A Framework for Analysis of Typed Logic Programs

Vitaly Lagoon and Peter J. Stuckey

Dept. of Computer Science and Software Engineering
University of Melbourne, Parkville 3052, Australia
{lagoon,pjs}@cs.mu.oz.au

Abstract. The paper presents a novel approach to the analysis of typed logic programs. We assume regular type descriptions of logic program variables provided by *regular tree grammars*. Types are used to identify components of terms which can be equated during the programs execution. This information is reflected in form of *set constraints* in the generic abstract domain called Type(X). The domain allows for abstract compilation of typed logic programs into *set logic programs*. We demonstrate how the analyzers of *groundness* and *sharing* of typed logic programs are obtained by applying the existing (untyped) techniques based on Pos and Sharing to set logic programs. The corresponding analyses in Type(Pos) and Type(Sharing) are more precise than those in Pos and Sharing.

1 Introduction

Recent logic programming languages Gödel [10], Mercury [19] and HAL [5] are strongly typed, using variations of Hindley-Milner [15] type systems. Prescriptive type information is useful for detecting many simple programming errors, and gives a new source of information to use in optimizing compilation. The advantages of type information in efficient execution are well illustrated by the performance of Mercury programs. Additionally systems [6] exist for automatically inferring regular tree types from programs. Once such a type is inferred we can use the information in later analyses to improve the accuracy of the results.

In this paper we examine how we can analyze (regular-tree) typed logic programs. Analysis of typed logic programs is important for optimizing compilation of Mercury and HAL.

Types allow analyses to be more accurate than analyses of the corresponding untyped programs. For example consider the analysis of the typed "append" program shown in Figure 1. The types of A, B, C, D and E are lists. A typed analysis separates a list variable v into two parts, the skeleton of the list L_v, and the elements of the list E_v. The analysis creates a CLP(B) program relating the boundness of these different parts of the list. The bottom-up abstract evaluation of abstract "append" results in:

$$\texttt{append}(\langle L_A, E_A\rangle, \langle L_B, E_B\rangle, \langle L_C, E_C\rangle) \texttt{ :- } L_A,\ L_B \leftrightarrow L_C,\ E_C \leftrightarrow E_A \wedge E_B.$$

`append(A,B,C) :-`	`append`$(\langle L_A, E_A\rangle, \langle L_B, E_B\rangle, \langle L_C, E_C\rangle)$ `:-`
`    A = [],`	$\quad L_A \leftrightarrow \mathsf{T}$, $E_A \leftrightarrow \mathsf{T}$,
`    B = C.`	$\quad L_B \leftrightarrow L_C$, $E_B \leftrightarrow E_C$.
`append(A,B,C) :-`	`append`$(\langle L_A, E_A\rangle, \langle L_B, E_B\rangle, \langle L_C, E_C\rangle)$ `:-`
`    A = [X|D],`	$\quad L_A \leftrightarrow L_D$, $E_A \leftrightarrow E_X \wedge E_D$,
`    C = [X|E],`	$\quad L_C \leftrightarrow L_E$, $E_C \leftrightarrow E_X \wedge E_E$,
`    append(D,B,E).`	$\quad$ `append`$(\langle L_D, E_D\rangle, \langle L_B, E_B\rangle, \langle L_E, E_E\rangle)$.

Fig. 1. Abstraction of `append`

showing that on success the list skeleton of A is fixed and the skeleton of B is fixed if and only if that of C is, and the elements are related in the usual way. This more specific information can be used to improve program optimizations. The usual groundness analysis of "append" only gives the answer $C \leftrightarrow A \wedge B$, i.e. C is ground if and only if A and B are ground. With this information it is not possible to determine that the goal `append([A,B],[C,D],E),append(E,F,G)` will result in the list skeleton of E being fixed. This might prevent a partial evaluator evaluating the second call to "append".

It is important to stress that the type-based analyses proposed in this paper are not only more expressive as can be seen from the above example but also more precise in a strict sense. As a simplest example consider the unification $f(A, B) = f(C, D)$. While the standard Pos approximation of groundness produces $(A \wedge B) \leftrightarrow (C \wedge D)$, in our type-based analysis we may get $(A \leftrightarrow C) \wedge (B \leftrightarrow D)$, which is clearly a more precise Pos approximation.

The rest of the paper is organized as follows. In the next section we give some preliminary definitions, in particular those for deterministic tree automata and deterministic regular tree grammars. In Section 3 we define how we can relate the types of variables occurring in constraints. In Section 4, we present the parametric abstract domain Type(X) which adds type information to an abstract domain X, as well as the corresponding abstraction of typed logic programs into programs over set constraints. In Sections 5 and 6 we present the techniques of groundness and sharing analysis of typed logic programs based on Type(X). Finally, in Section 7 we provide a brief comparison with some related works and conclude.

2 Preliminaries

In the following we assume a familiarity with the standard definitions and notation for logic programs as described in [13] and with the framework of abstract interpretation [4]. For a 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 $\mathcal{V}$. The set of atoms constructed using predicate symbols from Π and terms from $T(\Sigma, \mathcal{V})$ is denoted by $\mathcal{B}$. The set of variables occurring in a syntactic object s is denoted vars(s). We say that a term t is ground if vars$(t) = \emptyset$.

We assume the rules of the program are in *canonical form*. That is, procedure calls and rule heads are normalized such that parameters are distinct variables, unifications are broken into sequences of simple *unification constraints* and nested terms are flattened by introducing intermediate variables. Unification constraints are elements of the set $\mathcal{C}$ and can be either in the form $X = Y$, where both X and Y are variables, or in the form $X = f(Y_1, \ldots, Y_n)$, where $f/n \in \Sigma$, $n \geq 0$ and $X, Y_1, \ldots, Y_n$ are variables. A body of a rule in canonical form contains only unification constraints and procedure calls in the form $p(X_1, \ldots, X_n)$ where $p/n \in \Pi$ is a predicate symbol and $X_1, \ldots, X_n$ are distinct variables (parameters). In addition, the heads of each rule for predicate p are required to be identical atoms $p(X_{p1}, \ldots, X_{pn})$, and every variable not appearing in the head of a rule is required to appear in exactly one rule. It is straightforward to convert any logic program to this canonical form. The concrete (left-hand) side of "append" shown in Figure 1 is in canonical form. The domain of *facts* is denoted by $\mathcal{B_C}$ and consists of rules of the form $p(X_1, \ldots, X_n) \leftarrow C_1, \ldots, C_k$, where $C_1 \ldots C_k$ are unification constraints.

Regular Types Deterministic regular types are defined as a class of languages accepted by top-down deterministic regular tree automata [7]. The analysis presented below can be performed if such regular types are either given explicitly as in Mercury,[1] or inferred through any mechanism (for example [6]).

Definition 1 (Top-Down Deterministic Tree Automata). *A top-down deterministic finite tree automaton (top-down DFTA) is a tuple $\mathcal{A} = \langle q_0, Q, \Sigma, \Delta \rangle$, where Q is a set of states, $q_0 \in Q$ is an initial state and Δ is a set of transition rules in the form: $q(f(x_1, \ldots, x_n)) \to f(q_1(x_1), \ldots, q_n(x_n))$, such that no two rules have the same left-hand side.*

Top-down deterministic tree automata accept the class of languages called *regular types*, which is a proper subset of regular tree languages and is closed under intersection.

Example 1. The deterministic tree automata $A = \langle q_L, \{q_L, q_E\}, \{cons/2, a/0, b/0\}, \Delta \rangle$, where $\Delta = \{q_L(\text{nil}) \to \text{nil},\ q_L(\text{cons}(x, y)) \to \text{cons}(q_E(x), q_L(y)),\ q_E(a) \to a,\ q_E(b) \to b\}$ accepts ground lists of a's and b's.

Definition 2 (Regular Tree Grammar). *A regular tree grammar is a tuple $\mathcal{G} = \langle S, W, \Sigma, \Delta \rangle$, where: S is a starting non-terminal, W is a finite set of non-terminal symbols, s.t. $S \in W$, Σ is a set of function symbols, Δ is a set of productions in the form $X \to f(Y_1, \ldots, Y_n)$ s. t. $X, Y_1 \ldots Y_n \in W$ and $f/n \in \Sigma$.*

Definition 2 is more restrictive than in the standard definitions (e.g. [7]). We assume a *normal form* of productions which allows only one function symbol in each rule. However, this restriction is merely syntactic since productions with two or more function symbols can always be converted into a normal form by introducing new non-terminals. For instance, $A \to f(g(X))$ can be rewritten using a new non-terminal G as $A \to f(G)$, $G \to g(X)$.

[1] It is straightforward to convert the Hindley-Milner types of Mercury to (weaker) deterministic regular types.

Definition 3 (Deterministic Regular Tree Grammars). *A regular tree grammar* $\mathcal{G} = \langle V, W, \Sigma, \Delta \rangle$ *as given by Definition 2 is deterministic if for a non-terminal* $X \in W$ *and any two corresponding productions* $(X \to f(Y_1, \ldots, Y_n)) \in \Delta$ *and* $(X \to g(Z_1, \ldots, Z_m)) \in \Delta$ *we always have* $f/n \neq g/m$.

Deterministic regular tree grammars define the same class of languages as top-down deterministic tree automata. Moreover, by assuming the normal form of grammar we achieve the syntactic uniformity of the two formalisms. Transitions of an automaton can be converted into grammar productions and vice versa by associating each non-terminal symbol with a corresponding state of the automaton.

Example 2. The following regular tree grammar clearly defines the same language as accepted by the tree automaton in Example 1: $L \to nil \mid cons(E, L)$, $E \to a \mid b$.

The equivalence of regular tree grammars and tree automata allows us for the clear and expressive representation of derivations as transitions of a *deterministic rewriting system* in the form $N(c) \longrightarrow c$ for a constant c or in the form $N(f(t_1, \ldots, t_n)) \longrightarrow f(N_1(t_1), \ldots, N_n(t_n))$ for $f/n \in \Sigma$ and $n > 0$. Using this notation we say that the grammar $\mathcal{G} = \langle S, W, \Sigma, \Delta \rangle$ accepts (derives) a term t if $S(t) \overset{*}{\longrightarrow}_{\mathcal{G}} t$. Sometimes we restrict our attention to a single path or a segment of a path of a derivation tree starting from the root (S) and reaching a non-terminal N with a subterm t' of t: $S(t) \overset{*}{\longrightarrow}_{\mathcal{G}} N(t')$.

Definition 4 (Acceptance of Non-Ground Terms). *A regular tree grammar* $\mathcal{G}$ *accepts a non-ground term* t*, denoted* $t \in \mathcal{L}(\mathcal{G})$*, if and only if it accepts some ground instance of* t.

Example 3. The grammar shown in Example 2 accepts the terms $\mathrm{cons}(a, \mathrm{cons}(C, \mathrm{nil}))$ and $\mathrm{cons}(C, D)$ but not $\mathrm{cons}(\mathrm{cons}(C, \mathrm{nil}), \mathrm{nil})$.

We assume through the rest of this paper that the term “grammar” refers to a deterministic regular tree grammar provided by Definition 3. We also treat the terms “state” and “non-terminal” as equivalent and interchangeable due to the uniformity of the corresponding formalisms. As a consequence, we make no distinction between a “run” of a top-down deterministic tree automaton and a “derivation” using a deterministic regular tree grammar.

3 Properties of Typed Logic Programs

A *typed logic program* is a program in canonical form, together with an association of the variables of the program to regular tree grammars. For a variable $V \in \mathcal{V}$ we assume a deterministic regular tree grammar $\mathcal{G}_V = \langle S_V, W_V, \Sigma, \Delta_V \rangle$, conforming to Definitions 2 and 3. We assume grammars associated with different variables do not involve the same non-terminal symbol (if not we can rename appropriately).

In addition we require that all calls to a predicate name $p/n \in \Pi$ have the same type (a renaming of the grammar) as the head. This means that all

```
p(X) :- q(X).                p(X) :- X = Z, q(Z).
q(Y) :- ...                  q(Y) :- ...
```

Assuming the types of X, Y and Z are:

$$\begin{array}{ll} \mathcal{G}_X: & L_X \rightarrow nil \mid cons(E_X, L'_X), \;\; L'_X \rightarrow cons(E_X, L_X), \;\; E_X \rightarrow a \mid b. \\ \mathcal{G}_Y: & L_Y \rightarrow nil \mid cons(E_Y, L_Y), \;\; E_Y \rightarrow a \mid b. \\ \mathcal{G}_Z: & L_Z \rightarrow nil \mid cons(E_Z, L_Z), \;\; E_Z \rightarrow a \mid b. \end{array}$$

Fig. 2. Modifying a program so that each predicate call has the same type as the head.

implicit type casts of procedure parameters are converted into explicit unification constraints. For example the program in the left of Figure 2 must be rewritten as on the right.

The analyses presented in this paper assume that the program is well-typed in the sense of Mycroft and O'Keefe [17]. That is, the results of analyses approximate semantics computed through a sequence of type-correct resolution steps. A type-correct resolution step, whether it is top-down or bottom-up, is restricted to considering only a type-correct subset of the result.

Types of Terms Satisfying Program Constraints Let us now lift the concept of typing of program variables to the level of unification constraints. In particular, we are interested in the internal structure of grammars providing the types of terms satisfying the unification constraints.

The type of terms satisfying a constraint $X = Y$ is naturally computed from the associated grammars $\mathcal{G}_X = \langle S_X, W_X, \Sigma, \Delta_X \rangle$ and $\mathcal{G}_Y = \langle S_Y, W_Y, \Sigma, \Delta_Y \rangle$ as a grammar $\mathcal{G}_\cap = \langle S_\cap, W_\cap, \Sigma, \Delta_\cap \rangle$ for the intersection of (languages of) $\mathcal{G}_X$ and $\mathcal{G}_Y$, where $S_\cap = \langle S_X, S_Y \rangle$, $W_\cap$ is a reachable (through the rules of $\Delta_\cap$) subset of $W_X \times W_Y$ and $\langle N_X, N_Y \rangle \rightarrow f(\langle N_1, N'_1 \rangle, \ldots \langle N_n, N'_n \rangle) \in \Delta_\cap$ iff $N_X \rightarrow f(N_1, \ldots, N_n) \in \Delta_X$ and $N_Y \rightarrow f(N'_1, \ldots, N'_n) \in \Delta_Y$.

The type of terms satisfying a constraint $X = f(Y_1, \ldots, Y_n)$ is computed in two steps. First, the types of $Y_1 \ldots Y_n$ are combined into a new grammar $\mathcal{G}_Y = \langle S_f, W_Y, \Sigma, \Delta_Y \rangle$, defining the type of $f(Y_1, \ldots, Y_n)$, s.t. S_f is a new starting symbol, $W_Y = \{S_f\} \cup W_{Y_1} \cup \cdots \cup W_{Y_n}$ and $\Delta_Y = \{S_f \rightarrow f(S_{Y_1}, \ldots, S_{Y_n})\} \cup \Delta_{Y_1} \cup \cdots \cup \Delta_{Y_n}$. Then the intersection of (languages of) $\mathcal{G}_X$ and $\mathcal{G}_Y$ is computed as shown above.

It follows that the type of terms satisfying a program constraint corresponds to an intersection of (languages of) the grammars of two sides of the constraints. Each state of the grammar for this type is in the form $\langle A, B \rangle$, where A and B are non-terminals of the left-hand side and right-hand side grammars respectively.

Labelling The concept of labelling of non-terminals introduced in this paper is a basis for reasoning about the properties of possibly non-ground objects with respect to their types. The idea of labelling consists in associating the non-terminals of type grammars with sets of variables. In general, the need for augmenting the type information with some additional attributes naturally arises in the context of any analysis based on regular types. It is motivated by the insufficient expressive power of the underlying type mechanism. For instance, dependencies such as "two subterms are identical" or "two variables can share" are non-regular and thus, cannot be expressed as regular types.

Definition 5 (Labelling). *A variable V of a non-ground term t labels a non-terminal N of a grammar $\mathcal{G}$ if there exists a path in the run of $\mathcal{G}$ on t such that $S(t) \xrightarrow{*}_{\mathcal{G}} N(V)$, where S is a starting symbol of $\mathcal{G}$.*

In the following we assume a function $\zeta(\mathcal{G}, N, t)$ returning the set of labels of a non-terminal N of the grammar $\mathcal{G}$ on its run on a term t. By our assumption different grammars have different sets of non-terminal symbols and thus, it is always unambiguous to use the more compact notation $\zeta(N, t)$.

Example 4. Consider the labelling imposed by the term $t = [A, B|D]$ on the non-terminals for the regular tree grammar accepting lists: $L_X \rightarrow \text{nil} \mid \text{cons}(E_X, L_X)$. We can see that $\zeta(L_X, t) = \{D\}$ since $L_X(t) \xrightarrow{*} L_X(D)$ and $\zeta(E_X, t) = \{A, B\}$ since $L_X(t) \xrightarrow{*} E_X(A)$ and $L_X(t) \xrightarrow{*} E_X(B)$.

The concept of labelling of states (non-terminals) is a basis for the analyses shown below in this paper. Labelling allows us to associate different variables of a term with different non-terminals of the corresponding regular tree grammar. Thus, we can analyze the presence (or absence) of variables, groundness dependencies, sharing of variables, etc. on the *per-role* basis, computing more precise results than traditional untyped analyses usually operating on the *per-term* basis. For instance, this allows us to analyze groundness of "append" separately for list structure and list elements.

Labelling in Constraint Types Let's now apply the notion of labelling to the results about types of terms satisfying unification constraints. Assume a grammar $\mathcal{G}_\cap$ defining the type of terms satisfying a unification constraint C. Recall that all non-terminals of $\mathcal{G}_\cap$ can be viewed as pairs $\langle A, B\rangle$ where A and B are non-terminals of the left-hand side grammar $\mathcal{G}_X$ and right-hand side grammar $\mathcal{G}_Y$ respectively. Assume that a term t satisfying C labels some non-terminal A of $\mathcal{G}_X$ by a set of variables S on the run of $\mathcal{G}_X$ on t, i.e. $\zeta(\mathcal{G}_X, A, t) = S$. Now consider a run of $\mathcal{G}_\cap$ on t. Since $\mathcal{G}_\cap$ defines the intersection of (languages of) $\mathcal{G}_X$ and $\mathcal{G}_Y$, the elements of S can label only the non-terminals $\langle A, B\rangle$ of $\mathcal{G}_\cap$, i.e. the pairs with A in the first position. Thus, in a run of $\mathcal{G}_Y$ on t, the labels of S are divided between all such non-terminals B for which there exists a reachable non-terminal $\langle A, B\rangle$ of $\mathcal{G}_\cap$. Naturally, the same arguments can be applied in the opposite direction, i.e. by fixing a non-terminal $B \in W_Y$ and reasoning about its labels. Thus, for a term $t \in \mathcal{L}(\mathcal{G}_\cap)$ we have:

$$\forall A \in W_X .\ \zeta(A, t) \subseteq \textstyle\bigcup_{\langle A,B\rangle \in W_\cap} \zeta(B, t) \tag{1}$$

$$\forall B \in W_Y .\ \zeta(B, t) \subseteq \textstyle\bigcup_{\langle A,B\rangle \in W_\cap} \zeta(A, t). \tag{2}$$

Partitions of Non-Terminals As we can see, the sharing of labels by non-terminals of grammars $\mathcal{G}_X$ and $\mathcal{G}_Y$ is merely determined by the presence of particular pairs $\langle A, B\rangle$ in the set of reachable states of the intersection—the grammar $\mathcal{G}_\cap$. The two non-terminals $X \in W_X$ and $Y \in W_Y$ can have some labels in common if they can be related by some chain of states of $\mathcal{G}_\cap$, e.g. $\langle X, W\rangle \ldots \langle Z, W\rangle \ldots \langle Z, Y\rangle$. The more important fact is that a most general solution of the unification constraint cannot introduce sharing of labels between non-terminals that cannot be related through a chain of states of $\mathcal{G}_\cap$. In the following we formalize these two claims as theorems.

The notion of related states is formalized as the equivalence relation partitioning the state space W of $\mathcal{G}_\cap$ into the (disjoint) classes of states $W_\cap^{\mathcal{P}} = \{W_\cap^1, \ldots W_\cap^k\}$. The partitioning is induced by the following relation:

$$\forall\langle A, B\rangle \in W_\cap^i\ \forall\langle A', B'\rangle \in W_\cap^j.\ i \neq j \ \Rightarrow\ A \neq A' \wedge B \neq B'. \tag{3}$$

We say that two non-terminals $X, Y \in (W_X \cup W_Y)$ are *dependent* if the states of $\mathcal{G}_\cap$ including X and the states of $\mathcal{G}_\cap$ including Y appear in the same partition of $W_\cap$. We say that the non-terminals are *independent* otherwise.

Theorem 1. *Assume a unification constraint C, the grammars $\mathcal{G}_X$ and $\mathcal{G}_Y$ defining respectively the types of the left-hand side and the right-hand side of C, the grammar $\mathcal{G}_\cap$ such that $\mathcal{L}(\mathcal{G}_\cap) = \mathcal{L}(\mathcal{G}_X) \cap \mathcal{L}(\mathcal{G}_Y)$ and the partitioning $W_\cap^{\mathcal{P}} = \{W_\cap^1, \ldots, W_\cap^k\}$ of states of $\mathcal{G}_\cap$. Then for any $t \in \mathcal{L}(\mathcal{G}_\cap)$ and $i \in \{1 \ldots k\}$:*

$$\bigcup_{\langle X,Y\rangle \in W_\cap^i} \zeta(X, t) \ = \bigcup_{\langle X,Y\rangle \in W_\cap^i} \zeta(Y, t)$$

Proof. In two directions using Equations (1) and (2).

Theorem 1 provides us with the basic relation for the dependency between labelling sets of classes of non-terminals. This relation is based entirely on types of variables participating in the corresponding constraints and thus, can be computed statically. Now let us investigate the other side, i.e. the statically-known facts about independence of (labelling of) non-terminals.

Assume without loss of generality a constraint $C : X = f(Y_1, \ldots, Y_n)$, the grammars $\mathcal{G}_X$ and $\mathcal{G}_Y$ defining respectively the types of the left-hand side and the right-hand side of C and the grammar $\mathcal{G}_\cap$ defining the intersection of (languages of) $\mathcal{G}_X$ and $\mathcal{G}_Y$. Assume that the constraint C is solved in the environment $\psi = \{X/t_X, Y_1/t_1, \ldots Y_n/t_n\}$. Thus, solving C consists in finding the most general unifier of two terms: t_X and t_Y, where $t_Y = f(t_1, \ldots, t_n)$. Let $\theta = \mathrm{mgu}(t_X, t_Y)$. Now assume that A and B are two independent non-terminals of $\mathcal{G}_X$ and $\mathcal{G}_Y$ respectively. We claim that solving C cannot introduce any new sharing of labels of A and B. In other words, solving C cannot unify a variable from $\zeta(A, t_X)$ with a variable from $\zeta(B, t_Y)$.

Theorem 2. *For a unification problem $t_X \doteq t_Y$ with the types of t_X and t_Y given by the grammars $\mathcal{G}_X$ and $\mathcal{G}_Y$, the constraint solution $\theta = \mathrm{mgu}(t_X, t_Y)$ and two independent non-terminals A and B of $\mathcal{G}_X$ and $\mathcal{G}_Y$ respectively:*

$$\zeta(A, t_X) \cap \zeta(B, t_Y) = \emptyset \ \Rightarrow\ \zeta(A, t_X\theta) \cap \zeta(B, t_Y\theta) = \emptyset$$

Corollary 1. *Most general solutions of unification constraints never unify variables labelling independent non-terminals of type grammars.*

The results of Theorems 1 and 2 lead us to what we see as the main contribution of this work. Namely, the fact that by analyzing set constraints similar to the one in the claim of Theorem 1 we can obtain correct static approximations of properties such as groundness and sharing imposed by the corresponding unification constraint. Indeed, Theorem 1 approximates the ways in which labels (variables) can be unified when solving the corresponding unification constraint. Theorem 2 serves as a safety statement for Theorem 1 claiming that there are no other ways in which labels can be unified. Thus, correct descriptions of groundness and sharing dependencies can be derived using Theorem 1. Moreover, these descriptions are in general more precise than the results of traditional untyped analyses. This is because in our case the analysis is performed independently on the corresponding components of terms (i.e. per-role) and not on whole terms as in the traditional approach (e.g. [14]).

Note that in general the relations as shown in Theorem 1 may not include all non-terminals of the corresponding left-hand side and right-hand side grammars. Assume without loss of generality a non-terminal N of the left-hand side grammar $\mathcal{G}_X$. It is possible that the grammar defining the intersection does not contain any state in the form $\langle N, N' \rangle$. It means that no term satisfying the constraint can reach the state N on its run (derivation) in $\mathcal{G}_X$. Thus, $\zeta(N,t) = \emptyset$ for any t. We want to preserve this information in the abstract domains shown in the following sections. Thus, assuming the sets W_X, W_Y and $W_\cap$ of non-terminals of the left-hand side, right-hand side and the intersection grammars respectively we define a set of unreachable non-terminals:

$$W_\emptyset \;=\; \begin{array}{l} \left\{\, A \in W_X \;\middle|\; \forall B \in W_Y .\ \langle A, B\rangle \notin W_\cap \,\right\} \;\cup \\ \left\{\, B \in W_Y \;\middle|\; \forall A \in W_X .\ \langle A, B\rangle \notin W_\cap \,\right\} . \end{array} \tag{4}$$

4 Abstraction of Typed Logic Programs

The analyses described in this paper are constructed using a first order language, similar to that of logic programs, called *set logic programs*. In the previous works [2, 3] set logic programs are shown to be a clean and powerful formalism for designing program analyses.

Syntactically, we assume a set of variables $\mathcal{V}$ and an underlying alphabet $\Sigma^\oplus = \{\oplus/2, \emptyset/0\}$ consisting of a single binary function symbol $\oplus$ which "glues" elements together and a single constant symbol $\emptyset$ to represent the empty set. Abstract terms, or *set expressions*, are elements of the term algebra $\mathcal{T}(\Sigma^\oplus, \mathcal{V})$ modulo an equality theory consisting of the following axioms:

$$\begin{array}{llll} (x \oplus y) \oplus z = x \oplus (y \oplus z) & \text{(associativity)} & x \oplus x = x & \text{(idempotence)} \\ x \oplus y = y \oplus x & \text{(commutativity)} & x \oplus \emptyset = x & \text{(unity)} \end{array} \tag{5}$$

This equality theory is sometimes referred to as ACI1 and the corresponding equivalence relation on terms denoted $\approx$. This notion of equivalence suggests that abstract terms can be viewed as flat sets of variables. For example, the terms $x_1 \oplus x_2 \oplus x_3$, $x_1 \oplus x_2 \oplus x_3 \oplus \emptyset$, and $x_1 \oplus x_2 \oplus x_3 \oplus x_2$ can each be viewed as representing the set $\{x_1, x_2, x_3\}$ of three variables.

In this work we introduce the abstraction of unification constraints of typed logic programs into set constraints (ACI1-unification constraints). Set constraints are the elements of the set $\mathcal{C}^{\oplus} = \mathcal{T}(\Sigma^{\oplus}, \mathcal{V}) \times \mathcal{T}(\Sigma^{\oplus}, \mathcal{V})$ and are in the form $x \approx y$, where x and y are set expressions. The constraints are derived from the result of Theorem 1 as follows. Assuming a most general instance t satisfying the unification constraint we abstract each $\zeta(N, t)$ to an abstract variable also named N. This variable denotes the set of possible labels of the non-terminal N. We also let "$\oplus$" replace "$\cup$" in abstract constraints. This is done mainly because our abstract operations deal with abstract constraints literally and do not attach to the binary combinator "$\oplus$" any other meaning than prescribed by ACI1-axioms of Equation (5). The resulting set constraints reflect the relations between sets of concrete variables corresponding to dependent non-terminals of the type grammars, i.e. classes of variables which can interfere during the program execution.

Abstraction of Program Constraints Function σ_1, defined below, maps constraints of the form $X = Y$ to corresponding set constraints. We assume we are given the partition $W_{\cap}^{\mathcal{P}} = \{W_{\cap}^{1}, \dots W_{\cap}^{k}\}$ of non-terminals of the grammar $\mathcal{G}_{\cap}$ defining the type of terms satisfying the constraint and the set of unreachable non-terminals $W_{\emptyset}$ as defined by Equation (4).

$$\sigma_1(C) \; = \; \left\{ \bigoplus_{\langle A,B\rangle \in W_{\cap}^{i}} A \approx \bigoplus_{\langle A,B\rangle \in W_{\cap}^{i}} B \;\middle|\; i = 1 \dots k \right\} \;\cup\; \{\, N \approx \emptyset \mid N \in W_{\emptyset} \,\} \qquad (6)$$

Example 5. Consider the abstraction of the constraint $B = C$ from the first rule of concrete "append" (Figure 1). Assume that B and C are lists with the types defined as $L_A \to nil \mid cons(E_A, L_A)$ and $L_B \to nil \mid cons(E_B, L_B)$. Here we assume that E_A and E_B derive all possible terms thus, effectively providing the "any" type of elements. The grammar defining the intersection is: $\langle L_B, L_C\rangle \to \mathrm{nil} \mid \mathrm{cons}(\langle E_B, E_C\rangle, \langle L_B, L_C\rangle)$. Each of $\langle L_B, L_C\rangle$ and $\langle E_B, E_C\rangle$ forms an independent partition of the state space of this grammar. Thus, application of σ_1 of Equation (6) to the constraint produces: $\{L_B \approx L_C, \;\; E_B \approx E_C\}$. Note that the set $W_{\emptyset}$ is empty in this example.

The abstraction of constraints in the form $X = f(Y_1, \dots, Y_n)$ is almost the same as for $X = Y$. In addition we take into account the fact that only bound terms can satisfy the constraint. Thus, for any term t satisfying the constraint we have $\zeta(S_f, t) = \emptyset$, where S_f is a starting symbols of the right-hand side grammar. The function of abstraction σ_2 is defined for this case as:

$$\sigma_2(C) = \sigma_1(C) \;\cup\; \{S_f \;\approx\; \emptyset\} \qquad (7)$$

Example 6. Consider the abstraction of the constraint $A = [X|D]$ from the second rule of concrete "append" (Figure 1). Assume that A and D are lists defined respectively by the grammars $L_A \to \mathrm{nil} \mid \mathrm{cons}(E_A, L_A)$ and $L_D \to \mathrm{nil} \mid \mathrm{cons}(E_D, L_D)$. The intersection of the grammar for the right-hand side of the constraint with the grammar for A results in:

$$\begin{array}{lll} \mathcal{G}_{\cap} : \langle L_A, S_{\mathrm{cons}}\rangle & \to & \mathrm{cons}(\langle E_A, S_X\rangle, \langle L_A, L_D\rangle) \\ \quad\;\; \langle L_A, L_D\rangle & \to & \mathrm{nil} \mid \mathrm{cons}(\langle E_A, E_D\rangle, \langle L_A, L_D\rangle), \end{array}$$

where S_{cons} is a starting non-terminal of the grammar for the right-hand side. The partition of states of $\mathcal{G}_\cap$ is $\{\{\langle L_A, S_{\text{cons}}\rangle, \langle L_A, L_D\rangle\}, \{\langle E_A, S_X\rangle, \langle E_A, E_D\rangle\}\}$. Thus, $\sigma_2(A = [X|D]) = \{L_A \oplus L_A \approx S_{\text{cons}} \oplus L_D,\ E_A \oplus E_A \approx S_X \oplus E_D,\ S_{\text{cons}} \approx \emptyset\}$, which is simplified to $\{L_A \approx L_D,\ E_A \approx S_X \oplus E_D\}$ by substituting $\emptyset$ for S_{cons} and applying the idempotence and unity axioms.

Finally, the abstraction of unification constraints is formalized as a function σ based on the two cases of Equations (6) and (7):

$$\begin{aligned} &\sigma : \mathcal{C} \to \wp(\mathcal{C}^{\oplus}) \\ &\sigma(C) \;=\; \begin{cases} \sigma_1(C) & \text{if } C \text{ is in the form } X = Y \\ \sigma_2(C) & \text{if } C \text{ is in the form } X = f(Y_1, \ldots, Y_n) \end{cases} \end{aligned} \tag{8}$$

Note that $\sigma(C)$ constrains all non-terminals of the corresponding left-hand and right-hand side grammars of C. This is because each non-terminal of these grammars appears either as "reachable", i.e. as a part of some combined non-terminal of the intersection, or as "unreachable", i.e. as a member of $W_\emptyset$.

Program Abstraction Our goal is to obtain refined view of properties of program variables by considering the interaction of the corresponding components (non-terminals) of their associated types. Thus, the abstraction of concrete literals naturally replaces each variable with the corresponding tuple of non-terminals. Let function $\omega(\mathcal{G})$ return the non-terminals of $\mathcal{G}$ in an ordered tuple. The abstraction of literals is then defined as:

$$\begin{aligned} &\delta : \mathcal{B} \to \mathcal{B}^{\oplus} \\ &\delta(p(X_1, \ldots, X_n)) = p(\omega(\mathcal{G}_{X_1}), \ldots, \omega(\mathcal{G}_{X_n})). \end{aligned} \tag{9}$$

The concept of abstraction of literals and unification constraints is naturally lifted for abstracting programs. A *set logic program* is obtained by applying δ to all literals and σ to all unification constraints. The example of abstraction of "append" into a set logic program is shown in Figure 3.

In Section 3 we assume that all rule heads and calls corresponding to a predicate name $p/n \in \Pi$ have the same type. Thus, all abstract literals corresponding to p/n have the same form. This is important for evaluating semantics of abstract program since the corresponding parameters, i.e. tuples of non-terminals can be unified directly on the syntactic basis.

The Abstract Domain of Set Logic Programs The abstract domain is formalized as a lattice $\text{Type}(\mathsf{X}) = \langle \wp(\mathcal{B}^{\oplus}_{\mathcal{C}}), \subseteq \rangle$, where $\mathcal{B}^{\oplus}_{\mathcal{C}}$ is a set of *constrained abstract literals* in the form: $\pi \;\leftarrow\; \varphi_1 \wedge \cdots \wedge \varphi_n$. The mapping from concrete literals to abstract literals is provided by $\alpha^{\oplus}$:

$$\begin{aligned} &\alpha^{\oplus} : \mathcal{B}_{\mathcal{C}} \to \mathcal{B}^{\oplus}_{\mathcal{C}} \\ &\alpha^{\oplus}([p(X_1 \ldots, X_n) \leftarrow C_1 \wedge \cdots \wedge C_m]) = \\ &\qquad = \; p(\omega(\mathcal{G}_{X_1}), \ldots, \omega(\mathcal{G}_{X_n})) \leftarrow \left[\bigwedge_{\varphi \in \sigma(C_1)} \varphi \wedge \cdots \wedge \bigwedge_{\varphi \in \sigma(C_m)} \varphi \right] \end{aligned} \tag{10}$$

```
append(A,B,C) :-          append(⟨L_A, E_A⟩, ⟨L_B, E_B⟩, ⟨L_C, E_C⟩) :-
    A = [],                   L_A ≈ ∅,  E_A ≈ ∅,
    B = C.                    L_B ≈ L_C,  E_B ≈ E_C.

append(A,B,C) :-          append(⟨L_A, E_A⟩, ⟨L_B, E_B⟩, ⟨L_C, E_C⟩) :-
    A = [X|D],                L_A ≈ L_D,  E_A ≈ E_X ⊕ E_D,
    C = [X|E],                L_C ≈ L_E,  E_C ≈ E_X ⊕ E_E,
    append(D,B,E).            append(⟨L_D, E_D⟩, ⟨L_B, E_B⟩, ⟨L_E, E_E⟩).
```

Fig. 3. Abstraction of `append` to set logic program

Finally the Galois insertion between the $\wp(\mathcal{B}_\mathcal{C})$ and $\wp(\mathcal{B}_\mathcal{C}^\oplus)$ is provided by the following pair of *abstraction* and *concretization* functions:

$$\begin{array}{ll} \alpha : \wp(\mathcal{B}_\mathcal{C}) \to \wp(\mathcal{B}_\mathcal{C}^\oplus) & \gamma : \wp(\mathcal{B}_\mathcal{C}^\oplus) \to \wp(\mathcal{B}_\mathcal{C}) \\ \alpha(I) = \{\ \alpha^\oplus(a) \,\big|\, a \in I\ \} & \gamma(\mathcal{I}) = \bigcup \{\ I \,\big|\, \alpha(I) \subseteq \mathcal{I}\ \} \end{array} \tag{11}$$

5 Groundness Analysis of Typed Logic Programs

Groundness is probably the best-understood and most developed application of abstract analysis of logic programs. Most of the known optimizations found in compilers of (constraint) logic programming languages require some approximation of groundness to operate.

The best-known method of groundness analysis of logic programs consists in approximating groundness dependencies using the domain of *positive boolean formulas* called Pos. This method is proposed in [14]. Without loss of generality the abstraction of unification constraint $t_1 = t_2$ is defined by the following function α_{Pos}:

$$\begin{array}{l} \alpha_{\mathsf{Pos}} : \mathcal{C} \to \mathsf{Pos} \\ \alpha_{\mathsf{Pos}}([t_1 = t_2]) = \left[\bigwedge_{V \in \mathrm{vars}(t_1)} V \;\leftrightarrow\; \bigwedge_{W \in \mathrm{vars}(t_2)} W \right] \end{array} \tag{12}$$

Thus, the abstraction of ground t_1 or ground t_2 results in T (true) on the corresponding side of the formula. The notion of Pos-based groundness abstraction is naturally lifted for all syntactic objects such as constrained literals, rules and programs. The semantics of a CLP(B) program resulting from program abstraction provides a correct approximation of groundness dependencies of the original program. The abstract domain for groundness analysis is thus formalized in terms of (sets of) Pos-constrained literals ordered by logical consequence.

Example 7. The abstraction of "append" for Pos-based groundness analysis.

$\mathtt{append}(A, B, C)$ `:-` $A \leftrightarrow \mathsf{T}$, $B \leftrightarrow C$.
$\mathtt{append}(A, B, C)$ `:-` $A \leftrightarrow X \wedge D$, $C \leftrightarrow X \wedge E$, $\mathtt{append}(D, B, E)$.

The bottom-up evaluation of this program results in the well-known groundness approximation: $\mathtt{append}(A, B, C)$`:-` $C \leftrightarrow A \wedge B$. Note that this result is less precise than the result for typed "append" shown in Section 1.

It's clear that given type descriptions we can adopt a more refined view of groundness than in the untyped case. Namely, groundness of a term can be expressed in terms of *boundness* of all non-terminals of the corresponding grammar. If we can prove, for instance, that for some non-terminal N and for any t satisfying the constraint we have $\zeta(N,t) = \emptyset$, it means that the subterm corresponding to N is always instantiated. Consequently proving that $\zeta(N,t) = \emptyset$ for any N means that all terms satisfying the constraint are ground. The following two definitions formalize the concepts of boundness and groundness:

Definition 6 (Boundness). *Non-terminal N of the grammar $\mathcal{G}_V$ associated with a variable V is bound by a substitution θ if $\zeta(N, V\theta) = \emptyset$.*

Definition 7 (Groundness). *A variable V is ground under a substitution θ if all non-terminals of the associated grammar $\mathcal{G}_V$ are bound by θ.*

The domain for groundness analysis of typed logic programs is called Type(Pos). It is a proper abstraction of the generic domain Type(X) presented in Section 4 obtained by application of α_{Pos} of Equation (12) to ACI1-constraints of a set logic program. As an example, the application of α_{Pos} to the abstract "append" shown in Figure 3 results in the program shown earlier in Figure 1.

The semantics of the resulting abstract CLP(B) program is then evaluated either in a top-down or a bottom-up way. Thus, the approximation of boundness of a program is obtained by its abstraction into a set logic program followed by direct application of Pos-based groundness analysis.

Clearly, the analysis in Type(Pos) is at least as precise as the analysis in Pos. We formalize this relation showing that given a description in Type(Pos) we can always "abstract away" the types and get a description in Pos domain. In other words, we are showing that Pos is a proper abstraction of Type(Pos). Intuitively this claim is supported by the fact that groundness of a program variable V can be expressed as Pos-formula $V \leftrightarrow \bigwedge_{N \in W_V} N$, where W_V is the set of non-terminals of the type grammar $\mathcal{G}_V$.

Theorem 3. Pos *is a proper abstraction of* Type(Pos).

Example 8. The groundness description of typed "append" shown in Section 1 can be mapped onto the original program variables as

$$\begin{array}{ll}\texttt{append}(A,B,C) \texttt{ :- } & L_A \leftrightarrow \mathsf{T},\ L_B \leftrightarrow L_C,\ E_C \leftrightarrow E_A \wedge E_B,\\ & A \leftrightarrow L_A \wedge E_A,\ B \leftrightarrow L_B \wedge E_B,\ C \leftrightarrow L_C \wedge E_C.\end{array}$$

This description implies the standard result for untyped "append".

6 Sharing Analysis of Typed Logic Programs

Two or more variables in a logic program are said to *share* if in some execution of the program they are bound to terms which contain a common variable. The information about sharing of program variables provides the basis for many optimizations, such as occur check reduction [20] or elimination of run-time dataflow checks in and-parallel execution [9].

One of the more widely applied sharing analyses reported in the literature is the so called *set-sharing* analysis based on Sharing domain originally proposed by Jacobs and Langen [11]. The alternative domain for set-sharing analysis is recently presented in [3]. This domain denoted by $\mathsf{Sh}^{\oplus}$ is isomorphic to Sharing and based on set expressions. Sharing descriptions are obtained by abstracting original programs into set logic programs and evaluating their meaning using standard top-down or bottom-up semantics extended with ACI1-unification.

In this work we show how both both Sharing and $\mathsf{Sh}^{\oplus}$ can be upgraded with type information of Type(X) domain resulting respectively in Type(Sharing) and Type($\mathsf{Sh}^{\oplus}$). It turns out similarly to the case of groundness analysis that the abstract operations of Sharing and $\mathsf{Sh}^{\oplus}$ can be applied to set logic programs resulting from Type(X) abstraction. Moreover, sharing analysis based on $\mathsf{Sh}^{\oplus}$ can be applied directly to a result of Type(X) abstraction because of the same syntax used in both domains.

Similarly as groundness can be viewed as a combination of the corresponding boundness dependencies, set-sharing between program variables can be viewed as a combination of *aliasing* of labels of corresponding non-terminals. The following two definitions formalize the concepts of aliasing and sharing:

Definition 8 (Aliasing). *Non-terminals $N_1 \ldots N_n$ of the respective grammars $\mathcal{G}_1 \ldots \mathcal{G}_n$ associated with variables $V_1 \ldots V_n$ are aliased by a substitution θ if $\zeta(N_1, V_1\theta) \cap \cdots \cap \zeta(N_n, V_n\theta) \neq \emptyset$.*

Definition 9 (Sharing). *Program variables $V_1 \ldots V_n$ share under a substitution θ if there is a set of non-terminals $N_1 \ldots N_n$ of the respective grammars $\mathcal{G}_1 \ldots \mathcal{G}_n$ aliased by θ.*

Example 9. Consider the bottom-up sharing analysis of abstract "append" shown in Figure 3 using $\mathsf{Sh}^{\oplus}$. The result is:

$$\mathtt{append}(\langle L_A, E_A\rangle, \langle L_B, E_B\rangle, \langle L_C, E_C\rangle) \ \mathtt{:-}\ L_A \approx \emptyset,\ L_B \approx L_C,\ E_C \approx E_A \oplus E_B.$$

showing that L_A is ground; L_B and L_C can share; E_C can share with each of E_A and E_B, but E_A and E_B are independent. The analysis based on Sharing domain can also be applied producing the same (upto isomorphism) result. The abstraction in Sharing treats terms as sets of variables and thus can be applied to set constraints. The result of the analysis in Sharing is naturally $\psi = \{\emptyset, \{L_B, L_C\}, \{E_C, E_A\}, \{E_C, E_B\}\}$.

Note that the information computed in Example 9 approximates possible aliasing between the labels of the corresponding non-terminals. The information about actual sharing of program variables can be extracted by projecting the aliasing back on the type description. For instance, variables A and C of "append" can share because the elements of the lists can be aliased. Variables B and C can share because of two possible reasons: the structures can be aliased, provided by the aliasing of L_B and L_C or the elements can be aliased, provided by the aliasing of E_B and E_C. In general, if we adopt the syntax of $\mathsf{Sh}^{\oplus}$, the information about sharing of the original variables can be captured by introducing new set constraints in the form $V \approx N_1 \oplus \cdots \oplus N_k$, where V is a program variable and $N_1 \ldots N_k$ are non-terminals of the corresponding grammar $\mathcal{G}_V$. Thus, similarly to the case of groundness analysis we can show that the analysis in $\mathrm{Type}(\mathsf{Sh}^\oplus)$ is at least as precise as in $\mathsf{Sh}^\oplus$. The similar result for $\mathsf{Sharing}$ follows by the isomorphism.

Theorem 4. $\mathsf{Sh}^\oplus$ *is a proper abstraction of* $\mathrm{Type}(\mathsf{Sh}^\oplus)$.

7 Related Work and Conclusion

We briefly compare the approach proposed in this paper with some previous works presenting abstract domains based on regular types.

Janssens and Bruynooghe present in [12] a framework for program type and mode analysis of Prolog programs based on type graphs. The abstract domains based on *rigid types* and *integrated types* are capable of inferring types and modes of a program. The later domain can track *freeness* as an additional mode, thus significantly improving the precision of analysis for programs manipulating partially instantiated structures. The work of Mulkers *et al.* [16] applies the framework of [12] for tracking *liveness* of Prolog structures. The abstract domain developed in this work models the execution environment of a predicate using type graphs and special arcs representing structure sharing. In Bruynooghe *et al.* [1] the framework of [16] is further extended and applied to liveness analysis of statically typed languages such as Mercury.

The work of Van Hentenryck *et al.* [8] presents the systematic approach to the construction of abstract domains using the generic domain $\mathrm{Pat}(\mathcal{R})$. In particular, authors describe how the domain of type graphs Type can be integrated into $\mathrm{Pat}(\mathcal{R})$ producing $\mathrm{Pat}(\mathsf{Type})$. The resulting domain provides type and mode inference similarly to the domain of Janssens and Bruynooghe [12].

The main difference of our approach from the mentioned works is in the way we deal with regular types. In all cited works the abstract representation includes regular types as a component. Abstract operations presented in these works normally involve complex and costly manipulations with type graphs during the analysis. In contrast, our approach fully handles the type descriptions on the stage of program abstraction, i.e. by type-based abstract compilation into set logic programs. The most similar approach in a technical sense is that proposed by Naish [18]. In this work the problem of modes validation of a typed and moded logic program is reduced to checking the entailment on multiset constraints composed from parameters of polymorphic types.

We have presented an approach to analysing typed logic programs by mapping to a set logic program. It is a straightforward syntax-directed translation which can increase the size of the program by a factor d^2, where d is the largest number of nonterminals in a grammar, although in practice it is usually linear.

A practical advantage of the approach is that the type-based analyzer does not need to manipulate automata or graphs during the analysis. Instead, the actual analysis which comes after the type-based abstraction can be delegated to the existing analyzers thus reusing their expertise together with their code and optimizations.

An obvious direction for the future research is to extend our approach to directly handle parametric polymorphism in types. Another direction for the research consists in formalizing and extending the class of abstract domains that can be upgraded with type information using our method. Here we show that two important abstract domains in logic program analysis, Pos and Sharing, fall into this class.

References

1. M. Bruynooghe, G. Janssens, and A. Kågedal. Live-structure analysis for logic programming languages with declarations. In *Procs. 14th ICLP*, 33–47. MIT Press, 1997.
2. M. Codish and V. Lagoon. Type dependencies for logic programs using ACI-unification. *Theoretical Computer Science*, 238(1-2):131–159, 2000.
3. M. Codish, V. Lagoon, and F. Bueno. An algebraic approach to sharing analysis of logic programs. *JLP*, 42(2):111–149, 2000.
4. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM Symp. on Principles of Programming Languages*, 238–252, 1977.
5. B. Demoen, M. Garcia de la Banda, W. Harvey, K. Marriott, and P. Stuckey. An overview of HAL. In *Procs. CP99. LNCS* 1713, 174–188, 1999.
6. J. Gallagher and D. Waal. Fast and precise regular approximations of logic programs. In *Procs. 11th ICLP*, 599–613. MIT Press, 1994.
7. F. Gécseg and M. Steinby. *Tree Automata*. Akadémiai Kiadó, Budapest, 1984.
8. P. Van Hentenryck, A. Cortesi, and B. Le Charlier. Type analysis of Prolog using type graphs. *JLP*, 22(3):179–209, 1995.
9. M. V. Hermenegildo and K. J. Greene. &-Prolog and its performance: Exploiting independent And-Parallelism. In *Procs. 7th ICLP*, 253–268. MIT Press, 1990.
10. P. Hill and J. Lloyd. *The Gödel Language*. MIT Press, 1994.
11. D. Jacobs and A. Langen. Static analysis of logic programs for independent AND-parallelism. *JLP*, 13(2 & 3):291–314, 1992.
12. G. Janssens and M. Bruynooghe. Deriving descriptions of possible values of program variables by means of abstract interpretation. *JLP*, 13(2 & 3):205–258, 1992.
13. J. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 2nd edition, 1988.
14. K. Marriott and H. Søndergaard. Precise and efficient groundness analysis for logic programs. *ACM TOPLAS*, 2(4):181–196, 1993.
15. R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.
16. A. Mulkers, W. Winsborough, and M. Bruynooghe. Live-structure dataflow analysis for Prolog. *ACM TOPLAS*, 16(2):205–258, 1994.
17. A. Mycroft and R.A. O'Keefe. A polymorphic type system for Prolog. *Artificial Intelligence*, 23:295–307, 1984.
18. L. Naish. Mode checking using constrained regular trees. Technical Report 98/3, Department of Computer Science, University of Melbourne, Australia, 1998.
19. Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *JLP*, 29(1–3):17–64, 1996.
20. H. Søndergaard. An application of abstract interpretation of logic programs: Occur check reduction. In *Procs. ESOP86*, *LNCS* 213, 327–338, 1986.