Type Dependencies for Logic Programs Using ACI-Unification*

Michael Codish Vitaly Lagoon
Department of Mathematics and Computer Science
Ben-Gurion University of the Negev
PoB 653 Beer-Sheva 84105, Israel.
{codish,lagoon}@cs.bgu.ac.il

February 4, 1998

Abstract

This paper presents a new notion of typing for logic programs which generalizes the notion of directional types. The generation of *type dependencies* for a logic program is fully automatic with respect to a given domain of types. The analysis method is based on a novel combination of program abstraction and ACI-unification which is shown to be correct and optimal. Type dependencies are obtained by abstracting programs, replacing concrete terms by their types, and evaluating the meaning of the abstract programs using a standard semantics for logic programs enhanced by ACI-unification. This approach is generic and can be used with any standard semantics. The method is both theoretically clean and easy to implement using general purpose tools. The proposed domain of types is *condensing* which means that analyses can be carried out in both top-down or bottom-up frameworks with no loss of precision for goal-independent analyses. The proposed method has been fully implemented within a bottom-up approach and the experimental results are promising.

Keywords: ACI Unification, Type Analysis, Logic Programs

1 Introduction

One of the foremost impediments to providing efficient implementations of logic programming languages is the absence of the notions of types and modes in

*This research was supported by the Israel Science Foundation administered by the Israel Academy of Sciences and Humanities, by the Israel Ministry of Science and Arts and by the EC-Israel Collaboration Activity ISC-IL-90.

these languages. Acknowledging this state of affairs, the literature is rich in contributions with the common aim of providing adequate type and mode information to aid compilers in overcoming this obstacle (see for example [38]). Recently there has been a growing interest in the notion of *directional types* for logic programs which combine types and modes. Directional types, originally considered in [40, 11] and more recently in [1, 3, 37, 10, 9], have proven useful in applications such as proving termination of programs and controlling execution of programs through a delay mechanism. Directional types have a declarative aspect as well as an operational aspect prescribing not only types for the predicates in a program but also the intended ways in which these predicates will be called. Directional types associate *input* and *output* assertions with each predicate of a program. An input assertion describes the calls to a predicate. An output assertion describes the success patterns of a predicate under the assumption that it is called as specified by the corresponding input assertion.

This paper focuses on the specification of data descriptions to capture information about types and about *type dependencies* which characterize the set of possible directed types of a given program and hence generalize the notion of directional types. The key idea is to express the dependencies between the types of a predicates arguments similar to the way groundness dependencies are expressed in the Pos domain (e.g. [15, 36]). Type dependencies are first mentioned in [31] and are reminiscent of the implication types of [39].

As a typical example, consider the following directed type for the well-known *append* relation which specifies that the result of concatenating two lists of elements of type τ is a list of elements of type τ:

$$\texttt{type append(A,B,C):}\ A = list(\tau) \wedge B = list(\tau) \longrightarrow C = list(\tau)$$

A type dependency for the *append* relation will specify the dependencies between the three arguments for any call pattern. For example $A = list(\tau) \wedge B = list(\tau') \longrightarrow C = list(\tau \cup \tau')$ (appending lists of type τ and τ' gives a list of elements with both types) as well as $C = list(\tau) \longrightarrow A = list(\tau) \wedge B = list(\tau)$ (partitioning a list of type τ gives two lists of type τ). At the end of the paper (in Example 15) we come back to this example and illustrate how we propose to express type dependencies of this sort.

The framework of abstract interpretation [20] provides the basis for a semantic approach to dataflow analysis. A program analysis is viewed as a nonstandard, abstract semantics defined over a domain of data descriptions. An abstract semantics is constructed by replacing operations in a suitable concrete semantics with corresponding abstract operations defined on data descriptions. Program analyses are defined by providing finitely computable abstract interpretations which preserve interesting aspects of program behavior. Formal justification of program analyses is reduced to proving conditions on the relation between data and data descriptions and on the elementary operations defined on the data descriptions.

In this paper we present a new domain for the inference of type dependencies. The types in our domain are associative, commutative and idempotent terms (ACI for short). For the analysis of type dependencies we propose an abstraction which replaces concrete program terms by their types. The meaning of programs involving type expressions is formalized by enhancing a standard semantics for logic programs with ACI-unification [4]. As a consequence, type inference is reduced to evaluating the meaning of an abstract program. No additional proof theory, no special inference rules and no special inference procedures are required. The justification of a program analysis follows by formalizing the correspondence between concrete terms and type expressions and showing that the ACI-unification of type expressions mimics correctly the concrete unification of terms. The combination of program abstraction and ACI-enhanced concrete evaluation results in a promising technique which is both theoretically clean and easy to implement using general purpose tools.

The idea of obtaining static analyses by program abstraction is derived from ideas presented in [28, 19, 26, 15, 14] and has been applied in a variety of contexts, e.g. [44, 23, 7, 25, 24]. Unique to our approach is the combination of a simple syntax directed program abstraction and a standard equality theory for type expressions. We pay special attention to the role of variables in abstract programs and in their semantics. It turns out, similar to the case of classic semantics of logic programs, that there is a reason for introducing variables to the underlying abstract Herbrand base: For program analyses this facilitates efficient evaluation and a more concise representation. For type analysis this facilitates the specification of *parametric* types and type dependencies. In this sense we follow the approach described in [7] where the authors propose a clear and intuitive type analysis for logic programs based on parametric types.

The rest of the paper is organized as follows: Section 2 provides some preliminary definitions and clarifies the notation used throughout. Sections 3 and 4 introduce our abstract domain of symbolic types and its abstract operations based on ACI unification. A bottom-up polymorphic analysis based on the abstraction of the s-semantics and several examples of its use are described in Section 5. Section 6 summarizes and provides our conclusion. This paper is an extended version of [17] which was inspired by the earlier work presented in [14].

2 Preliminaries

Syntax: In the following we assume a familiarity with the standard definitions and notation for logic programs as described in [35, 2]. For any set of function symbols Σ and variables $\mathcal{V}$, we let $T(\Sigma, \mathcal{V})$ denote the set of terms constructed using symbols from Σ and variables from $\mathcal{V}$. Substitutions are mappings from $\mathcal{V}$ to $T(\Sigma, \mathcal{V})$ and defined as usual. We assume a first order language with a fixed set of predicate symbols Π, a fixed signature Σ and a countable set of variables $\mathcal{V}$. The set of atoms constructed using predicate symbols from Π and terms from $T(\Sigma, \mathcal{V})$ is denoted by $\mathcal{B}_{\mathcal{V}}$. The elements of $\wp(\mathcal{B}_{\mathcal{V}})$, i.e. sets of atoms are called interpretations. Throughout this paper, syntactic objects (e.g. terms, atoms and interpretations) will be viewed modulo renaming of variables. We assume the standard ordering on terms and other syntactic objects. We let $t_1 \leq t_2$ denote that t_1 is less general than t_2. The most general unifier of t_1 and t_2 is denoted $mgu(t_1, t_2)$ if it exists.

For a syntactic object s and a set of equivalence classes of objects I, we denote by $\langle c_1, \ldots, c_n \rangle \ll_s I$ that $c_1, \ldots, c_n$ are representatives of elements of I renamed apart from s and from each other.

Semantics: The (semantic based) type analysis described in this paper is generic and independent of any particular (e.g. top-down or bottom-up) concrete semantic definition. Examples will be given within a bottom-up approach based on abstractions of the s-semantics described in [21, 22, 8] which is a non-ground version of the standard minimal model semantics for logic programs. The s-semantics of a program P is a set of possibly non-ground atoms which characterizes both declarative as well as operational properties of a program. In particular: (a) the ground instances of the s-semantics gives precisely the minimal model of P; and (b) the computed answer substitutions for any initial goal G with P are determined by solving G using the (potentially infinite) set of atoms in the s-semantics of P. The s-semantics of P is evaluated as the least fixed point of an immediate consequence operator $T_P : \wp(\mathcal{B}_{\mathcal{V}}) \rightarrow \wp(\mathcal{B}_{\mathcal{V}})$ similar to the standard minimum Herbrand model semantics. However in this case the underlying domain involves possibly non-ground atoms from the set $\mathcal{B}_{\mathcal{V}}$.

$$\mathrm{T_P(I)} = \left\{ \mathrm{h}\theta \;\middle|\; \begin{array}{l} C \equiv h \leftarrow b_1, \ldots, b_n \in P, \\ \langle a_1, \ldots, a_n \rangle \ll_C I, \\ \theta = mgu(\langle b_1, \ldots, b_n \rangle, \langle a_1, \ldots, a_n \rangle) \end{array} \right\} \tag{1}$$

As a basis for designing program analyses, the s-semantics has proven particularly useful (see for example [5] and [13]). This is due to the fact that on the one hand, it captures both declarative as well as operational properties, and on the other hand because of the mathematical simplicity of its definition. An important advantage of using a non-ground semantics is that the meaning of a program fragment is determined by the symbols it contains without considering the entire underlying alphabet.

Example 1 *Consider the "append" relation*

```
append([],Ys,Ys).
append([X|Xs],Ys,[X|Zs]) :- append(Xs,Ys,Zs).
```

The standard meaning of this program depends on the underlying alphabet (often assumed to be determined by the symbols occuring in the program and hence on the context). In the context of a program involving digits, the minimal model will contain *facts of the form* $append([1,2],[3,4,5],[1,2,3,4,5])$ *while in the context of characters it will contain elements of the form* $append([a,b],[c,d],[a,b,c,d])$. *Consequently the meaning of a program module must be reconsidered with each change in the underlying alphabet of discourse. In contrast to the minimal model semantics, a typical element in the s-semantics of the append program is a universally quantified atom of the form* $append([X_1, X_2], Y, [X_1, X_2 \mid Y])$.

The advantages of using a non-ground semantics become more tangible in cases where we wish to actually *compute* program meanings — as in the case of semantic based program analysis.

E-unification: Given a pair of terms t_1 and t_2 (or other syntactic objects), the associated unification problem is to decide if there exists a substitution θ which unifies t_1 and t_2. In this paper we consider the E-unification problem. Namely, the problem of deciding if two terms are unifiable in the presence of an equality theory $=_E$. For example if the function symbol $f/2$ is commutative, then the terms $f(a,X)$ and $f(b,Y)$ are unifiable by $\{X \mapsto b,\ Y \mapsto a\}$. Of course we are also concerned with unification algorithms which compute such unifiers if they exist. The standard unification used in logic programs is syntactic and assumes the *empty* equality theory. In this paper we are concerned with ACI-unification, which is an instance of E-unification in the presence of an equality theory which is *associative*, *commutative* and *idempotent.*

Formally, for a fixed signature Ω, countable set of variables $\mathcal{V}$ and equality theory $E \subseteq T(\Omega,\mathcal{V}) \times T(\Omega,\mathcal{V})$, E-unification consists in solving equations over the E-free algebra $[T(\Omega,\mathcal{V})]_{=_E}$. For this purpose we consider substitutions with the instantiation ordering defined as usual:

$\mu_1 \leq_E \mu_2$ iff there exists a substitution ψ such that $\mu_1 =_E \mu_2 \circ \psi$.

If E is the empty equality theory, we write $\mu_1 \leq \mu_2$. An E-unifier of the terms τ_1, τ_2 is a substitution μ such that $\tau_1\mu =_E \tau_2\mu$. The set of all E-unifiers of τ_1 and τ_2 is denoted by $U_E(\tau_1,\tau_2)$. A *complete set* $cU_E(\tau_1,\tau_2)$ of E-unifiers of τ_1 and τ_2 satisfies the conditions:

- $cU_E(\tau_1,\tau_2) \subseteq U_E(\tau_1,\tau_2)$, and
- for all $\mu \in U_E(\tau_1,\tau_2)$ there exists $\psi \in cU_E(\tau_1,\tau_2)$ such that $\mu \leq_E \psi$.

A *minimal complete set* $\mu U_E(\tau_1,\tau_2)$ of unifiers of τ_1 and τ_2, is a complete set satisfying the additional condition:

- for all $\mu_1, \mu_2 \in \mu U_E(\tau_1,\tau_2)$, $\mu_1 \leq_E \mu_2$ implies $\mu_1 = \mu_2$.

A minimal complete set of E-unifiers may not always exist and even if it does may not be finite. This depends on the equality theory E. For example, the syntactic unification problem (empty theory) is *unitary* — a minimal complete set always exists and has cardinality ≤ 1. For ACI-unification, a minimal complete set always exists and is finitary [4]. The underlying decision problem of ACI-unification is in general, NP-complete [30]. The algorithms for ACI-unification which are described in the literature compute a *complete* set of ACI-unifiers which is in general not minimal. For efficiency in applications which use these unifiers, such a set should be close to minimal in practice.

Abstract Interpretation: For semantic based program analysis we assume the standard framework of abstract interpretation [20] as defined in terms of *Galois insertions*. A Galois insertion is a quadruple $((A, \sqsubseteq_A), \alpha, (B, \sqsubseteq_B), \gamma)$ where:

1. $(A, \sqsubseteq_A)$ and $(B, \sqsubseteq_B)$ are compete lattices of *concrete* and *abstract domains* respectively;
2. $\alpha : A \rightarrow B$ and $\gamma : B \rightarrow A$ are monotonic functions called *abstraction* and *concretization functions respectively*; and
3. $a \sqsubseteq_A \gamma(\alpha(a))$ and $\alpha(\gamma(b)) = b$ for every $a \in A$ and $b \in B$.

3 A Domain of Symbolic Types

This section presents a step-by-step construction of the abstract domain of symbolic types which is very similar to the construction of the concrete domain. The syntactic objects are abstract terms, abstract atoms, abstract substitutions and abstract interpretations. We formalize the relation between abstract and concrete interpretations in terms of a Galois insertion provided by the definition of suitable functions of abstraction and concretization.

Syntax: We define a first order language for abstract terms using the vocabulary Π, Σ^α and $\mathcal{V}$ which shares the same predicate symbols and variables with the concrete domain. The set Σ^α includes: a binary set constructor $\oplus$ and a collection of *monomorphic* and *polymorphic* description symbols. The monomorphic symbols are constants (e.g. $nat/0$, $char/0$, $atom/0$) and the polymorphic symbols are unary (e.g. $list/1$, $tree/1$). Intuitively, the description symbols represent sets of function symbols in the corresponding concrete alphabet Σ. For example, the description symbol *list* might be defined to represent the $cons/2$ symbol in the concrete alphabet and the description constant *nat* might represent the symbols $s/1$ and $0/0$.

The elements of the term algebra $T(\Sigma^\alpha, \mathcal{V})$ are associated with an equality theory $=_{ACI}$ induced by the following axioms:

$$\begin{array}{ll} (x \oplus y) \oplus z = x \oplus (y \oplus z) & \text{(associativity)} \\ x \oplus y = y \oplus x & \text{(commutativity)} \\ x \oplus x = x & \text{(idempotence)} \end{array} \tag{2}$$

giving $\oplus$ the flavor of a set constructor.

Example 2 *The three expressions $a \oplus X \oplus b \oplus a$, $b \oplus X \oplus a \oplus X$ and $a \oplus b \oplus X$ are equivalent with respect to the theory of Equation (2).*

We say that an abstract term is *polymorphic* if it involves unary function symbols and is *monomorphic* otherwise.

Type abstraction: The relation between concrete and abstract terms is specified by an abstraction function $\alpha_\Sigma : \Sigma \rightarrow \Sigma^\alpha$ which associates each concrete function symbol with a corresponding description symbol from Σ^α partitioning the set of concrete function symbols. In addition, a function $\pi : \Sigma \rightarrow \wp(N)$ associates each function symbol $f/n \in \Sigma$ with a set of designated argument positions which is a subset of $\{1, \ldots, n\}$. The type of a term $f(t_1, \ldots, t_n)$ is determined by $\alpha_\Sigma(f)$ (the description of f), and by the types of its subterms. Intuitively, $\alpha_\Sigma(f)$ is a description of f and $\pi(f)$ specifies a designated set of "recursive" argument positions of f. The type of a term is determined by traversing its recursive paths (as specified by π) and collecting descriptions of the functions symbols encountered (as specified by α_Σ). When a function symbol associated with a monomorphic description symbol is encountered, then its non-recursive arguments are ignored. When a function symbol associated with a polymorphic description symbol is encountered, then the types of its non-recursive arguments are introduced as the argument to the corresponding unary description symbol. Using α_Σ and π we define an abstraction function *type* relating concrete to abstract terms.

$$type : T(\Sigma, \mathcal{V}) \rightarrow T(\Sigma^\alpha, \mathcal{V})$$

$$type(t) = \begin{cases} t & \text{if } t \text{ is a variable} \\ \alpha_\Sigma(f) \bigoplus\limits_{i \in \pi(f)} type(t_i) & \textit{if } t = f(t_1, \ldots, t_n) \\ & \text{and } \alpha_\Sigma(f) \text{ is a constant} \\ \bigoplus\limits_{i \notin \pi(f)} \alpha_\Sigma(f)(type(t_i)) \oplus & \text{if } t = f(t_1, \ldots, t_n) \\ \qquad \bigoplus\limits_{i \in \pi(f)} type(t_i) & \text{and } \alpha_\Sigma(f) \text{ is unary} \end{cases} \tag{3}$$

We say that a type definition is monomorphic if it involves only monomorphic description symbols. Otherwise a type definition is polymorphic. Consider a monomorphic type definition as a first example.

Example 3 *Assume definitions for α_Σ and π for which:*

$$\begin{array}{llll} \alpha_\Sigma(cons/2) = list & \pi(cons/2) = \{2\} & \alpha_\Sigma(s/1) = nat & \pi(s/1) = \{1\} \\ \alpha_\Sigma(nil/0) = list & \pi(nil/0) = \emptyset & \alpha_\Sigma(0/0) = nat & \pi(0/0) = \emptyset. \end{array}$$

Substituting these definitions in Equation (3) gives the following equation for term abstraction:[1]

$$\text{type(T)} = \begin{cases} T & \text{if } T \text{ is a variable} \\ nat & \text{if } T = 0 \\ nat \oplus type(X) & \text{if } T = s(X) \\ list & \text{if } T = [\,] \\ list \oplus type(Xs) & \text{if } T = [X|Xs] \\ other & \text{otherwise} \end{cases} \tag{4}$$

Consider the symbolic types of the following terms:

$$\begin{array}{l} type([0,0]) = list \oplus type([0]) = list \oplus list \oplus type([\,]) = list \\ type([X,Y]) = list \oplus type([Y]) = list \oplus list \oplus type([\,]) = list \\ type([0|X]) = list \oplus type(X) = list \oplus X \\ type(X) = X \\ type(s(0)) = nat \oplus type(0) = nat \end{array}$$

Observe that the type of a proper (closed) list is "list"; the type of an open ended list is "list $\oplus$ X"; and a variable V has the type "V". Observe also that the type of a natural number is "nat". Consider also the types of the following, perhaps strangely constructed terms:

$$\begin{array}{l} type([0|s(0)]) = list \oplus type(s(0)) = list \oplus nat \\ type(s([0])) = nat \oplus type([0]) = nat \oplus list \end{array}$$

Our notion of types does not distinguish between these two terms, both of which may be considered erroneous for the given definition. The abstract term nat $\oplus$ list indicates that a type error has occurred. By not distinguishing the types of these kinds of concrete terms, type unification remains more "syntactic" and more efficient. Types of this sort do not actually occur in the analysis of "correct" programs. For readability, we may equate such abstract terms with a special type description "error".

The following example, illustrates type abstraction when polymorphic description symbols are involved. In this case type expressions in the abstraction capture not only *primary types* but also *secondary types* expressing information such as "list of characters", or "list of X".

Example 4 *Consider the following definition of polymorphic type abstraction derived from Equation (3) using suitable definitions of α_Σ and π. In this example the definition provides polymorphic types for lists and binary trees capturing also information about the types of the terms in the non-recursive positions of these data structures. Note that the symbol list is now unary (polymorphic). The definition also includes an abstraction*

[1] Formally α_Σ must induce a partition of Σ. In the examples we adhere to the policy that all function symbols not specially mentioned are mapped to the special descriptive constant *other*.

for numbers (num) instead of the abstraction of natural numbers in successor notation of Equation (4).

$$\text{type(T)} = \begin{cases} T & \text{if } T \text{ is a variable} \\ num & \text{if } T \text{ is a number} \\ nil & \text{if } T = [\,] \\ list(type(X)) \oplus type(Xs) & \text{if } T = [X|Xs] \\ void & \text{if } T = void \\ tree(type(X)) \oplus \\ \quad \oplus type(L) \oplus type(R) & \text{if } T = tree(X, L, R) \\ other & \text{otherwise} \end{cases} \tag{5}$$

Consider the type abstraction of the following terms:

$$\begin{aligned} type([0,0]) &= list(type(0)) \oplus type([0]) = \\ &= list(num) \oplus list(type(0)) \oplus type([\,]) = \\ &= list(num) \oplus list(num) \oplus nil = \\ &= list(num) \oplus nil \end{aligned}$$

$$\begin{aligned} type([X,Y]) &= list(type(X)) \oplus list(type(Y)) \oplus nil = \\ &= list(X) \oplus list(Y) \oplus nil \end{aligned}$$

$$type([X|Y]) = list(X) \oplus Y$$

$$\begin{aligned} type(tree(1, void, void)) &= tree(num) \oplus type(void) \oplus type(void) = \\ &= tree(num) \oplus void \end{aligned}$$

The list $[0,0]$ is abstracted to $list(num) \oplus nil$ which describes any recursive structure constructed using $cons/2$ where the recursion terminates by $[\,]$ and all non-recursive arguments are numbers. Thus, this type expression indeed describes a list of numbers. Similarly to the case of monomorphic type abstraction, more instantiated terms have more instantiated types. For instance $[0,0]$ and $[X,Y]$ have types $list(num) \oplus nil$ and $list(X) \oplus list(Y) \oplus nil$, respectively. Note that applying the substitution $\{X \mapsto num, Y \mapsto num\}$ to $list(X) \oplus list(Y) \oplus nil$ gives the type expression $list(num) \oplus nil$. Note also that the type of an open-ended recursive data structure (e.g. $[X|Y]$) is more general than that of the corresponding closed structure (e.g. $[X,Y]$).

The notation of Equation (5) for the abstraction of the empty list and the empty tree is more cumbersome than that of Equation (4) for monomorphic types. Indeed it is harder to "read" polymorphic types. An alternative is to consider an empty structure (e.g. nil) as $list(\bot)$ where $\bot$ denotes an empty type (see for example [17]). However in this case additional axioms must be introduced to the underlying equality theory, (e.g. $X \oplus \bot = X$ and $f(X \oplus Y) = f(X) \oplus f(Y)$ for every $f/1 \in \Sigma^\alpha$). While types become easier to read, this complicates the unification algorithm and introduces more "semantics" to the abstract terms without providing more information about types.

Symbolic types: It is apparent that not every abstract term represents the "type" of some concrete term. The domain of symbolic types $\Upsilon \subseteq T(\Sigma^\alpha, \mathcal{V})$ is the image of $T(\Sigma, \mathcal{V})$ under the function $type$:

$$\Upsilon = \left\{ \, type(t) \;\middle|\; t \in T(\Sigma, \mathcal{V}) \, \right\}. \tag{6}$$

Example 5 *Assume a definition of the function type which is the identity for concrete constants (e.g. $\alpha_\Sigma(f/0) = f/0$ and $\pi(f/0) = \emptyset$). In this case, the abstract terms $f \oplus g$ and $X \oplus g$ are valid syntactic objects in $T(\Sigma^\alpha, \mathcal{V})$ but there does not exist a concrete term t such that $type(t) = f \oplus g$, or such that $type(t) = X \oplus g$.*

Abstract atoms are entities of the form $p(\tau_1, \ldots, \tau_n)$ where $p/n \in \Pi$ and $\tau_1, \ldots, \tau_n \in \Upsilon$. We denote by $\mathcal{B}^\alpha_\mathcal{V}$ the set of abstract atoms modulo the equivalence provided by Equation (2). In the following, we often abuse notation and write π instead of $[\pi]_{=_{ACI}}$ for an abstract atom π.

Abstract substitutions are substitutions from $\mathcal{V}$ to Υ. Thus, abstract substitution resemble the concrete ones and all of their standard operations are defined as usual. For example, the application of an abstract substitution μ to a symbolic type τ is defined as usual by replacing occurrences of each variable X in τ by the type $\mu(X)$. The composition of abstract substitutions is also defined as usual, by applying the second substitution to the symbolic types in the range of the first. We often refer to abstract substitutions as *ACI-substitutions* because the terms in the range of these substitutions respect $=_{ACI}$ equivalence.

Abstract terms (or atoms) are associated with an ordering much the same as concrete syntactic objects. We say that an abstract term τ_1 is less general than an abstract term τ_2, denoted $\tau_1 \preceq \tau_2$, if there exists a substitution ψ such that $\tau_1 =_{ACI} \tau_2\psi$. In other words, the ordering of abstract terms is given by "$\leq_{ACI}$". Thereby, $(\tau_1 \preceq \tau_2) \wedge (\tau_2 \preceq \tau_1) \Leftrightarrow (\tau_1 =_{ACI} \tau_2)$. This ordering extends as usual for other syntactic objects such as tuples of atoms.

The abstract domain: The abstract domain for type analysis is formalized as a domain of sets of abstract atoms with a suitable notion of ordering and equivalence. We consider the *lower power domain* (or Hoare power domain [27]) with the ordering defined as

$$\mathcal{I}_1 \preceq \mathcal{I}_2 \Leftrightarrow \forall \pi_1 \in \mathcal{I}_1 \; \exists \pi_2 \in \mathcal{I}_2 : \; \pi_1 \preceq \pi_2. \tag{7}$$

The corresponding equivalence relation:

$$\mathcal{I}_1 \approx \mathcal{I}_2 \;\Leftrightarrow\; (\mathcal{I}_1 \preceq \mathcal{I}_2) \wedge (\mathcal{I}_2 \preceq \mathcal{I}_1) \tag{8}$$

induces a partial order on $[\wp(\mathcal{B}^\alpha_\mathcal{V}\;)]_\approx$. The elements of this domain, i.e. subsets of $\mathcal{B}^\alpha_\mathcal{V}$ modulo the equivalence of Equation (8), are called *abstract interpretations*. In the following by abuse of notation we denote $[\wp(\mathcal{B}^\alpha_\mathcal{V}\;)]_\approx$ by $\wp(\mathcal{B}^\alpha_\mathcal{V})$.

The above partial order provides a basis for relating the abstract interpretations to *downward-closed sets* also called *c-interpretations* [21]. A set $\mathcal{I}$ of abstract atoms is downward-closed if $\pi \in \mathcal{I}$ and $\pi' \preceq \pi$ implies $\pi' \in \mathcal{I}$. Let $\mathcal{I}{\downarrow}$ denote a minimal downward-closed set containing $\mathcal{I}$. Equation (8) provides $\mathcal{I} \approx \mathcal{I}{\downarrow}$ for any $\mathcal{I} \in \wp(\mathcal{B}^\alpha_\mathcal{V}\;)$, i.e. any member of $\wp(\mathcal{B}^\alpha_\mathcal{V})$ is equivalent to some downward-closed set. Thus, the domain is, in fact, a lower power domain, partially ordered by set inclusion. This observation trivially implies the following lemma:

Lemma 3.1 $(\wp(\mathcal{B}^{\alpha}_{\mathcal{V}}), \preceq)$ *is a complete lattice.*

PROOF. *If L is a set of downward-closed sets, i.e. elements of $\wp(\mathcal{B}^{\alpha}_{\mathcal{V}})$, then $\cap L$ and $\cup L$ are downward-closed, therefore $lub(L) \approx \cup L$ and $glb(L) \approx \cap L$.* □

The following theorem establishes that in the case of monomorphic type abstraction the abstract domain is finite. This result is used below to establish the termination of monomorphic type analyses which can be viewed as chains in the abstract domain.

Theorem 3.2 *If Σ^{α} contains no polymorphic description symbols then $\wp(\mathcal{B}^{\alpha}_{\mathcal{V}})$ is finite.*

PROOF. *The elements of $\wp(\mathcal{B}^{\alpha}_{\mathcal{V}})$ are sets of equivalence classes of abstract atoms $[\pi]_{=_{ACI}}$. It suffices to prove that for any predicate symbol p/n the number of associated equivalence classes of abstract atoms $[\pi]_{=_{ACI}}$ is finite. We prove that each equivalence class of abstract atoms constructed using p/n has a representative with a number of variables bounded by $2^n - 1$. Assume an atom $\pi = p(\tau_1, \ldots, \tau_n)$ has more than $2^n - 1$ distinct variables. Then there are at least two variables X and Y occuring together in exactly the same arguments of π. Consider the atom $\pi' = \pi \cdot \{X \mapsto Z,\ Y \mapsto Z\}$. By construction we have $\pi' \preceq \pi$ and it is easy to see that $\pi \preceq \pi'$ with $\pi = \pi' \cdot \{Z \mapsto (X \oplus Y)\}$. Thus, π and π' are in the same equivalence class and $|vars(\pi')| = |vars(\pi)| - 1$. So, for any atom constructed using p/n there exists an equivalent atom with all variables occuring in distinct subsets of argument positions, i.e. an atom with at most $2^n - 1$ variables.* □

The relation between the abstract and concrete domains is formalized by lifting the *type* abstraction of Equation (3) to atoms and other syntactic constructs such as tuples of atoms, clauses, etc. In each case the abstraction, denoted in the following as σ, is obtained by application of *type* separately to each term of the construct. In particular, for atoms:

$$\begin{array}{l} \sigma : \mathcal{B}_{\mathcal{V}} \rightarrow \mathcal{B}^{\alpha}_{\mathcal{V}} \\ \sigma(p(t_1, \ldots, t_n)) = p(type(t_1), \ldots, type(t_n)) \end{array} \tag{9}$$

A Galois insertion is obtained by lifting of σ to introduce an abstraction function α on interpretations and defining a corresponding concretization function in the standard way:

$$\begin{array}{ll} \alpha : \wp(\mathcal{B}_{\mathcal{V}}) \rightarrow \wp(\mathcal{B}^{\alpha}_{\mathcal{V}}) & \gamma : \wp(\mathcal{B}^{\alpha}_{\mathcal{V}}) \rightarrow \wp(\mathcal{B}_{\mathcal{V}}) \\ \alpha(I) = \left\{ \ \sigma(a) \ \middle| \ a \in I \ \right\} & \gamma(\mathcal{I}) = \bigcup \left\{ \ I \ \middle| \ \alpha(I) \preceq \mathcal{I} \ \right\} \end{array} \tag{10}$$

Theorem 3.3 $\langle \wp(\mathcal{B}_{\mathcal{V}}), \alpha, \wp(\mathcal{B}^{\alpha}_{\mathcal{V}}), \gamma \rangle$ *is a Galois insertion.*

PROOF.

- *α and γ are monotonic — immediately by the definitions.*

- $\forall I \in \wp(\mathcal{B}_{\mathcal{V}}) : \gamma(\alpha(I)) = \bigcup \{ \ I' \ \big| \ \alpha(I') \preceq \alpha(I) \ \} \supseteq I$,
  since $I \in \{ \ I' \ \big| \ \alpha(I') \preceq \alpha(I) \ \}$;
- $$\forall \mathcal{I} \in \wp(\mathcal{B}^{\alpha}_{\mathcal{V}}) : \ \alpha(\gamma(\mathcal{I})) \ = \ \alpha \left(\bigcup \{ \ I \ \big| \ \alpha(I) \preceq \mathcal{I} \ \}\right) \\ = \ \bigcup \{ \ \alpha(I) \ \big| \ \alpha(I) \preceq \mathcal{I} \ \} = \mathcal{I},$$
  since $\mathcal{I}$ *is downwards-closed.*

□

We say that an abstract atom π is a type description for a concrete atom a, denoted $\pi \propto a$ if $a \in \gamma(\{\pi\})$, or equivalently, if $\sigma(a) \preceq \pi$. Similarly we say that an abstract interpretation $\mathcal{I}$ is a type description for a concrete interpretation I if $I \subseteq \gamma(\mathcal{I})$, or equivalently, if $\alpha(I) \preceq \mathcal{I}$. The following example illustrates the concept of type description:

Example 6 *Assume the monomorphic type abstraction for lists and natural numbers given in Equation (4). We illustrate the notion of type description providing two groups of abstract and concrete interpretations and the table summarizing the "$\propto$" relation between them:*

$$\mathcal{I}_1 = \{ \ p(list), \ p(X), \ q(nat) \ \} \quad I_1 = \{ \ p([0]), \ q(0) \ \}$$
$$\mathcal{I}_2 = \{ \ p(list \oplus X), \ q(nat) \ \} \quad I_2 = \{ \ p(0), \ q(0) \ \}$$
$$\mathcal{I}_3 = \{ \ p(list), \ q(nat) \ \} \quad I_3 = \{ \ p([X|Y]) \ \}$$

	$\mathcal{I}_1$	$\mathcal{I}_2$	$\mathcal{I}_3$
I_1	$\propto$	$\propto$	$\propto$
I_2	$\propto$	$\not\propto$	$\not\propto$
I_3	$\propto$	$\propto$	$\not\propto$

Example 7 *Assume the monomorphic type abstraction of Equation (4) and consider the "append" relation from Example 1. The abstract interpretation*

$$\mathcal{I} = \left\{ \ append(list, X, X), \ append(list, X, list \oplus X) \ \right\}$$

is a type description of the s-semantics of the "append" program. It is also a type description of the minimal Herbrand model for "append" (which consists of the ground instances of the s-semantics). Observe that both the atoms in $\mathcal{I}$ have $append(list, list, list)$ as an instance and hence describe any atom of the form $append(l_1, l_2, l_3)$ where l_3 is equal to the concatenation of lists l_1 and l_2. Note however that $\mathcal{I}$ describes also those atoms in the minimal model of "append" which are not in the intended semantics, such as $append([\ ], 0, 0)$ and $append([0], 0, [0|0])$. The types of these atoms are $append(list, nat, nat)$ and $append(list, nat, list \oplus nat)$ which are both instances of the abstract atoms in $\mathcal{I}$. Although the presence of such counterintuitive atoms in the minimal model of "append" is often overlooked, a correct approximation of the minimal model should describe them.

The next example characterizes the set of atoms described by an abstract interpretation which will later turn out to be the result of a polymorphic analysis of the append program.

Example 8 *Assume the polymorphic type abstraction of Equation (5). The abstract interpretation*

$$\mathcal{I} = \left\{ \begin{array}{l} append(nil,\ A,\ A), \\ append(list(A) \oplus nil,\ B,\ list(A) \oplus B) \end{array} \right\}$$

describes polymorphic types for the "append" relation. The first abstract atom approximates all the cases of appending to an empty list. In this case the second and the third argument of "append" are the same and so are their types. The expression $list(A) \oplus nil$ *in the second abstract atom denotes any right recursive structure of* $cons/2$ *constructors each of which has an object of type* A *tacked on the left argument position, and all this terminated by a* [ ] *constructor. Let us consider several instances of the second abstract atom:*

1. *If* $B = nil$*, then the corresponding abstract atom*

   $$append(list(A) \oplus nil,\ nil,\ list(A) \oplus nil)$$

   describes the situation when a list of elements of type A *is appended to an empty list. The result is a list of elements of type* A*.*

2. *If* $B = list(C) \oplus nil$*, then the corresponding abstract atom*

   $$append(list(A) \oplus nil,\ list(C) \oplus nil,\ list(A) \oplus list(C) \oplus nil)$$

   describes the situation when a list of elements of type A *is appended to a nil-terminating list of elements of type* C*. The result is a list of elements of types* A *and* C*. Note that* $list(A) \oplus list(C) \oplus nil$ *describes exactly that. The ACI axioms allow the* $cons/2$ *constructors with left type* A *and those with left type* C *to intermingle in any order.*

4 Abstract Operations on Types

The concept of ACI-unification plays a central role in the type analysis presented in this paper. This section describes one of the main contributions of this work demonstrating that ACI-unification provides a correct and optimal unification algorithm for type analysis. As described in Section 2, ACI-unification is an instance of the more general notion of E-unification. The ACI-unification of two terms τ_1 and τ_2 consists in finding a substitution μ satisfying $\tau_1\mu =_{ACI} \tau_2\mu$. The underlying decision problem — deciding whether two terms are ACI-unifiable is NP-complete [30].

It is interesting to note that the restricted ACI-unification problem for monomorphic types is P-time decidable. The algorithm is also simpler — an implementation is illustrated in Appendix A. However, even for monomorphic types, there may be exponentially many most general unifiers for a pair of terms and all of these must be considered for program analysis. Our experimental evaluation (described below) indicates that symbolic type analysis based on ACI-unification is a feasible reality and if correctly designed does not exhibit exponential behaviour.

Correctness of ACI unification: Correctness of ACI-unification as an abstract unification algorithm for symbolic types follows from the monotonicity of type abstraction with respect to the ordering on terms and symbolic types.

Lemma 4.1 *For a concrete term t and a substitution θ:*

$$type(t\theta) = type(t) \cdot \{X \mapsto type(X\theta) \mid X \in dom(\theta)\}.$$

Proof. Let us denote $\mu = \{X \mapsto type(X\theta) \mid X \in dom(\theta)\}$. The proof is by induction on the depth of t: If $depth(t) = 0$ then t is either a constant or a variable. If t is a constant or a variable which is not in the domain of θ then $type(t\theta) = type(t) = type(t)\mu$ and thus the lemma holds. Now assume that $t \in dom(\theta)$. In this case $type(t)\mu = t\mu$ because t is a variable and $t\mu = type(t\theta)$ by definition of μ.

For the induction step assume that the lemma holds for all terms with depth less or equal to some n and consider a term $t = f(t_1, \ldots, t_k)$ with depth $n+1$. We distinguish two cases:

1. If $\alpha_\Sigma(f)$ is a constant then:

$$\begin{aligned} type(t\theta) &= \alpha_\Sigma(f) \bigoplus_{i\in\pi(f)} type(t_i\theta) = \alpha_\Sigma(f) \bigoplus_{i\in\pi(f)} type(t_i)\mu = \\ &= \left(\alpha_\Sigma(f) \bigoplus_{i\in\pi(f)} type(t_i)\right)\mu = type(t)\mu; \quad \text{and} \end{aligned}$$

2. If $\alpha_\Sigma(f)$ is a unary function symbol then:

$$\begin{aligned} type(t\theta) &= \alpha_\Sigma(f)\left(\bigoplus_{i\notin\pi(f)} type(t_i\theta)\right) \oplus \bigoplus_{i\in\pi(f)} type(t_i\theta) = \\ &= \alpha_\Sigma(f)\left(\bigoplus_{i\notin\pi(f)} type(t_i)\mu\right) \oplus \bigoplus_{i\in\pi(f)} type(t_i)\mu = type(t)\mu \end{aligned}$$

□

The result about monotonicity of type abstraction is straightforward from Lemma 4.1.

Theorem 4.2 *For two concrete terms t_1 and t_2:*

$$t_1 \leq t_2 \Rightarrow type(t_1) \preceq type(t_2).$$

Proof. Straightforward by Lemma 4.1 since $t_1 \leq t_2$ implies that there exists a substitution θ such that $t_1 = t_2\theta$. □

Example 9 *Consider the terms $t_1 = [0,0]$, $t_2 = [X,Y]$ and $t_3 = [U|V]$ from Example 3 together with the corresponding notion of types. Observe that $t_1 \leq t_2 \leq t_3$ and that $type(t_1) = list$, $type(t_2) = list$ and $type(t_3) = list \oplus V$. Note that the ordering of the types preserve the ordering of the original terms. Namely, $type(t_1) \preceq type(t_2) \preceq type(t_3)$ with $type(t_3) \cdot \{V \mapsto list\} = type(t_2)$. The type of t_3 reflects the fact that the open ended list t_3 is strictly more general than the closed lists t_1 and t_2.*

Corollary 4.3 *For any pair of atoms (or tuples of atoms) a_1 and a_2:*

$$a_1 \leq a_2 \Rightarrow \sigma(a_1) \preceq \sigma(a_2).$$

The following result establishes the correctness of ACI-unification with respect to our type domain. The theorem also proves correctness of the abstract application of an ACI-substitution to a symbolic type, which mimics the application of a concrete substitution to a concrete term.

Theorem 4.4 (ACI-unification is a correct abstract unification)
Let a_1 and a_2 be concrete atoms and π_1 and π_2 be abstract atoms such that $\pi_1 \propto a_1$ and $\pi_2 \propto a_2$ and let $\theta = mgu(a_1, a_2)$. Then there exists a unifier $\mu \in cU_{ACI}(\pi_1, \pi_2)$ s.t.

1. $\pi_1\mu \propto a_1\theta$,
2. $\forall h \in \mathcal{B}_\mathcal{V} : \ \sigma(h)\mu \propto h\theta$.

Proof. *Consider the tuples $\langle h, a_1\rangle$ and $\langle h, a_2\rangle$, where h is an arbitrary concrete atom. Observe that $\langle h, a_1\rangle\theta \leq \langle h, a_1\rangle$, $\langle h, a_2\rangle\theta \leq \langle h, a_2\rangle$ and hence by Corollary 4.3:*

$$\begin{array}{lllll}\sigma(\langle h, a_1\rangle\theta) & \preceq & \sigma(\langle h, a_1\rangle) & \preceq & \langle\sigma(h), \pi_1\rangle \\ \sigma(\langle h, a_2\rangle\theta) & \preceq & \sigma(\langle h, a_2\rangle) & \preceq & \langle\sigma(h), \pi_2\rangle\end{array}$$

Since θ is a unifier of a_1 and a_2 we have $\sigma(\langle h, a_1\rangle\theta) = \sigma(\langle h, a_2\rangle\theta)$ and thus, $\sigma(\langle h, a_1\rangle\theta)$ is a common ACI-instance of $\langle\sigma(h), \pi_1\rangle$ and $\langle\sigma(h), \pi_2\rangle$. Hence, the complete set $cU_{ACI}(\langle\sigma(h), \pi_1\rangle, \langle\sigma(h), \pi_2\rangle)$ contains a unifier μ s.t. $\sigma(\langle h, a_1\rangle\theta) \preceq \langle\sigma(h), \pi_1\rangle\mu$ or, separating the tuples, $\sigma(h\theta) \preceq \sigma(h)\mu$ and $\sigma(a_1\theta) \preceq \pi_1\mu$. Now the claim of the theorem follows observing that $cU_{ACI}(\langle\sigma(h), \pi_1\rangle, \langle\sigma(h), \pi_2\rangle)$ corresponds to some complete set of ACI-unifiers of π_1 and π_2. Namely, $\mu \in cU_{ACI}(\pi_1, \pi_2)$. □

Corollary 4.5 (Correctness of abstract unification for tuples)
Theorem 4.4 holds if a_1 and a_2, are tuples of concrete atoms and π_1 and π_2 are tuples of abstract atoms such that $\pi_1 \propto a_1$ and $\pi_2 \propto a_2$.
Proof. *Straightforward.* □

<u>**Optimality of ACI unification:**</u> Consider the abstract atoms π_1 and π_2 and the sets of concrete atoms described by each one of them: $I_1 = \{\ a \mid \ \pi_1 \propto a \ \}$ and $I_2 = \{\ a \mid \ \pi_2 \propto a \ \}$, or equivalently $I_1 = \gamma(\{\pi_1\})$ and $I_2 = \gamma(\{\pi_2\})$. Note that for any $a_1 \in I_1$ and $a_2 \in I_2$ all common instances of a_1 and a_2 are in $I_1 \cap I_2$ because I_1 and I_2 are downward-closed.

Let $cI_{ACI}(\pi_1, \pi_2)$ be a "complete set" of common ACI-instances of two abstract atoms π_1 and π_2. Namely, the set obtained by application of the members of $cU_{ACI}(\pi_1, \pi_2)$ to π_1, or equivalently, to π_2. This set is complete in the sense that any common instance of π_1 and π_2 is also an instance of some element from $cI_{ACI}(\pi_1, \pi_2)$.

Theorem 4.4 implies that $cI_{ACI}(\pi_1, \pi_2)$ approximates the set $I_1 \cap I_2$. Let us demonstrate that this approximation is precise, i.e. the set $cI_{ACI}(\pi_1, \pi_2)$ approximates *only* the atoms of $I_1 \cap I_2$.

Theorem 4.6 (ACI-unification is an optimal abstract unification) *Let π_1 and π_2 be two abstract atoms and I_1 and I_2 be the sets of concrete atoms approximated by π_1 and π_2 respectively. Then the approximation of $I_1 \cap I_2$ by $cI_{ACI}(\pi_1, \pi_2)$ is precise.*

PROOF. *We have to demonstrate that only the atoms of $I_1 \cap I_2$ are approximated by $cI_{ACI}(\pi_1, \pi_2)$. Assume by contradiction that there exists an atom b such that for some $\pi \in cI_{ACI}(\pi_1, \pi_2)$, $\pi \propto b$ and $b \notin (I_1 \cap I_2)$. This is possible only if $\pi_1 \not\propto b$ or $\pi_2 \not\propto b$. However, π is a common instance of π_1 and π_2, implying $\pi_1 \propto b$ and $\pi_2 \propto b$. From this contradiction we conclude that $cI_{ACI}(\pi_1, \pi_2)$ does not approximate any atom not in $I_1 \cap I_2$.* □

Thus, the ACI-unification of π_1 and π_2 provides an optimal abstract unification algorithm for type analysis since $cI_{ACI}(\pi_1, \pi_2)$ approximates only the unifiers of the pairs of concrete atoms described respectively by π_1 and π_2.

Theorems 4.4 and 4.6 provide us with the following result generalizing correctness and optimality of ACI-unification for types:

Theorem 4.7 *Let π_1 and π_2 be two abstract atoms. Then*

$$\gamma(cI_{ACI}(\pi_1, \pi_2)) = \gamma(\{\pi_1\}) \cap \gamma(\{\pi_2\}).$$

PROOF. *By Theorem 4.4 the set $\gamma(cI_{ACI}(\pi_1, \pi_2))$ contains all the elements of $\gamma(\{\pi_1\}) \cap \gamma(\{\pi_2\})$ and by Theorem 4.6 contains no other elements.* □

ACI-unification algorithm: The algorithm of ACI-unification used in our implementation is a simplification of the more general algorithm described in [34]. The advantage of this algorithm is that it allows lazy generation of solutions and is easily implemented in declarative languages such as Prolog taking advantage of built-in backtracking. The algorithm unifies tuples of ACI-terms, e.g. $\langle \tau_1, \ldots, \tau_n \rangle$ with $\langle \tau'_1, \ldots, \tau'_n \rangle$ thus solving systems of ACI-equations. This algorithm as well as all other published algorithms of general ACI-unification [41] computes a complete but not necessary minimal set of unifiers. Of course a minimal complete set of unifiers can be obtained in a second phase by choosing the most general unifiers from the computed set. As far as we know the direct, one phase and efficient computation of a minimal complete set of ACI unifiers is still considered an open problem.

Example 10 *The complete set of ACI-unifiers computed in our implementation for the unification of $A \oplus B$ with $C \oplus d$ consist of the following seven unifiers.*

$$cU_{ACI}(A \oplus B, C \oplus d) =$$

$$= \left\{ \begin{array}{l} \{A \mapsto (X \oplus d),\ B \mapsto (Y \oplus d),\ C \mapsto (X \oplus Y)\}, \\ \{A \mapsto (X \oplus d),\ B \mapsto Y,\ C \mapsto (X \oplus Y)\}, \\ \{A \mapsto (X \oplus d),\ B \mapsto d,\ C \mapsto X\}, \\ \{A \mapsto X,\ B \mapsto (Y \oplus d),\ C \mapsto (X \oplus Y)\}, \\ \{A \mapsto X,\ B \mapsto d,\ C \mapsto X\}, \\ \{A \mapsto d,\ B \mapsto (X \oplus d),\ C \mapsto X\}, \\ \{A \mapsto d,\ B \mapsto X,\ C \mapsto X\} \end{array} \right\}$$

This complete set of unifiers is also minimal. However, in the general case the set of unifiers computed by the unification algorithm we use may contain redundant unifiers.

5 A Case Study: Abstracting the s-Semantics

This section illustrates the construction of a bottom-up type analyzer based on the abstraction of the s-semantics[2] defined in Equation (1) and using the abstract operations based on ACI-unification. We demonstrate that bottom-up type analysis provides a precise description of the (non-ground) minimal model of a given program and that this information can be used to approximate the answers for arbitrary initial goals. It is straightforward to obtain also information about call patterns using Magic Sets [6] as described for example in [14, 15].

Symbolic types are condensing: Jacobs and Langen [29] prove that top-down and bottom-up (as well as goal-dependent and goal-independent) analyses are guaranteed to be equally precise when they involve an abstract unification algorithm which is idempotent and commutative and a least upper bound function which is additive. It is interesting to note that our definitions satisfy these properties almost trivially and hence our abstract domain is condensing in the terminology of [36].

Idempotence, means that performing the unification algorithm twice does not give more information. In other words, the unification algorithm makes use of all of the information in the type descriptions in "one shot". Commutativity means that the order in which we solve ACI equations does not influence the precision of the result. In other words the unification algorithm is confluent. Additivity means that no precision is lost in the abstract domain when performing least upper bound operations. In our case, idempotence is trivial because abstract substitutions are substitutions in the formal sense and the abstract unifiers are idempotent abstract substitutions. Commutativity is trivial

[2] The observant reader will notice that perhaps the c-semantics [21] is more appropriate, as the underlying domain of symbolic types is downwards closed. While this is true, we still choose the s-semantics simply because it is more commonly known.

because the minimal set of most general unifiers is unique and does not depend on the order in which equations are solved. Note that our unification algorithm computes a complete but not minimal set of unifiers and potentially solving the equations in different orders might result in different complete sets. However, since our domain is downwards closed, any complete set of unifiers gives the same set of downwards closed instances of the minimal set of most general unifiers. The least upper bound operation in the abstract domain is trivially additive, because it is defined as set union. Hence $\gamma(\mathcal{I}_1) \cup \gamma(\mathcal{I}_2) = \gamma(\mathcal{I}_1 \sqcup \mathcal{I}_2)$.

Abstracting the s-semantics is straightforward and involves replacing the concrete operations in the definition of Equation (1) by corresponding abstract operations. The type analysis of a program P is then obtained as the least fixed point of the resulting immediate consequence operator $\mathcal{T}_P : \wp(\mathcal{B}^{\alpha}_{\mathcal{V}}) \rightarrow \wp(\mathcal{B}^{\alpha}_{\mathcal{V}})$:

$$\mathcal{T}_P(\mathcal{I}) = \left\{ \sigma(h)\mu \;\middle|\; \begin{array}{l} C \equiv h \leftarrow b_1, \ldots, b_n \in P, \\ \langle b_1^{\alpha}, \ldots, b_n^{\alpha} \rangle \ll_C \mathcal{I}, \\ \mu \in cU_{ACI}(\langle \sigma(b_1), \ldots, \sigma(b_n) \rangle, \langle b_1^{\alpha}, \ldots, b_n^{\alpha} \rangle) \end{array} \right\} \tag{11}$$

The following theorem implies that the fixpoint of this operator exists and can be computed by iterated evaluation.

Theorem 5.1 *$\mathcal{T}_P$ is monotonic and continuous.*

Proof. *Monotonicity is straightforward. Let us show that $\mathcal{T}_P$ is continuous. Let $\mathcal{K}$ be a chain (i.e. a totally ordered set) of abstract interpretations. We have to prove that $\mathcal{T}_P(\bigcup \mathcal{K}) = \bigcup \mathcal{T}_P(\mathcal{K})$.*

- ($\succeq$) *Immediate by monotonicity.*
- ($\preceq$) *Let $h^{\alpha} \in \mathcal{T}_P(\bigcup \mathcal{K})$. Then there exists a clause $(h \leftarrow b_1, \ldots, b_n) \in P$, abstract atoms $b_1^{\alpha}, \ldots, b_n^{\alpha} \in \bigcup \mathcal{K}$, and an abstract unifier*

  $$\mu \in cU_{ACI}\left(\langle \sigma(b_1), \ldots, \sigma(b_n) \rangle, \langle b_1^{\alpha}, \ldots, b_n^{\alpha} \rangle\right)$$

  such that $h^{\alpha} = \sigma(h)\mu$. Then there exist interpretations $\mathcal{I}_1, \ldots, \mathcal{I}_n \in \mathcal{K}$ such that $b_1^{\alpha} \in \mathcal{I}_1, \ldots, b_n^{\alpha} \in \mathcal{I}_n$. Let $\mathcal{I} = \max\{\, \mathcal{I}_1, \ldots, \mathcal{I}_n \,\}$. Then $h^{\alpha} \in \mathcal{T}_P(\mathcal{I})$ and $\mathcal{T}_P(\mathcal{I}) \preceq \bigcup \mathcal{T}_P(\mathcal{K})$.

□

Corollary 5.2 *The least fixpoint of $\mathcal{T}_P$ exists and*

$$lfp(\mathcal{T}_P) = \bigcup_{n \in \omega} (\mathcal{T}_P)^n(\emptyset).$$

Proof. *By the Knaster-Tarski theorem [42].* □

We prove that least fixed point of $\mathcal{T}_P$ correctly approximates the atoms contained in the least fixed point of T_P.

Theorem 5.3 (safety)

$$\forall I \in \wp(\mathcal{B}_{\mathcal{V}}) : \mathcal{T}_P(\alpha(I)) \propto T_P(I).$$

PROOF. *Assume there is an abstract atom* $h^\alpha \in \alpha(T_P(I))$. *Then there is a clause* $(h \leftarrow b_1, \ldots, b_n) \in P$, *atoms* $b'_1, \ldots, b'_n \in I$ *which are renamed apart and a unifier* $\theta = mgu(\langle b_1, \ldots, b_n\rangle, \langle b'_1, \ldots, b'_n\rangle)$ *such that* $h^\alpha = \sigma(h\theta)$. *Then* $\alpha(I)$ *contains abstract atoms* $\sigma(b'_1), \ldots, \sigma(b'_n)$ *which are also renamed apart. By Theorem 4.4 and Corollary 4.5 there exists an abstract unifier* $\mu \in cU_{ACI}(\langle\sigma(b_1), \ldots, \sigma(b_n)\rangle, \langle\sigma(b'_1), \ldots, \sigma(b'_n)\rangle)$ *such that* $\sigma(h)\mu$ *approximates* $h\theta$, *i.e.* $h^\alpha \preceq \sigma(h)\mu$. *Therefore,* $\alpha(T_P(I)) \preceq \mathcal{T}_P(\alpha(I))$, *or equivalently* $\mathcal{T}_P(\alpha(I)) \propto T_P(I)$. □

Theorem 5.4 (partial correctness) $lfp(\mathcal{T}_P) \propto lfp(T_P)$.

PROOF. *By induction we prove* $\forall n \in \omega : (\mathcal{T}_P)^n(\emptyset) \propto (T_P)^n(\emptyset)$ *with the trivial basis* $\emptyset \propto \emptyset$ *and the induction step provided by Theorem 5.3. Then the claim of the theorem follows by Corollary 5.2.* □

<u>Termination</u>: For monomorphic analysis, termination is a straightforward consequence of the finiteness of the abstract domain. In contrast, polymorphic type analysis as defined above are not guaranteed to terminate. In our implementation, reasonable results are obtained using a simple depth-k abstraction.

Theorem 5.5 (termination) $lfp(\mathcal{T}_P)$ *is finitely computable for monomorphic type abstraction.*
PROOF. *Immediate by Theorem 3.2 and Corollary 5.2.* □

The following example illustrates the non-termination of a polymorphic type analysis.

Example 11 *Consider the program:*

```
p([X]) :- p(X).
p(0).
```

The model of this program consists of atoms in the form: $p(0)$, $p([0])$, $p([[0]])$ *and so on. The corresponding polymorphic type expressions based on the abstraction of Equation (5) in the abstract model are:* $p(num)$, $p(list(num) \oplus nil)$, $p(list(list(num) \oplus nil) \oplus nil)$ *etc. These expressions correspond to an infinite sequence of types: "number", "list of numbers", "list of lists of numbers" etc.*

Depth-k abstraction is applied to approximate abstract atoms. The "depth" of an abstract atom is the maximal depth of the symbolic types in its arguments and reflects the number of levels of polymorphism. The value of k can be determined by the user and may depend on the precision requirements, as well, on the available computational resources. It is clear that with the growth of k the number of possible types and the number of ACI unifiers for type expressions grow exponentially. For most practical reasons two or three levels of polymorphism in types are sufficient.

Abstract compilation: It is convenient to view a type analysis in our approach as consisting of two phases: First the program is abstracted for the given choice of types. The result is an abstract program which we call a *set logic program* reflecting the fact that its terms may be viewed as "flat sets" specified using the $\oplus/2$ constructor. The analysis is performed in a second stage using an operator similar to that of Equation (11) which is parameterized by an abstract program:

$$\mathcal{T}'_{\sigma(P)}(\mathcal{I}) = \left\{ h^\alpha\mu \left| \begin{array}{l} C \equiv h^\alpha \leftarrow b_1^\alpha, \ldots, b_n^\alpha \in \alpha(P), \\ \langle b'_1, \ldots, b'_n \rangle \ll_C \mathcal{I}, \\ \mu \in cU_{ACI}(\langle b_1^\alpha, \ldots, b_n^\alpha \rangle, \langle b'_1, \ldots, b'_n \rangle) \end{array} \right. \right\} \tag{12}$$

This approach is sometimes referred to as *abstract compilation* and is clearly equivalent to the abstract interpretation described above. Namely $lfp(\mathcal{T}'_{\sigma(P)}) = lfp(\mathcal{T}_P)$.

The meaning of an abstract program is easily evaluated using a simple Prolog meta-interpreter similar to those described in [15, 17, 14]. The main difference is that the interpreter must be enhanced replacing standard unification by ACI-unification. A simple Prolog interpreter for symbolic types is given in Appendix B. Appendix A contains the code implementing the ACI-unification algorithm for monomorphic types. A complete implementation of the monomorphic type analyzer for both goal-dependent and goal-independent analyses is available from `ftp://ftp.cs.bgu.ac.il/pub/people/codish/monotypes.tar.gz`.

Examples: The following two examples illustrate the bottom-up analysis of monomorphic types based on Equation (4) using abstract compilation. The result of an analysis is a goal-independent representation of the type dependencies in the program. Given the result of an analysis and given an abstract atom describing the (input) types in a call pattern, we can derive a description of the (output) types in the corresponding success patterns. This is because the s-semantics characterizes the answers for arbitrary goals and hence, the abstract semantics approximates the answers for arbitrary goals. The formal justification of this is described in [5] and [13]. Similar proofs for the *Pos* domain are given in [15]. In addition to answers (or success patterns), call patterns can be approximated by applying a suitable program transformation such as Magic Sets. A detailed description of this approach with an application for the *Pos* domain can be found in [15].

Example 12 *Consider the "append" program (left) and its monomorphic type abstraction (right):*

```
append([],Ys,Ys).                  append(list,Ys,Ys)
append([X|Xs],Ys,[X|Zs]) :-        append(list⊕Xs,Ys,list⊕Zs) :-
   append(Xs,Ys,Zs).                  append(Xs,Ys,Zs).
```

We describe the step by step evaluation of the least fixed point of the operator of Equation (12) for the abstract program. In the first iteration we obtain from the first clause

$$\mathcal{I}_1 = \mathcal{T}'_{\sigma(P)}(\emptyset) = \left\{ \ append(list, Ys, Ys) \ \right\}.$$

In the second iteration a renaming of the fact in $\mathcal{I}_1$ *is unified with the body of the second clause producing the binding*

$$\left\{ \ Xs \mapsto list, Ys \mapsto Ys', Zs \mapsto Ys' \ \right\}$$

and we obtain

$$\mathcal{I}_2 = \mathcal{T}'_{\sigma(P)}(\mathcal{I}_1) = \left\{ \begin{array}{l} append(list, Ys, Ys) \\ append(list, Ys', list \oplus Ys') \end{array} \right\}.$$

The next iteration generates the abstract atom $append(list \oplus list, Ys', list \oplus list \oplus Ys')$ *which is equivalent to* $append(list, Ys', list \oplus Ys')$ *and so* $\mathcal{I}_2$ *is a fixed point. The atoms in* $\mathcal{I}_2$ *can be "queried" to describe the success patterns for various initial goals. For instance:*

`?- append(list,list,X).`	`?- append(X,Y,list).`
`X = list`	`X = list,    Y = list`

A call to "append" in which the first two arguments are lists is answered by a substitution which binds the third argument to a list. A call to "append" in which the third argument is a list is answered by a substitution which binds the first two arguments to lists.

Example 13 *Consider a concrete program specifying the "inorder" relation on a binary tree (left) and its monomorphic type abstraction (right) together with the clauses from Example 12:*

```
inorder(void,[]).
inorder(tree(X,L,R),I) :-
   inorder(L,IL), inorder(R,IR),
   append(IL,[X|IR],I).
```

```
inorder(tree,list).
inorder(tree⊕L⊕R,I) :-
   inorder(L,IL), inorder(R,IR),
   append(IL,list⊕IR,I).
```

The analysis of the abstract "inorder" program produces the following atoms:

$$lfp(\mathcal{T}_{\sigma(P)}) = \left\{ \begin{array}{l} inorder(tree, list) \\ append(list, A, A) \\ append(list, A, list \oplus A) \end{array} \right\}$$

The result of the analysis indicates that a query to "inorder" in which the first argument is a binary tree is answered by a substitution which binds the second argument to a list.

The goal-dependent type analysis of "inorder" can be obtained using bottom-up abstract semantics applied to the result of the Magic Sets transformation [6] of the program. We denote the result of the transformation of a program P with respect to a *goal G by $M(P;G)$. The result of the analysis based on Magic Sets for an initial goal of the form $inorder(tree, X)$ results in the following set of atoms:*

$$lfp(\mathcal{T}_{\sigma(M(P;G))}) \left\{ \begin{array}{l} call\_inorder(tree, X), \\ call\_append(list, list, Y), \\ answer\_inorder(tree, list), \\ answer\_append(list, list, list) \end{array} \right\}$$

indicating that all of the calls to inorder/2 in the computations are of the same form as the initial call, and that the predicate append is used in this program only to concatenate lists.

The next example illustrates a polymorphic type analysis based on the abstraction of Equation (5). The example is taken from [9] where the authors argue that the types in this program cannot be computed precisely in type systems which do not maintain dependency information. This is exactly the strong point of our domain and we can handle this example successfully.

Example 14 *Consider the following task: Given a binary tree T whose nodes are labeled with natural numbers, compute a binary tree with the same structure as T, but where every node is labeled with the maximal number in T. The following Prolog program solves this task in one pass, first constructing a binary tree in which all nodes are labeled by the same variable Max, and then unifying Max with the maximal number found in the input tree:*

```
maxtree(Tree, NewTree) :- maxtree(Tree, Max, Max, NewTree).

maxtree(void, _, 0, void).
maxtree(tree(X,L,R), Max, MaxSoFar, tree(Max,NewL,NewR)) :-
   maxtree(L, Max, MaxL, NewL),  maxtree(R, Max, MaxR, NewR),
   max3(X,MaxL,MaxR,MaxSoFar).

max3(A,B,C,Max) :- max(A,B,MaxAB), max(MaxAB,C,Max).
max(A,B,A) :- A >= B.
max(A,B,B) :- A < B.
```

The abstract bottom-up semantics of the above program computed by our analyzer is the following:[3]

$$\left\{ \begin{array}{l} maxtree(void,\ void) \\ maxtree(tree(num) \oplus void,\ tree(num) \oplus void) \\ maxtree(void,\ A, num, void) \\ maxtree(tree(num) \oplus void,\ A,\ num,\ tree(A) \oplus void) \\ maxtree(tree(num) \oplus void,\ num,\ num,\ tree(num) \oplus void) \\ max3(num, num, num, num) \\ max(num, num, num) \end{array} \right\} \qquad (13)$$

[3] We assume that Prolog built-ins are given suitable definitions in abstract programs. For example `num<num` and `num>=num`.

The analysis captures correctly (and precisely) the fact that both arguments of `maxtree/2` *are bound to binary trees of numbers upon success. It is interesting to observe that there are three different call patterns to* `maxtree/4` *in a computation starting from the call of the type* $maxtree(tree(num) \oplus void,\ X)$*. Assuming that the initial call is of the form* $maxtree(tree(num) \oplus void,\ X)$ *and the evaluation strategy is from left-to-right, the fist call to* `maxtree/4` *is of the form* $maxtree(tree(num) \oplus void, A, A, B)$*. The recursive calls are of the form* $maxtree(tree(num) \oplus void, A, B, C)$ *or* $maxtree(void, A, B, C)$*. The answers for these call patterns are obtained by "solving" them with the elements in Equation (13):*

```
?- maxtree(tree(num)⊕void, A, A, B).
A = num, B = tree(num)⊕void.

?- maxtree(tree(num)⊕void, A, B, C).
A = X, B = num, C = tree(X)⊕void; (or)
A = num, B = num, C = tree(num)⊕void.

?- maxtree(void, A, B, C).
A = X, B = num, C = void.
```

Example 15 *Consider again the result of a polymorphic type dependency analysis of the append relation described in Example 8.*

$$\mathcal{I} = \left\{ \begin{array}{l} append(nil,\ A,\ A), \\ append(list(A) \oplus nil,\ B,\ list(A) \oplus B) \end{array} \right\}$$

It is interesting to observe some of the directional types expressed by $\mathcal{I}$*:*

1. *For a call to append in which the first two arguments are lists of type A:*
   ```
   ?- append(list(A)⊕nil, list(A)⊕nil, Z).
   Z = list(A)⊕nil
   ```
2. *For a call to append in which the first two arguments are lists of type A and type B:*
   ```
   ?- append(list(A)⊕nil, list(B)⊕nil, Z).
   Z = list(A)⊕list(B) ⊕nil
   ```
3. *For a call to append in which the third argument is a list of type A:*
   ```
   ?- append(X,Y,list(A)⊕nil).
   X = list(A)⊕nil, Y = list(A)⊕nil;
   X = nil, Y = list(A)⊕nil;
   X = list(A)⊕nil, Y = nil.
   ```

An implementation: The implementation of the described type analysis is in principle based on the simple Prolog interpreter described in Appendix B. Several additional optimizations have been introduced to the actual prototype: We have based the analysis on a semi-naive interpreter which is described in [12]. For goal-dependent analysis, we used the interpreter for *induced Magic Sets* described in [12] as an alternative to applying the Magic Sets transformation. The implementation is small, consisting of some 200 lines of Prolog code.

In addition to the Prolog implementation we have experimented with an implementation based on tabled resolution implemented in XSB which is described in [16]. The experimental evaluation of the technique indicates that in spite of the fact that the basic underlying unification algorithm is NP-complete analyses are usually quite fast. It appears that the actual terms being unified in program analyses tend to be small, often having only a few most general unifiers. A collection of small and medium sized benchmarks have been obtained with reasonable timings. These are described in [17] and [16].

6 Conclusion

This paper presents a new notion of typing for logic programs which generalizes the notion of directional types. We assume that a particular notion of types is provided by the user and then automatically infer type dependencies for a given program with respect to this definition. The analysis method is based on a novel combination of program abstraction and ACI-unification which is shown to be correct and optimal. Type dependencies are obtained by abstracting programs, replacing concrete terms by their types, and evaluating the meaning of the abstract programs using a standard semantics for logic programs enhanced by ACI-unification. This approach is generic and can be used with any standard semantics. The domain of symbolic types is shown to be condensing which means that no precision is lost when performing bottom-up or goal-independent analysis.

A similar approach based on ACI1-unification has recently been applied for sharing analysis of logic programs [18]. A complete description of both types and sharing analyses can be found also in [32].

Acknowledgment This paper has benefited from the discussions and comments of several colleagues: Bart Demoen collaborated on an earlier paper which influenced this one and later helped get the benchmarks for this paper working fast. Konstantinos Sagonas helped port the implementation to XSB which made the benchmarks work even faster. Giorgio Levi insisted that we check that the domain of symbolic types is condensing. Robert Constable, Saumya Debray, John Gallagher and Roberto Giacobazzi made useful comments on an earlier version.

Appendix A: ACI-unification Procedure

The code of ACI-unification presented below implements the unification of monomorphic types. It is based on the algorithm of [34] restricted and optimized for the expressions containing only constants and variables. The algorithm relies on the notion of a *unification graph* corresponding to the unification problem.

Definition A.1 *Let τ_1 and τ_2 be abstract terms with the corresponding sets of constitutes (constants and variables) V_1 and V_2 respectively. The unification graph corresponding to ACI-unification of τ_1 and τ_2 is the bipartite graph $G_{\tau_1,\tau_2} = (V_1 \cup V_2, E)$, where $(u,v) \in E$ iff $u \in V_1$, $v \in V_2$ and u is unifiable with v.*

Example A.1 *Figure 1 illustrates the unification graphs corresponding to the ACI-unification of $A \oplus a$ and $B \oplus b$ (left) and of $A \oplus B$ and $C \oplus d$ (right).*

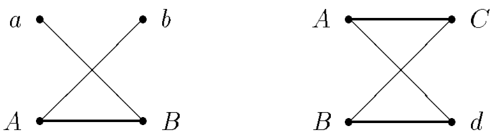
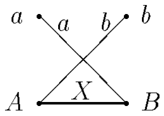


Figure 1: Unification graphs for $A \oplus a \stackrel{?}{=} B \oplus b$ and $A \oplus B \stackrel{?}{=} C \oplus d$.

The algorithm of unification of two monomorphic types τ_1 and τ_2 starts with construction of a graph $G_{\tau_1,\tau_2} = (V, E)$ corresponding to this unification. Then it marks each edge $(u,v) \in E$ by a most general common instance of u and v. On the following stages the algorithm incrementally chooses subsets of edges $E' \subseteq E$ such that the subgraphs $G'_{\tau_1,\tau_2} = (V, E')$ have no disconnected nodes. If no such a subset of edges exists the unification fails. Upon success of this stage the algorithm converts the subgraph G'_{τ_1,τ_2} to a unifier of two types as follows. For each node $v \in V$ corresponding to a variable X of τ_1 or τ_2 an equation $X = z_1 \oplus \ldots \oplus z_n$ is constructed, where $z_1 \ldots z_n$ are the labels of the edges of E' adjacent to v. Note that if τ_1 and τ_2 have a common variable, two nodes of G_{τ_1,τ_2} correspond to this variable. The unifier ψ of τ_1 and τ_2 is the set of substitutions $X \mapsto z_1 \oplus \ldots \oplus z_n$. We can see that ψ indeed unifies τ_1 and τ_2 observing that $\tau_1\psi$ and $\tau_2\psi$ are composed from the set of labels of edges in E.

Example A.2 *Consider the corresponding graph for the unification of $A \oplus a$ with $B \oplus b$ (Figure 1, left). We mark each edge of this graph by most general instances of corresponding nodes as described above. The graph we obtain is:*



Note that the edge between A and B is marked by X which is a most general common instance of A and B modulo renaming. Now we construct a unifier considering all the edges of the unification graph and binding variables of the expressions to the sets of labels of corresponding adjacent edges. Namely, we *produce a unifier $\psi = \{\ A \mapsto (X \oplus b),\ B \mapsto (X \oplus a)\ \}$. The other unifier computed by our algorithm for this case is obtained by considering only the edges (a, B) and (b, A). Namely, $\psi' = \{\ A \mapsto b,\ B \mapsto a\ \}$. Note that ψ' is subsumed by ψ.*

The unification of polymorphic types is performed by almost the same technique. However, the phase of solving equations constructed from the subgraph G'_{τ_1,τ_2} must be extended to handle also polymorphic constitutes. Similarly to the general case of concrete unification, polymorphic constitutes with the same principal functor are unified recursively by ACI-unification of their arguments.

```
%%-------------------------------------------------------------------
%% Unify two monomorphic types A and B. Both A and B are assumed to be
%% flat sorted lists of their constitutes. A complete set of solutions
%% is obtained by backtracking through this code.
%%
%% Library "ordsets" (Quintus) or similar for manipulating
%% sets in ordered list representation is required.

aci_unify(A,B) :-
        (ground(A), ground(B) -> A==B
          ;
         ord_union(A,B,ToCover), ord_setproduct(A,B,AxB),
         drop_nonuni(AxB,UTable),
         covering_u_table(UTable,ToCover,CovUTable),
         keysort(CovUTable,CovUTableS),
         table2graph(CovUTableS,CovUGraph),
         u_graph_proceed(CovUGraph)
        ).

%% Throw away pairs of non-unifiable terms (nonexistent edges in graph)
drop_nonuni([],[]).
drop_nonuni([X-Y|AxB],Out) :-
        (\+ \+ X=Y -> Out = [X-Y|NewAxB] ; Out = NewAxB),
        drop_nonuni(AxB,NewAxB).

%% Compute a set of edges covering all nodes in the unfication graph.
%% On backtrack this predicate generates all the solutions.
covering_u_table([],[],[]).
covering_u_table([X-Y|XsYs],ToCover,[X-L,Y-L|UTRest]) :-
        (X @< Y ->
         ord_subtract(ToCover,[X,Y],ToCoverRest)
          ;
         ord_subtract(ToCover,[Y,X],ToCoverRest)),
        covering_u_table(XsYs,ToCoverRest,UTRest).
covering_u_table([_|XsYs],ToCover,UTable) :-
        covering_u_table(XsYs,ToCover,UTable).
```

```
%% Convert a list of edges [Vi-Vj,Vm-Vn,...] to graph
%% representation: [ V1-[Vm,...,Vn], V2-[Vr,...,Vs], ... ].
table2graph([],[]).
table2graph([V-I],[V-[I]]) :- !.
table2graph([V-I,W-J|Vs],Out) :-
        ( V==W ->
          Out = [V-[I|Is]|VsClps], Cont = [V-Is|VsClps]
          ;
          Out = [V-[I]|VsClps], Cont = VsClps),
        table2graph([W-J|Vs],Cont).

%% Try to unify each node with the set of nodes linked to it.
u_graph_proceed([]).
u_graph_proceed([V-E|G]) :- (var(V) -> V = E ; match_list(E,V)),
        u_graph_proceed(G).

match_list([],_).
match_list([S|Xs],S) :- match_list(Xs,S).
```

Appendix B: A Simple Prolog interpreter for symbolic types

The meaning of an abstract program is easily evaluated using a simple Prolog meta-interpreter similar to those described in [15, 17, 14]. The main difference is that the interpreter must be enhanced replacing standard unification by ACI-unification.

The evaluator depicted in Figure 2 is very similar to the one presented in [15]. The main difference is that standard unification is replaced by ACI unification (the predicate `unify/2`) and a normal form for abstract terms is maintained (the predicate `canonical/2`) so that the variance test for newly computed atoms degenerates into a syntactic variance check. Each clause $h \leftarrow b_1, \ldots, b_n$ in the abstract program is represented as a fact of the form $abs\_clause(h, (b_1, \ldots, b_n))$, which is added to the clauses in Figure 2. Initially, only `fact(true)` is known. New facts derived by an iteration of the predicate `tp/0` are asserted to the program if their semantic variants are not already there. The evaluation is initiated by the query `?- iterate_tp` which leaves the result of the analysis in the Prolog database as a set of facts of the form `fact(Atom)`.

```
iterate_tp :- tp.
iterate_tp :- retract(database_changed), iterate_tp.

tp :- abs_clause(H,B), prove(B), canonical(H,HS),
                cond_assert(fact(HS)), fail.

prove((B,Bs)) :- fact(BX), unify(B,BX), prove(Bs).
prove(B) :- fact(BX), unify(B,BX).

cond_assert(G) :- in_database(G) , !.
cond_assert(G) :- assert(G), cond_assert(database_changed).

in_database(G) :- clause(B,true), variant(B,G).

fact(true).
```

Figure 2: Goal-Independent Evaluator Based on $\mathcal{T}_P$

References

[1] A. Aiken and T. K. Lakshman. Directional Type Checking of Logic Programs. In Le Charlier [33], pages 43–60.

[2] K. R. Apt. Introduction to Logic Programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models

and Semantics, pages 495–574. Elsevier, Amsterdam and The MIT Press, Cambridge, 1990.

[3] K. R. Apt. Declarative programming in Prolog. In D. Miller, editor, *Logic Programming - Proceedings of the 1993 International Symposium*, pages 12–35, Vancouver, Canada, 1993. MIT Press.

[4] F. Baader and J. Siekmann. Unification theory. In C. Hogger D. Gabbay and J. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 2, pages 41–126. Oxford Science Publications, 1994.

[5] R. Bagnara, R. Giacobazzi, and G. Levi. An Application of Constraint Propagation to Data-flow Analysis. In *Proc of Ninth IEEE Conference on AI Applications*, pages 270–276, Orlando FL, 1993. IEEE Computer Society Press.

[6] F. Bancilhon, D. Maier, Y. Sagiv, and J.D. Ullman. Magic Sets and Other Strange Ways to Implement Logic Programs. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 1–15, 1986.

[7] R. Barbuti and R. Giacobazzi. A bottom-up polymorphic type inference in logic programming. *Science of Computer Programming*, 19(3):281–313, December 1992.

[8] A. Bossi, M. Gabbrielli, G. Levi, and M. Martelli. The s-semantics approach: Theory and applications. *Journal of Logic Programming*, 1994. to appear.

[9] J. Boye. *Directional Types In Logic Programming.* PhD thesis, Linköping University (Sweden), 1996.

[10] J. Boye and J. Małuszyński. Two aspects of directional types. In L. Sterling, editor, *Proceedings of the Twelfth International Conference on Logic Programming*, Kanagawa, Japan, 1995. The MIT Press.

[11] F. Bronsard, T. K. Lakshman, and U. S. Reddy. A framework of directionality for proving termination of logic programs. In Apt, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 321–335, Washington, USA, November 1992. The MIT Press.

[12] M. Codish. Efficient goal directed bottom-up evaluation of logic programs. Technical report, Ben-Gurion University, January 1997. `ftp://ftp.cs.bgu.ac.il/pub/people/codish/fastmagic.ps.gz`.

[13] M. Codish, D. Dams, and E. Yardeni. Bottom–Up Abstract Interpretation of Logic Programs. *Theoretical Computer Science*, 124(1):93–125, February 1994.

[14] M. Codish and B. Demoen. Deriving Polymorphic Type Dependencies for Logic Programs using Multiple Incarnation of Prop. In Le Charlier [33], pages 281–296.

[15] M. Codish and B. Demoen. Analyzing logic programs using "PROP"-ositional logic programs and a magic wand. *Journal of Logic Programming*, 25(3):249–274, December 1995.

[16] M. Codish, B. Demoen, and K. Sagonas. General purpose semantic based analysis using XSB. Technical report, Ben-Gurion University of the Negev, January 1997. `ftp://ftp.cs.bgu.ac.il/pub/people/codish/absxsb.ps`.

[17] M. Codish and V. Lagoon. Type dependencies for logic programs using ACI-unification. In *Proc. Fourth Israel Symposium on Theory of Computing and Systems*, pages 136–145. IEEE Computer Society Press, 1996.

[18] M. Codish, V. Lagoon, and F. Bueno. An algebraic approach to sharing analysis of logic programs. In P. Van Hentenryck, editor, *Proceedings of the Fourth International Static Analysis Symposium*, volume 1302 of *LNCS*, pages 68–82. Springer Verlag, sep 1997.

[19] A. Cortesi and G. Filé. Abstract Interpretation of Logic Programs: an Abstract Domain for Groundness, Sharing, Freeness and Compoundness Analysis. In P. Hudak and N. D. Jones, editors, *Proceedings of the ACM SIGPLAN Symposium on partial evaluation and semantics based program manipulation, PEPM'91*, number 26 in Sigplan notices, pages 52–61, 1991. also available as Technical Report 4/91 of Dip. di Matematica Università di Padova (Italy).

[20] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc., Fourth ACM Symp. on Principles of Programming Languages*, pages 238–252. ACM Press, January 1977.

[21] M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. Declarative Modeling of the Operational Behavior of Logic Languages. *Theoretical Computer Science*, 69(3):289–318, 1989.

[22] M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. A Model-Theoretic Reconstruction of the Operational Semantics of Logic Programs. *Information and Computation*, 102(1):86–113, 1993.

[23] T. Frühwirth. Type inference by program transformation and partial evaluation. In H. Abramson and M. H. Rogers, editors, *Meta-Programming in Logic Programming*, pages 263–282. The MIT Press, 1989.

[24] T. Frühwirth, E. Shapiro, M. Y. Varai, and E. Yardeni. Logic programs as types for logic programs. In Albert R. Meyer, editor, *Proceedings of the 6th Annual IEEE Symposium on Logic in Computer Science*, pages 300–309, Amsterdam, The Netherlands, July 1991. IEEE Computer Society Press.

[25] J. Gallagher and D. A. de Waal. Fast and Precise Regular Approximation of Logic Programs. In Van Hentenryck [43], pages 599–613. ISBN 0-262-72022-1.

[26] R. Giacobazzi, S. K. Debray, and G. Levi. A Generalized Semantics for Constraint Logic Programs. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 581–591, ICOT, Japan, June 1992. ACM, Association for Computing Machinery.

[27] C. A. Gunter and D. S. Scott. Semantic domains. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 8, pages 633–674. North-Holland, New York,, 1990.

[28] M. Hermenegildo, R. Warren, and S. K. Debray. Global Flow Analysis as a Practical Compilation Tool. *Journal of Logic Programming*, 13(4):349–366, August 1992.

[29] D. Jacobs and A. Langen. Static Analysis of Logic Programs for Independent and Parallelism. *Journal of Logic Programming*, 13(1, 2, 3 and 4):291–314, 1992.

[30] D. Kapur and P. Narendran. Complexity of unification problems with associative-commutative operators. *Journal of Automated Reasoning*, 9(2):261–288, October 1992.

[31] M. Kifer and J. Wu. A first-order theory of types and polymorphism in logic programming. In *IEEE Symposium on Logic in Computer Science*, 1991.

[32] V. Lagoon. Logic program analysis using set logic programs. Master's thesis, Ben-Gurion University of the Negev (Israel), 1998.

[33] Baudouin Le Charlier, editor. *1st International Symposium on Static Analysis*, volume 864 of *LNCS*, Namur, Belgium, September 1994. Springer Verlag.

[34] P. Lincoln and J. Christian. Adventures in associative-commutative unification. *Journal of Symbolic Computation*, 8:217–240, 1989. Also appears in *Unification*, edited by Claude Kirchner (Academic, 1990), pages 393–416.

[35] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Berlin, 1987. Second edition.

[36] K. Marriott and H. Søndergaard. Precise and Efficient Groundness Analysis for Logic Programs. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 2(4):181–196, March 1993.

[37] D. Pedreschi. A proof method for run-time properties of Prolog programs. In Van Hentenryck [43], pages 584–598. ISBN 0-262-72022-1.

[38] Frank Pfenning, editor. *Types in Logic Programming*, Cambridge, Massachusetts, 1992. MIT Press.

[39] C. Pyo and U.S. Reddy. Inference of Polymorphic Types for Logic Programming. In E. Lusk and R. Overbeek, editors, *Proc. North American Conf. on Logic Programming'89*, pages 1115–1132. The MIT Press, Cambridge, Mass., 1989.

[40] Y. Rouzaud and L. Nguyen-Phuong. Integrating modes and subtypes into a Prolog type-checker. In Apt, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 85–97, Washington, USA, 1992. The MIT Press.

[41] M. E. Stickel. A unification algorithm for associative-commutative functions. *Journal of the ACM*, 28(2):423–434, April 1981.

[42] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math.*, 5:285–309, 1955.

[43] Pascal Van Hentenryck, editor. *Proceedings of the Eleventh International Conference on Logic Programming*, Santa Margherita Ligure, Italy, 1994. The MIT Press. ISBN 0-262-72022-1.

[44] J. Xu and D. S. Warren. A type inference system for Prolog. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 604–619, Seatle, 1988. ALP, IEEE, The MIT Press.