \text{Cut}_k(\mathcal{D}^n)$, where $\text{Cut}_k((\tau_1, \ldots, \tau_n))$ is the set of type patterns $(\tau_1^k, \ldots, \tau_n^k)$ such that $\tau_i^k$ is obtained by substituting with a fresh variable each subtype $\tau'$ in $\tau$ such that $\text{depth}(\tau) - \text{depth}(\tau') = k$.

**Proposition 4.12.** $\mathcal{D}_k^n$ is a finite complete lattice with respect to the $\leq_r$ ordering relation.

**Example 4.** The finite complete lattice of types corresponding to the previous infinite one, with $n = 1$ and $k = 3$, is given in Fig. 2.

We can now consider the approximated lattice of type patterns as the basis to construct abstract atoms.

**Definition 4.13.** $\alpha_r : T_2(\text{Var})^n \rightarrow \mathcal{D}_k^n$ is such that

$$
\forall \bar{t} \in T_2(\text{Var})^n, \quad \alpha_r(\bar{t}) = \text{Cut}_k([\bar{t}]_{\approx_r}).$

![Fig. 2. The approximated lattice of types for $k = 3$: $\mathcal{D}_3^1$.](image)
The domain of abstract \( \tau \)-interpretations is composed by atoms defined on type patterns. The abstract base \( \mathcal{B}^k_p \) is defined as follows:

**Definition 4.14** (Base of abstract \( \tau \)-interpretations).

\[
\mathcal{B}^k_p = \left\{ \frac{p(t_1, \ldots, t_n)}{p \in \Pi^k_p, (t_1, \ldots, t_n) \in \mathcal{D}^k_p} \right\}.
\]

Thus an abstract atom is a predicate symbol in the language applied to a type pattern.

A well-typed atom is an abstract atom having no ERR types in the corresponding pattern. The notion of well-typed atoms extends naturally to any syntactic object.

**Definition 4.15.** An abstract \( \tau \)-interpretation \( I^* \) for the polymorphic type inference is any subset of the abstract base.

**Proposition 4.16.** \( (\mathcal{P}(\mathcal{B}^k_p), \subseteq) \) is a finite lattice.

The relation between the concrete (infinite) lattice of (standard) interpretations, \( (\mathcal{P}(B_p), \subseteq) \) and \( (\mathcal{P}(\mathcal{B}^k_p), \subseteq) \), is established by a Galois insertion.

- Let \( \alpha : (\mathcal{P}(B_p), \subseteq) \rightarrow (\mathcal{P}(\mathcal{B}^k_p), \subseteq) \) be such that:

\[
\alpha(I) = \left\{ \frac{p(\Psi)}{p(t_1, \ldots, t_n) \in I, \Psi = \alpha_r(t_1, \ldots, t_n)} \right\}.
\]

- Let \( \gamma : (\mathcal{P}(\mathcal{B}^k_p), \subseteq) \rightarrow (\mathcal{P}(B_p), \subseteq) \) be such that

\[
\gamma(I^*) = \bigcup \{ I \mid \alpha(I) \subseteq I^* \}.
\]

**Theorem 4.17.** The pair \( (\alpha, \gamma) \) is a Galois insertion of \( (\mathcal{P}(\mathcal{B}^k_p), \subseteq) \) into \( (\mathcal{P}(B_p), \subseteq) \).

**Proof.** Notice that \( \alpha \) is additive: let \( D \subseteq \mathcal{P}(B_p) \) be a possibly denumerable collection of concrete interpretations \( \{ I_1, \ldots, I_n \} \).

\[
p(\Psi) \in \alpha(\bigcup D)
\]

iff \( \exists p(t_1, \ldots, t_n) \in \bigcup D \land \Psi = \alpha_r(t_1, \ldots, t_n) \)

iff \( \exists I \in D \) such that \( p(t_1, \ldots, t_n) \in I \land \Psi = \alpha_r(t_1, \ldots, t_n) \)

iff \( p(\Psi) \in \bigcup_{I \in D} \alpha(I) \).

- \( \alpha \)- and \( \gamma \)-monotonicity follow by definition.
- By \( \alpha \)-additivity:

\[
\alpha(\gamma(I^*)) = \alpha(\bigcup \{ I \mid \alpha(I) \subseteq I^* \}) = I^*.
\]

- \( \gamma(\alpha(I)) \subseteq I \) follows by definition. \( \square \)
4.4. Abstract interpretation and type inference

In this section we analyze the abstract operators involved in the definition of the associated abstract interpretation. Following the framework in [3], we define an abstract unification operator and a type substitution application. These operators allow to define an abstract immediate consequence operator whose least fixpoint is a correct approximation of the concrete success set.

**Definition 4.18 (Abstract unification).** Let \( p(t_1, \ldots, t_n) \) and \( p(\tau_1, \ldots, \tau_n) \) be respectively a concrete and an abstract atom. We define the goal:

\[
G' = \{ :- \text{Type}(t_1, \tau_1), \ldots, \text{Type}(t_n, \tau_n) \}.
\]

The abstract unification is defined as follows:

\[
\alpha\text{-mgu}(p(t_1, \ldots, t_n), p(\tau_1, \ldots, \tau_n)) =
\begin{cases}
\theta^*, & \text{if } G' \models^* \emptyset \land \emptyset = \emptyset_{t_1, \ldots, t_n}, \\
\text{Fail}, & \text{otherwise}.
\end{cases}
\]

\( \alpha\text{-mgu} \) returns a type substitution defined on the set of variable symbols belonging to the concrete atom. It is easy to extend the previous definition to any number of concrete and abstract atoms to unify.

Let us consider now the \( \alpha\text{-apply} \) operator which "applies" a type substitution \( \theta^* \) to a concrete atom \( p(t_1, \ldots, t_n) \). By considering the logic program associated with the type definition section, we can give a procedural method to compute the operator.

**Definition 4.19 (Type substitution application).** Given a concrete atom \( p(t_1, \ldots, t_n) \) and a type substitution \( \theta^* \),

\[
\alpha\text{-apply}(p(t_1, \ldots, t_n), \theta^*) = \{ p(Cut_k((\tau_1, \ldots, \tau_n))) \}
\]

where \( \{ \tau_1/\xi_1, \ldots, \tau_n/\xi_n \} \) is the answer substitution computed in \( P^* \) by the goal

\[
\{ :- \text{Type}(t_1, \theta^*, \xi_1), \ldots, \text{Type}(t_n, \theta^*, \xi_n) \}.
\]

If the computation fails, \( \alpha\text{-apply} \) returns a type pattern having the extra type symbol \( \text{ERR} \) for each subgoal which fails (this actually requires a proof procedure quite different from the standard one, but it is easy to define a meta-interpreter to do it).

The computation of \( \alpha\text{-apply} \) is always finite. The result is the type configuration of the predicate arguments. Moreover \( \alpha\text{-apply} \) reports any type error occurring during the substitution application.

According to the bottom-up abstract interpretation framework in [3], we can specialize the general abstract transformation map associated with any logic program.
Definition 4.20. The mapping $\mathcal{T}_p: \mathcal{P}(\mathcal{B}_p^I) \rightarrow \mathcal{P}(\mathcal{B}_p^I)$ is defined as follows:

$$
\mathcal{T}_p(I^*) = \left\{ \begin{array}{l}
\alpha - \text{apply}(H, \emptyset^*) \\
C = H \leftarrow B_1, \ldots, B_n \in P \\
\langle B_1^*, \ldots, B_n^* \rangle \ll C I^* \\
\emptyset^* = \alpha - \text{mgu}((B_1, \ldots, B_n), (B_1^*, \ldots, B_n^*)) \\
\emptyset^* \neq \text{Fail}
\end{array} \right\}
$$

where $\ll$ extends naturally on type variables as well as concrete variables and selects only the well-typed atoms in $I^*$.

Theorem 4.21. $\mathcal{T}_p$ is monotonic on $(\mathcal{P}(\mathcal{B}_p^I), \subseteq)$.

Proof. Let $I_1^* \subseteq I_2^*$. With an abuse of notation let us denote by

$$
\alpha - \text{mgu}((B_1, \ldots, B_n), I)
$$

$$
= \left\{ \begin{array}{l}
\emptyset^* \\
\langle B_1^*, \ldots, B_n^* \rangle \ll C I \\
\emptyset^* = \alpha - \text{mgu}((B_1, \ldots, B_n), (B_1^*, \ldots, B_n^*)) \\
\emptyset^* \neq \text{Fail}
\end{array} \right\}
$$

Thus: $\alpha - \text{mgu}((B_1, \ldots, B_n), I_1^*) \subseteq \alpha - \text{mgu}((B_1, \ldots, B_n), I_2^*)$. The thesis follows by the definition of $\alpha$-apply. $\Box$

Corollary 4.22. Under the previous hypothesis, there exists a positive integer $h$ such that $\text{lfp}(\mathcal{T}_p^*) = \mathcal{T}_p^h$.

Let us denote by $SS_p^*$ the $\text{lfp}(\mathcal{T}_p^*)$. The meaning of our type system is more formally given by the following theorem, which establishes the connection between (concrete) successful derivations and the abstract success set $SS_p^*$.

Let us denote by $G \overset{\alpha}{\longrightarrow} \square_p$ a well-typed successful derivation, namely a sequence of goals $G_1, \ldots, G_m$ containing only well-typed atoms.

Theorem 4.23 (Soundness). Let $P$ be a logic program and let $G$ be a goal $:- B_1, \ldots, B_n$. If $G \overset{\alpha}{\longrightarrow} \square_p$, then there exists $\langle B_1^*, \ldots, B_n^* \rangle \ll G SS_p^*$ such that $\alpha - \text{mgu}((B_1, \ldots, B_n), \emptyset, (B_1^*, \ldots, B_n^*)) \neq \text{Fail}$.

Proof. The proof is given in Appendix A. $\Box$

$SS_p^*$ contains polymorphic type patterns for each predicate symbol. It provides a description of the success set in terms of well-typed terms. It is easy to observe that the greater the description limit $k$, the greater the type information associated with the abstract interpretation. A good choice of the description limit $k$ may be

$$
k > \max \{p\text{depth}(A) | A \text{ is an atom defined in } P\},
$$

where, given an atom $p(t_1, \ldots, t_n)$:

$$
p\text{depth}(p(t_1, \ldots, t_n)) = \max \{\text{depth}(t_i) | i \in \{1, \ldots, n\}\}.
$$
4.5. Examples

In this section we analyze some examples of type inference in HCL. In our $\mathcal{T}_p$-based analysis, all the types of success patterns are generated.

**Example 5.** Let us consider the following logic program:

```prolog
append(nil, X, X).
append(cons(X, Y), Z, cons(X, U)) :- append(Y, Z, U).
```

whose domain of types is built starting from the following type declarations:

```prolog
type nat:
  0 -> nat
  s(nat) -> nat

type list(a):
  nil -> list(\bot)
  cons(a, list(a)) -> list(a)
end
end
```

Using the definition (with $k=2$) we compute the abstract success set in two steps:

1. $G(0) = \{\text{append(W, P, P)}\}$
2. $G(G(0)) = \{\text{append(W, P, P), append(W, P, P), append(W, P, P)}\}$
3. $G(y > T; (0)) = \text{fixpoint}$.

In fact, from the first clause in the program we have $F; (0)$. To compute $F; (S; (0))$ we have to consider that

```prolog
\text{append(append(cons(X, Y), 4 cons(X, V), \{W/\}, P/P, P/U}) = \{append(Zist(l), list(t), list(t))}.\n```

The abstract success set gives the type structure of the predicate `append`, i.e. all the well-typed solutions of goals are instances of these type patterns. This example also points out a problem in this definition of `append`: there are possible solutions which are instances of the pattern `append(Zist(l), p, p)`. In fact the goal `append(nil, s(0), s(0))` is provable in contrast to the intuition. The following definition of `append` overcomes this problem.

```prolog
append(nil, nil, nil).
append(nil, cons(X, Y), cons(X, Y)) :- append(nil, Y, Y).
append(cons(X, Y), Z, cons(X, U)) :- append(Y, Z, U).
```

The abstract success set is

$$SS_p^- = \begin{cases} 
\text{append(list(\bot), list(\bot), list(\bot))} \\
\text{append(list(\bot), list(\alpha), list(\alpha))} \\
\text{append(list(\alpha), list(\bot), list(\alpha))} \\
\text{append(list(\alpha), list(\alpha), list(\alpha))} 
\end{cases}.$$  

This example shows that type inference can be used as a tool for verifying the relations between the implementation and the intuition about a program.
The following example shows the approximation level introduced by depth-\( k \) abstraction.

**Example 6.** Let us consider the logic program:

\[
\begin{align*}
\text{iszero}(0). \\
\text{p}(\text{cons}(X,Y)) \leftarrow \text{iszero}(X). \\
\text{p}(\text{cons}(\text{cons}(X,Y),Z)) \leftarrow \text{p}(\text{cons}(X,Y)).
\end{align*}
\]

with the previous type definitions. Given a finite positive integer \( k \), we obtain in \( k + 1 \) steps the following sequence:

\[
\begin{align*}
S_1 &= \{\text{iszero}(\text{nat})\}; \\
S_2 &= \{\text{iszero}(\text{nat}), \text{p}(\text{list}(\text{nat}))\}; \\
S_3 &= \{\text{iszero}(\text{nat}), \text{p}(\text{list}(\text{nat})), \text{p}(\text{list}(\text{list}(\text{nat}))\}; \\
&\vdots \\
S_{k+1} &= \{\text{iszero}(\text{nat}), \text{p}(\text{list}(\text{nat})), \text{p}(\text{list}(\text{list}(\text{nat}))), \ldots, \text{p}(\text{list}^k(\text{nat}))\}; \\
S_{k+2} &= S_{k+1} (\text{fixpoint}).
\end{align*}
\]

The approximation of the cut at depth \( k \) is introduced because the type of predicate \( p \) is the infinite set \( \{\text{nat}, \text{list}(\text{nat}), \text{list}(\text{list}(\text{nat})), \ldots\} \), which is not computable in a finite number of steps.

**Example 7.** Let us consider a program (studied in [19]), still based on the previous type declarations:

\[
\begin{align*}
\text{map-plus}(\text{nil},X,\text{nil}). \\
\text{map-plus}(\text{cons}(X,Y),Z,\text{cons}(U,W)) &\leftarrow \\
&\text{plus}(X,Z,U),\text{map-plus}(Y,Z,W). \\
\text{plus}(0,X,X). \\
\text{plus}(\text{s}(X),Y,\text{s}(Z)) &\leftarrow \text{plus}(X,Y,Z).
\end{align*}
\]

The abstract success set for the program is computed as follows:

\[
\begin{align*}
\mathcal{T}_p^1 &= \{\text{map-plus}(\text{nil},\text{nil})\}; \\
\mathcal{T}_p^2 &= \left\{\begin{array}{l}
\text{map-plus}(\text{list}(\perp),\beta,\text{list}(\perp)) \\
\text{map-plus}(\text{list}(\text{nat}),\xi,\text{list}(\xi)) \\
\text{p}(\text{nat},\delta,\delta) \\
\text{p}(\text{nat},\text{nat},\text{nat})
\end{array}\right\}; \\
\mathcal{T}_p^3 &= \left\{\begin{array}{l}
\text{map-plus}(\text{list}(\perp),\beta,\text{list}(\perp)) \\
\text{map-plus}(\text{list}(\text{nat}),\xi,\text{list}(\xi)) \\
\text{p}(\text{nat},\text{nat},\text{nat}) \\
\text{p}(\text{nat},\delta,\delta) \\
\text{p}(\text{nat},\text{nat},\text{nat})
\end{array}\right\}; \\
\mathcal{T}_p^4 &= \mathcal{T}_p^3 (\text{fixpoint}).
\end{align*}
\]
The abstract success set shows that in any refutation for predicate `map-plus`, if the second argument has type $\alpha$, the third one has a result of type $\text{list}(\alpha)$ and vice versa, while there is a successful computation with any term as second argument if both the first and the third are $\text{nil}$.

The difference with [19] is that, by using bottom-up abstract interpretation, we obtain goal-independent information: given any goal, the type of the possible type-correct solutions is obtained by abstractly unifying it with atoms in $SS^\rho_P$, as proved in Theorem 4.23. Thus, given the goal $:-\text{map-plus}(X, s(0), Y)$:

\[
\begin{align*}
\alpha\text{-mgu}(\text{map-plus}(X, s(0), Y), \text{map-plus}(\text{list}(\text{nat}), \zeta, \text{list}(\zeta))) &= \{\text{list}(\text{nat})/X, \text{list}(\text{nat})/Y\}, \\
\alpha\text{-mgu}(\text{map-plus}(X, s(0), Y), \text{map-plus}(\text{list}(\text{nat}), \text{nat}, \text{list}(\text{nat}))) &= \{\text{list}(\text{nat})/X, \text{list}(\text{nat})/Y\}, \\
\alpha\text{-mgu}(\text{map-plus}(X, s(0), Y), \text{map-plus}(\text{list}(\zeta), \zeta, \text{list}(\zeta))) &= \{\text{list}(\zeta)/X, \text{list}(\zeta)/Y\},
\end{align*}
\]

we obtain that $\text{list}(\text{nat})$ and $\text{list}(\bot)$ are the type informations associated with the variables $X$ and $Y$, in the success hypothesis.

The following example shows the treatment of type errors in our approach and introduces some problems related with the use of our type system. In the following section we will discuss the possibility, given by the abstract interpretation framework, to extend the type inference system to overcome these problems.

**Example 8.** Consider the following simple program:

\[
p(\text{cons}(a, \text{nil})). \\
p(\text{cons}(0, X)) :- p(X).
\]

with the type declarations

\[
\begin{align*}
\text{type} &\quad \text{nat}: & & & \text{type} &\quad \text{list}(\alpha): \\
&\quad 0\rightarrow \text{nat} & & & \quad \text{nil}\rightarrow \text{list}(\bot) \\
&\quad s(\text{nat})\rightarrow \text{nat} & & & \quad \text{cons}(\alpha, \text{list}(\alpha))\rightarrow \text{list}(\alpha)
\end{align*}
\]

\[
\begin{align*}
\text{type} &\quad \text{char}; & & & \text{type} &\quad \text{list}(\alpha): \\
&\quad a\rightarrow \text{char} & & & \quad \text{nil}\rightarrow \text{list}(\bot) \\
&\quad \ldots & & & \quad \text{cons}(\alpha, \text{list}(\alpha))\rightarrow \text{list}(\alpha)
\end{align*}
\]

The type inference system returns the following abstract success set:

\[
SS^\rho_P = \{p(\text{list}(\text{char})), p(\text{ERR})\}.
\]

The atom $p(\text{ERR})$ comes from the second clause in which the argument of the head should be a list containing both characters and natural numbers. The type of this
list cannot be approximated by $\text{list}(\alpha)$. In our system the type $\text{list}(\alpha)$ represents the class of lists with elements of any type, but homogeneous. Thus $\text{list}(\alpha)$ indicates the lists of natural numbers and the lists of characters but not the lists of naturals and characters.

The type inference does not capture the possibility of computing heterogeneous (from a type viewpoint) structures. All the heterogeneous structures are mapped into the $\text{ERR}$ class.

5. Modifying the type inference

In this section we discuss possible changes of the type inference system. Due to the underlying abstract interpretation framework, such changes are achieved by modifying the domain of types. The first extension deals with the problem of decomposing the $\text{ERR}$ class into different, more descriptive, subclasses. The second change provides a description of the whole success set and allows a stronger notion of soundness.

5.1. Decomposing the ERR class

Example 8 of the previous section shows that the result of type inference could be improved: the $\text{ERR}$ class is too comprehensive. In order to have a more precise approximation of the success set, the type inference scheme can be extended to capture the type of heterogeneous data structures. Such a type inference may be defined by allowing type union in the domain of types. The idea is to define a new domain of types in which the most descriptive type between two incomparable types is not $\text{ERR}$ but the type union of them. Thus, in the new domain, types are seen as sets, although, abusing the notation we omit the $\{}$ parentheses for singletons. In Example 8 the most descriptive type between $\text{list}(\text{char})$ and $\text{list}(\text{nat})$ should be $\text{list}(\text{char} \cup \text{nat})$, which describes heterogeneous lists with both characters and natural numbers as elements. It is conceptually easy to extend the Prolog program $P'$ to cope with $\cup$. We omit such a definition here because the program is very long in spite of its conceptual simplicity. Let us denote by $T_\alpha(\text{Var}_r)$ the domain of type terms extended with the $\cup$ (idempotent, commutative, and associative) binary type constructor. According to this view, the new preordering relation on types is:

$$\forall \tau_1, \tau_2 \in T_\alpha(\text{Var}_r), \, \tau_1 \leq_c \tau_2 \iff \text{one of the following conditions hold:}$$

- there exist $\delta^r$ and $\eta^r$ such that $\tau_1 \sim ((\tau_2 \delta^r) \eta^r)$;
- $\tau_1 = f(\tau_{i_1}, \ldots, \tau_{i_n})$, $\tau_2 = f(\tau'_{i_1}, \ldots, \tau'_{i_n})$, and $\tau'_i \leq_c \tau_i$ for all $i \in \{1, \ldots, n\}$;
- $\forall \tau'_i \in \tau_1$, $\exists \tau'_2 \in \tau_2$ such that $\tau'_i \leq_c \tau'_2$.

The previous ordering extends naturally on type patterns. The domain of type patterns ($\bar{\Delta}^n$) is a subset of $T_\alpha(\text{Var}_r)^n / \sim$, isomorphic to $T_2(\text{Var})^n / \sim$, where

$$\forall \tau_1, \tau_2 \in T_\alpha(\text{Var}_r)^n, \, \tau_1 \sim \tau_2 \iff \tau_1 \leq_c \tau_2 \text{ and } \tau_2 \leq_c \tau_1.$$
As an example, in the domain $\mathcal{D}$, the types $\tau_1 = \text{list}(\alpha) \cup \text{list}(\text{nat})$ and $\tau_2 = \text{list}(\alpha)$ are identified because $\tau_1 \sim \tau_2$. The same holds for $\tau_1 = \text{list}(\text{nat})$ and $\tau_2 = \text{list}(\text{nat} \cup \perp_r)$. The depth-$k$ approximation is exactly the same as in the previous case. Note that we have an $ERR$ equivalence class which is smaller than the previous one because we obtain a set of equivalence classes associated with each heterogeneous type structure.

![Diagram](image)

Fig. 3. The modified finite lattice of types: $\mathcal{D}_3$. 
Example 9. Let us consider the type declarations

\[
\text{type} \quad \text{nat} : \quad \text{type} \quad \text{list}(\alpha) :
\]
\[
0 \to \text{nat} \quad \text{nil} \to \text{list}(\bot),
\]
\[
s(\text{nat}) \to \text{nat} \quad cons(\alpha, \text{list}(\beta)) \to \text{list}(\alpha \cup \beta)
\]

\text{end} \quad \text{end}

Given \( k = 3 \), the corresponding abstract domain of types is shown in Fig. 3. By considering this new domain we have that the term \( \text{cons}(0, \text{cons}(\text{cons}(0, \text{nil}), \text{nil})) \) \( \text{nil} \) has type \( \text{list}(\text{nat} \cup \text{list}(\text{nat})) \) while the term \( \text{cons}(0, 0) \) has type \( \text{ERR} \).

Example 10. Consider the following program, where 1 means \( s(0) \), 2 means \( s(s(0)) \) and so on:

\[
\text{plus}(0,X,X).
\]
\[
\text{plus}(s(X),Y,s(Z)) :- \text{plus}(X,Y,Z).
\]
\[
\text{char}\_\text{weight}(a,1).
\]
\[
\text{char}\_\text{weight}(b,2).
\]
\[
\ldots
\]
\[
\text{char}\_\text{weight}(z,26).
\]
\[
\text{word}\_\text{weight}(\text{nil},0).
\]
\[
\text{word}\_\text{weight}(\text{cons}(X,Y),W) :- \text{char}\_\text{weight}(X,S),
\]
\[
\text{word}\_\text{weight}(Y,V),
\]
\[
\text{plus}(S,V,W).
\]
\[
\text{map}\_\text{weight}(\text{nil},\text{nil}).
\]
\[
\text{map}\_\text{weight}(\text{cons}(X,Z),\text{cons}(X,\text{cons}(Y,W))) :-
\]
\[
\text{word}\_\text{weight}(X,Y),
\]
\[
\text{map}\_\text{weight}(Z,W).
\]

with the type declarations

\[
\text{type} \quad \text{nat} : \quad \text{type} \quad \text{list}(\alpha) :
\]
\[
0 \to \text{nat} \quad \text{nil} \to \text{list}(\bot),
\]
\[
s(\text{nat}) \to \text{nat} \quad cons(\alpha, \text{list}(\beta)) \to \text{list}(\alpha \cup \beta)
\]

\text{end} \quad \text{end}

\text{type} \quad \text{char} :
\[
a \to \text{char}
\]
\[
\ldots
\]
\[
z \to \text{char}
\]

\text{end}

The predicate \( \text{map}\_\text{weight} \) is intended to build (given a list of words) a new list in which every word is followed by its weight (the sum of the weights of its characters).
Using the new domain of types the approximated success set for the program is:

\[ SS^r_p = \begin{cases} 
  \text{plus(nat, } \delta, \delta) \\
  \text{plus(nat, nat, nat)} \\
  \text{char_weight(char, nat)} \\
  \text{word_weight(list(⊥_r), nat)} \\
  \text{word_weight(list(char)), nat)} \\
  \text{map_weight(list(list(⊥_r)), list(⊥_r) \cup nat)} \\
  \text{map_weight(list(list(char)), list(char) \cup nat))} 
\end{cases} \]

From the type inference we can deduce that the solutions of goals for the predicate `map_weight` have the following three possible forms:

1. if the first argument is a `nil` value, then the second is a `nil` value; or
2. if the first argument is a list of `nil` values, then the second argument results in a list in which the elements are either `nil` values or natural numbers; or
3. if the first argument is a list of words (lists of characters), then the second argument results in a list of words and natural numbers.

It is important to remark that with the domain of types for homogeneous data-structures, the type of `map_weight` would be

\[ \begin{cases} 
  \text{map_weight(list(⊥_r), list(⊥_r))} \\
  \text{map_weight(list(list(⊥_r)), ERR)} \\
  \text{map_weight(list(list(char)), ERR)} 
\end{cases} \]

5.2. Capturing the whole success set

The approximated description of the success set provided by the type inference for heterogeneous data-structures does not capture the possibility of `type-wrong` solutions. Consider the program composed by the only clause `p(cons(0, Y))` in the world of homogeneous lists. Its approximated success set is \( SS^r_p = \{ p(list(nat)) \} \). Consider now the goal \(-p(cons(X, cons(a, Z)))\) which is not abstractly unifiable with \( p(list(nat)) \), but, despite of this, it is refutable. Its refutation leads to the answer substitution \( \{0/X\} \) which, applied to the goal, makes it not well-typed. The origin of this behaviour is the consideration that a type name is more descriptive than a variable. In fact, the reduction of \( p(cons(0, X)) \) leads to \( p(cons(nat, list(\alpha))) \) which is reduced to \( p(list(nat)) \). This reduction is based on the idea of homogeneity of lists: if one element is a natural number, the others must have the same type. The reduction is also based on the idea that, in the term \( cons(0, X) \), \( X \) must be a list. To obtain an approximation of the whole success set (thus providing a stronger notion of correctness) we have to change completely the point of view. We have to consider descriptivity as comprehensivity. Following this view, lists are heterogeneous and a variable represents terms of any type, and not only the ones leading to type-correct solutions. For the previous one-clause program,
Polymorphic type inference in logic programming

$p(\text{cons}(0, X))$ can be reduced to $p(\text{cons}(\text{nat}, \alpha))$; and, because $\alpha$ can be any type, $SS_p^p$ is now \{\begin{align*} p(\text{list}(\alpha)), & \quad p(\text{list}(\text{nat})), \quad p(\text{ERR}) \end{align*}\}. This means that the solutions for $p$ are heterogeneous lists, list of naturals, or type-wrong solutions. However, this $SS_p^p$ does not give the idea that type-wrong solutions are obtained only by starting with type-wrong goals. Although this type inference captures the whole success set, we still consider that the abstract scheme we presented in Section 4 is more useful to understand and to reason about programs. Goals not abstractly unifiable with atoms in the abstract success have no solutions or only type-wrong solutions. In a sense, these goals “do not correspond to the philosophy of the program”.

These two informally described extensions give the idea of the flexibility of the approach. Due to the bottom-up abstract interpretation scheme, modifications of the type inference can be achieved by modifying the abstract domain.

6. Concluding remarks

In this paper we have presented a polymorphic type inference scheme for logic programs. It can handle both parametric polymorphism and type union, which denote essential aspects in the logic programming style. The approach is based on a bottom-up abstract interpretation technique which is able to collect information on the whole success set of the program independently from the possible goals submitted to the refutation process. Moreover, by allowing non-ground terms into the semantics, we obtain an approach to the polymorphism with a clear declarative foundation. We have considered a depth-$k$ abstraction technique [32] to get finite domains. Different techniques can be used for this purpose, like the widening/narrowing-based approach in [9] to ensure the termination of the abstract interpretation on the infinite lattice of (all) types. By considering a different (compositional) underlying semantics [4], our type inference can be adapted to deal with program modules, thus allowing more powerful analysis like compositional type inference of logic programs and inter-modules type checking, as recently proposed in [8].

There are several directions for further research. It is possible to extend our type inference scheme to meta-predicates. New type constructors should be included in the domain of types, like $\text{predn}(\alpha_1, \ldots, \alpha_n)$, denoting the type of $n$-ary predicates. Moreover we should extend the type declarations to predicates too.

Our type inference, together with a suitable type declaration for predicates, may be usefully used in program debugging and in compiler optimizations. In particular, since it returns an abstract type version of the success set of the program, we observe the applicability of the inferred type information to statically handle (at compile-time) AND-parallelism in logic programming as in determining errors between consumer and producer interchange [16]. Another interesting application of the derivation of polymorphic types in HCL is about the interconnection between first-order logic and discrete programming [18]. To limit the potentially exponential
growth in these methods, in fact, we can usefully use the concept of type for predicate
variables and constants. This technique is widely used for curtailing instantiations
of the first-order logic into the propositional one. Thus, it restricts the number of
possible instantiations of predicates, in order to apply quantitative methods (e.g.
\textit{mathematical programming}) instead of the usual inefficient symbolic calculation.

\textbf{Appendix A}

We prove a weak notion of soundness for the abstract interpretation (Theorem
4.23) with respect to the following \( T_p \)-based semantics capturing "well-typed"
derivations.

**Definition A.1 \((T_p^\text{wt})\ operator).** Let us consider \( I \in \mathcal{P}(B_p) \):

\[
T_p^{\text{wt}}(I) = \begin{cases} 
C : A := B_1, \ldots, B_n \in P, \quad \langle B'_1, \ldots, B'_n \rangle \prec_c I \\
(B_1, \ldots, B_n) \varnothing = B'_1, \ldots, B'_n \\
(\sigma \in \text{dom}(\sigma) \subseteq \text{vars}(A) - \text{vars}(B_1, \ldots, B_n)) \\
A \text{ well-typed} \Rightarrow A\sigma \text{ well-typed}
\end{cases}
\]

**Proposition A.2.** \( T_p^{\text{wt}} \) is continuous on \( (\mathcal{P}(B_p), \subseteq) \).

**Proof.** The proof is standard [12]. \( \square \)

The fixpoint of \( T_p^{\text{wt}} \) is the set of atoms (ground or non-ground) with successful
well-typed derivations. The following theorem specifies the soundness of \( T_p^{\text{wt}} \)-based
semantics with respect to the semantics of well-typed derivations.

**Theorem A.3.** Let \( P \) be a logic program. If \( p(\overline{f}) \xrightarrow{\sigma}^*\emptyset \), then \( p(\overline{f}) \varnothing \in T_p^{\text{wt}} \uparrow \omega \).

**Proof.** By induction on the length \( n \) of the derivation \( (\rightarrow^*) \):

\textit{Base case.} If \( p(\overline{f}) \xrightarrow{0} \emptyset \), then \( p(\overline{f}) \varnothing \) is a well-typed instance of a unit clause in
\( P \). Thus \( p(\overline{f}) \varnothing \in T_p^{\text{wt}} \uparrow \omega \);

\textit{Inductive case.} If \( p(\overline{f}) \xrightarrow{\sigma^*}^{n+1} \emptyset \), then there exists a clause \( C : p(\overline{f}) := B_1, \ldots, B_n \in P \)
such that

\[
p(\overline{f}) \xrightarrow{\sigma^*}^{n+1} B_1, \ldots, B_n \xrightarrow{\sigma^*}^{n} \emptyset
\]

where \( p(i) \varnothing = p(\overline{f}) \varnothing \) and \( \varnothing = \varnothing' \varnothing'' \). For each \( i = 1, \ldots, n : B_i \varnothing \varnothing'' \xrightarrow{\sigma^*}^{n} \emptyset \) and \( n_i \leq n \)
(see [12]). By the inductive hypothesis, \( (B_1, \ldots, B_n) \varnothing \in T_p^{\text{wt}} \uparrow \omega \) and is well-typed.

By the definition of \( T_p^{\text{wt}} \), we have \( p(\overline{f}) \varnothing \in T_p^{\text{wt}} \uparrow \omega \). \( \square \)

The proof of the Theorem 4.23 is based on the following lemma:
Lemma A.4. \( \forall n \in \omega, \forall p(\bar{t}) \in T_p^{\omega} \uparrow n, \exists p(\bar{\tau}) \in T_p^{\omega} \uparrow n \) such that \( \alpha\text{-mgu}(p(\bar{t}), p(\bar{\tau})) \neq \text{Fail} \).

Proof. By induction on \( n \):

Base case. Straightforward.

Inductive case. Let \( p(\bar{t}) \sigma \in T_p^{\omega} \uparrow (n + 1) = T_p^{\omega}(T_p^{\omega} \uparrow n) \). By definition:

\[
C : p(\bar{t}) :- B_1, \ldots, B_n \in P, \quad \left( B_1', \ldots, B_n' \right) \ll C T_p^{\omega} \uparrow n,
\]

\[
(B_1, \ldots, B_n) \sigma = B_1', \ldots, B_n', \quad (B_1', \ldots, B_n') \sigma \text{ is well-typed},
\]

\[
dom(\sigma) \subseteq \text{vars}(\bar{t}) - \text{vars}(B_1, \ldots, B_n),
\]

\[
p(\bar{t}) \text{ well-typed} \Rightarrow p(\bar{t}) \sigma \text{ well-typed}.
\]

By the inductive hypothesis \( B_1', \ldots, B_n' \in T_p^{\omega} \uparrow n \) and

\[
\alpha\text{-mgu}((B_1', \ldots, B_n'), (B_1', \ldots, B_n')) \neq \text{Fail}.
\]

Then \( \alpha\text{-mgu}((B_1, \ldots, B_n) \sigma, (B_1', \ldots, B_n')) \neq \text{Fail} \). Since \( (B_1, \ldots, B_n) \sigma \) is well-typed, we have:

\[
\alpha\text{-mgu}((B_1, \ldots, B_n), (B_1', \ldots, B_n')) = \sigma^\tau \land \sigma^\tau \neq \text{Fail}.
\]

Thus \( \alpha\text{-apply}(p(\bar{t}), \sigma^\tau) \in T_p^{\omega} \uparrow (n + 1) \).

We prove that

\[
\alpha\text{-mgu}(p(\bar{t}) \sigma, \alpha\text{-apply}(p(\bar{t}), \sigma^\tau)) \neq \text{Fail}.
\]

This is equivalent to proving that\(^1\) \( \alpha\text{-mgu}(p(\bar{t}) \sigma, \alpha\text{-apply}(p(\bar{t}), \sigma^\tau)) \neq \text{Fail} \). Assume that

\[
\alpha\text{-mgu}(p(\bar{t}) \sigma, \alpha\text{-apply}(p(\bar{t}), \sigma^\tau)) = \text{Fail},
\]

then there exist \( t/x \in \theta(\bar{t}) \) and \( \tau/x \in \theta^\tau \) such that \( x \in \text{vars}(B_1, \ldots, B_n) \) and

\[
:- \text{Type}(t, \tau) \approx \tilde{\rho} \cdot \text{Fail}.
\]

Thus \( \alpha\text{-mgu}((B_1, \ldots, B_n) \sigma, (B_1', \ldots, B_n')) \) must fail, leading to a contradiction. \( \square \)

Theorem 4.23 (Soundness). Let \( P \) be a logic program and let \( G \) be a goal: \( :- B_1, \ldots, B_n \).
If \( G \xrightarrow{\rho} \Diamond \), then there exists \( (B_1', \ldots, B_n') \ll G SS^\tau_p \) such that \( \alpha\text{-mgu}((B_1, \ldots, B_n) \sigma, (B_1', \ldots, B_n')) \neq \text{Fail} \).

Proof. It follows by Theorem A.3, \( T_p^{\omega} \)-continuity and Lemma A.4. \( \square \)

\(^1\) Notice that if a well-typed term \( t \) has type \( \tau \), any well-typed instance \( t_\mu \) has type \( \tau' \) and there exists a type substitution \( \mu^\tau \) such that \( \tau^\mu = \tau' \).
References


Polymorphic type inference in logic programming


