Abstract interpretation over non-deterministic finite tree automate for set-based analysis of logic programs

Gallagher, John Patrick; Puebla, G.

Published in:
Practical aspects of declarative languages, 4th International Symposium, PADL 2002

Publication date:
2002

Document Version
Publisher's PDF, also known as Version of record

Citation for published version (APA):
Gallagher, J. P., & Puebla, G. (2002). Abstract interpretation over non-deterministic finite tree automate for set-based analysis of logic programs. In S. Krishnamurthi, & C. R. Ramakrishnan (Eds.), *Practical aspects of declarative languages, 4th International Symposium, PADL 2002: Portland, OR, USA* (pp. 243-261). Springer. Lecture Notes in Computer Science Vol. 2257

Abstract Interpretation over Non-Deterministic Finite Tree Automata for Set-Based Analysis of Logic Programs

John P. Gallagher[1] and Germán Puebla[2]

[1] University of Bristol, Dept. of Computer Science, BS8 1UB Bristol, UK
E-mail: john@cs.bris.ac.uk
[2] Universidad Politécnica de Madrid, Facultad de Informática,
E-28660 Boadilla del Monte, Madrid
E-mail: german@fi.upm.es

Abstract. Set-based program analysis has many potential applications, including compiler optimisations, type-checking, debugging, verification and planning. One method of set-based analysis is to solve a set of *set constraints* derived directly from the program text. Another approach is based on abstract interpretation (with widening) over an infinite-height domain of regular types. Up till now only deterministic types have been used in abstract interpretations, whereas solving set constraints yields non-deterministic types, which are more precise. It was pointed out by Cousot and Cousot that set constraint analysis of a particular program P could be understood as an abstract interpretation over a finite domain of regular tree grammars, constructed from P. In this paper we define such an abstract interpretation for logic programs, formulated over a domain of non-deterministic finite tree automata, and describe its implementation. Both goal-dependent and goal-independent analysis are considered. Variations on the abstract domains operations are introduced, and we discuss the associated tradeoffs of precision and complexity. The experimental results indicate that this approach is a practical way of achieving the precision of set-constraints in the abstract interpretation framework.

1 Introduction

Recursively defined sets of terms are familiar to us as approximations of the runtime values of program variables. For example, the expression *intlist* $::= [\ ]; [int|intlist]$ defines a set called *intlist* containing all lists of integers, where *int* denotes the set of integers. Such expressions are sometimes used by the programmer to restrict the values that an argument or variable is allowed to take, but in this paper we are concerned with deriving such descriptions statically, rather than prescribing them.

Derivation of set expressions such as these has many applications including type inference [16, 8], debugging [24], assisting compiler optimisations [25, 34], optimising a theorem prover [14], program specialisation [20], planning [4] and verification [8]. The first work in this area was by Reynolds [33]; other early research was done by Jones and Muchnick [27, 26]. In the past decade two different approaches to deriving set expressions have been followed. One approach is based on abstract interpretation [25, 34, 19, 13, 30], and the other on solving set constraints derived from the program text [22, 16, 21, 2, 1, 28, 9, 32]. In abstract interpretation the program is executed over an abstract *type domain*, program variables taking on abstract values represented by types rather than standard values. In set-constraint analysis, program variables are also interpreted as taking on sets of values, but a set of inclusion relations is derived from the program text and then solved.

Cousot and Cousot pointed out [13] that set constraint solving of a particular program P could be understood as an abstract interpretation over a finite domain of tree grammars, constructed from P. Set constraint analysis can be seen as one of a range of related "grammar-based" analyses. One practical advantage of seeing set constraint solving as abstract interpretation (noted by Cousot and Cousot) is that set-constraint-based analysis can be combined with other analysis domains, using well established principles. A second advantage is that various tradeoffs of precision against efficiency can be exploited without departing from the abstract interpretation framework.

In this paper we pursue the idea of an abstract interpretation corresponding to set constraints in more depth. After reviewing the basic notions of non-deterministic finite tree automata in Section 2, we construct an abstract domain for a given logic program in Section 4. In Section 5 we construct abstract interpretations for logic programs over this domain. These include two variants that we call the variable-based and the argument-based interpretations. We also consider both goal-dependent and goal-independent interpretations. Our implementation is described in Section 6 and the results of experiments in Section 7. The results are discussed in Section 8.

2 Preliminaries

Let Σ be a set of ranked function symbols. We refer to elements of Σ as $f_j^{n_j}$ where $n_j \geq 0$ is the rank (arity) of function symbol (functor) f_j. If $n_j = 0$ we call f_j a *constant.* The set of *ground terms* (or *trees*) Term_Σ associated with Σ is the least set containing the constants and all expressions $f_j^{n_j}(t_1, \ldots, t_{n_j})$ such that $t_1, \ldots, t_{n_j}$ are elements of Term_Σ.

Finite tree automata provide a means of finitely describing possibly infinite sets of ground terms, just as finite automata describe sets of strings. A non-deterministic finite tree automaton (NFTA) is defined as a quadruple $\langle Q, q_0, \Sigma, \Delta \rangle$, where Q is a finite set of *states*, $q_0 \in Q$ is called the accepting state, Σ is a set of ranked function symbols and Δ is a set of *transitions.* Each element of Δ is of the form $f_j^{n_j}(q_1, \ldots, q_{n_j}) \rightarrow q$, where $f_j^{n_j} \in \Sigma$ and $q, q_1, \ldots, q_{n_j} \in Q$.

Let $R = \langle Q, q_0, \Sigma, \Delta \rangle$ be an NFTA; a *derivation* in R is a labelled tree τ such that each node of τ is labelled with a term from Term_Σ and a state from Q, satisfying the following condition. The state labelling the root node is q_0, and if any node p is labelled with term $f_j^{n_j}(t_1, \ldots, t_{n_j})$ and state q then there is a transition $f_j^{n_j}(q_1, \ldots, q_{n_j}) \rightarrow q \in \Delta$ and p has n_j children $p_1, \ldots, p_{n_j}$ labelled with terms $t_1, \ldots, t_{n_j}$ and states $q_1, \ldots, q_{n_j}$ respectively. In particular, if p is a leaf node, then p is labelled with a constant f_j^0 and some state q, and there is a transition $f_j^0 \rightarrow q$.

We say that a term t is *accepted* by automaton R if there is a derivation in R whose root node is labelled with t. The set of all terms accepted by automaton R is called the *(tree) language* of R, denoted $L(R)$. Two automata R_1, R_2 are *equivalent*, written $R_1 \cong R_2$, iff $L(R_1) = L(R_2)$. $\mathsf{empty}(R)$ is true iff $L(R)$ is empty, and $\mathsf{nonempty}(R)$ is the same as $\neg\mathsf{empty}(S)$. An automaton R_1 is contained in automaton R_2, written $R_1 \preceq R_2$ iff $L(R_1) \subseteq L(R_2)$.

An automaton with transitions Δ is called (top-down) deterministic if there are no two transitions in Δ with both the same right-hand-side q and the same function symbol $f_j^{n_j}$ on the left. Deterministic automata are less expressive than NFTAs in general, unlike finite automata for string languages. There are NFTAs for which there is no equivalent deterministic finite tree automaton.

Let $R_1 = \langle Q_1, q_1, \Sigma, \Delta_1 \rangle$ and $R_2 = \langle Q_2, q_2, \Sigma, \Delta_2 \rangle$ be NFTAs. The *product* automaton $R_1 \times R_2$ is defined as the automaton $\langle Q_1 \times Q_2, (q_1, q_2), \Sigma, \Delta_1 \times \Delta_2 \rangle$ where

$$\begin{aligned}\Delta_1 \times \Delta_2 = \{f_j^{n_j}((q_1, q_1'), \ldots, (q_{n_j}, q_{n_j}'))) \rightarrow (q, q') \mid \\ f_j^{n_j}(q_1, \ldots, q_{n_j}) \rightarrow q \in \Delta_1 \\ f_j^{n_j}(q_1', \ldots, q_{n_j}') \rightarrow q' \in \Delta_2\}\end{aligned}$$

The language accepted by $R_1 \times R_2$ is $L(R_1) \cap L(R_2)$.

NFTAs can be extended to allow ϵ-transitions, without altering their expressive power. An ϵ-transition is of the form $q \rightarrow q'$. Such transitions can be removed from Δ, after adding all transitions $f_j^{n_j}(q_1, \ldots, q_{n_j}) \rightarrow q'$ such that there is a transition $f_j^{n_j}(q_1, \ldots, q_{n_j}) \rightarrow q$ in Δ, and q' is reachable from q using only ϵ-transitions. Given a set of transitions Δ containing ϵ-transitions, the result of eliminating them will be called $\mathsf{elim}_\epsilon(\Delta)$.

NFTAs are quite expressive, as we will see from examples, yet key properties are decidable. It is decidable whether an automaton is empty, and whether a given term is accepted by an automaton. Containment, and hence equivalence, is also decidable.

We will use the following shorthand notation. If we name an automaton R_{q_0} then q_0 is its accepting state. If two automata R_{q_1} and R_{q_2} appear in the same context, we mean that they differ only in their accepting state.

If R_q contains two transitions $f_j^{n_j}(q_1, \ldots, q_{n_j}) \rightarrow q$ and $f_j^{n_j}(q_1', \ldots, q_{n_j}') \rightarrow q$, and $R_{q_k} \preceq R_{q_k'}$ for $1 \leq k \leq n_j$, then the transition $f_j^{n_j}(q_1, \ldots, q_{n_j}) \rightarrow q$ is *redundant.* Clearly we can remove redundant transitions from an automaton without altering its language.

As we will be applying NFTAs in the context of logic programming, it will be convenient to adopt the notation of *regular unary logic (RUL) programs* to describe NFTAs. An RUL clause is a formula of the form $q(f(x_1, \ldots, x_n)) \leftarrow q_1(x_1), \ldots, q_n(x_n)$ where $x_1, \ldots, x_n$ are distinct variables. An NFTA $\langle Q, q_0, \Sigma, \Delta \rangle$ can be translated to an RUL program where Q is a set of unary predicate symbols, and each transition $f_j^{n_j}(q_1, \ldots, q_{n_j}) \rightarrow q \in \Delta$ is represented as the RUL clause $q(f_j^{n_j}(x_1, \ldots, x_{n_j})) \leftarrow q_1(x_1), \ldots, q_{n_j}(x_{n_j})$. Thus in this representation, Δ is an RUL program. There is then a straightforward correspondence between derivations and acceptance in NFTAs and logic program computations. In particular, the term t is accepted by the automaton R_q iff $\Delta \cup \{\leftarrow q(t)\}$ has an SLD refutation, where Δ is the set of transitions of R.

Further details on NFTAs and their properties can be found elsewhere [12].

3 Core Semantics

In this section we develop bottom-up semantics for definite logic programs, parameterised by a domain of interpretation, and certain operations on that domain. Thus we follow the established method in abstract interpretation of providing *core semantics* that can be instantiated to yield either the standard (concrete) semantics, or some other abstract semantics.

We start from the familiar T_P operator associated with a definite program P. We write the definition of T_P as follows, introducing operators $\mathsf{project}$, reduce and $\bigsqcup$ that will be abstracted later on.

$$T_P(I) = \bigsqcup_P \{\mathsf{project}(H,\theta) \mid H \leftarrow B \in P, \quad \theta \in \mathsf{reduce}(B,I)\}$$

Let B_P be the Herbrand base of P, and $D_P = 2^{B_P}$. The concrete domain $(D_P, \subseteq, \emptyset, B_P)$ is a complete lattice. In the concrete semantics, $I \in D_P$, $\mathsf{reduce}((B_1,\ldots,B_k),I)$ is the set of all ground substitutions θ, whose domain is $\mathsf{vars}(B_1,\ldots,B_k)$ and range is the Herbrand universe of P, such that $\{B_1\theta,\ldots,B_k\theta\} \subseteq I$. $\mathsf{project}(H,\theta)$ is $H\theta$, and $\bigsqcup_P(S)$ set of ground instances (over the Herbrand universe of P) of elements of S.

This can easily be seen to be equivalent to the more familiar presentation of T_P [29], and we have the well known result that the least fixed point (lfp) of T_P (with respect to the partial order on D_P) is the least Herbrand model of P, $\mathsf{M}[P]$. The least fixed point is the limit of the sequence $\{T_P^n(\emptyset)\}$, $n = 0, 1, \ldots$.

In the following sections, we will develop abstract instances of the core semantics. We start by defining abstract domains, and then we define the abstract versions of reduce, project and $\bigsqcup$.

4 Abstract Domains of NFTAs

Let P be a definite logic program and $\mathsf{M}[P]$ its minimal Herbrand model. Consider the set of *occurrences* of subterms of the heads of clauses in P, including the heads themselves; call this set $\mathsf{headterms}(P)$. $\mathsf{headterms}(P)$ is the set of program points that we want to observe. We are interested in analysing the set of terms that can occur at each of these positions in instances of clauses satisfied by $\mathsf{M}[P]$.

A function S will be defined from $\mathsf{headterms}(P)$ to a set of identifiers. The states of an NFTA will be constructed from these identifiers in Section 4; in fact, an automaton state will correspond to a set of identifiers. For instance, we might assign an identifier, say q_X, to an occurrence of a variable X in some clause head. The set of terms accepted at state $\{q_X\}$ in the automaton that is produced (Section 5) will approximate the set of terms that could appear as instances of X at that position. There will be one or more transitions in the automaton of the form $f(Q_1,\ldots,Q_k) \rightarrow \{q_X\}$, where $Q_1,\ldots,Q_k$ are themselves sets of identifiers.

Thus if S maps two distinct elements of $\mathsf{headterms}(P)$ to the same state, then we will not be able to distinguish the sets of terms that occur at the two positions. We will consider two variants of the mapping, called S^P_{var}, the *variable-based* mapping, and S^P_{arg}, the *argument-based* mapping, which differ in the degree to which they distinguish different positions.

The S mapping is built from several components, representing the mappings of arguments, variables, and other terms that occur in the clause heads. Let $\mathsf{Q}, \mathsf{Args}$ and V be disjoint infinite sets of identifiers. The mapping id_P is chosen to be any injective mapping $\mathsf{headterms}(P) \rightarrow \mathsf{Q}$. The set of *argument positions* is the set of pairs $\langle p, j\rangle$ such that p is an n-ary predicate of the language and $1 \leq j \leq n$. The function argpos is some injective mapping from the set of argument positions to Args, that is, giving a unique identifier to each argument position. Let varid be an injective mapping from the set of variables of the language to V. Let type and any be distinguished identifiers not in $\mathsf{Q} \cup \mathsf{Args} \cup \mathsf{V}$.

We will assume for convenience that the clauses of programs have been *standardised apart*; that is, no variable occurs in more than one clause. The following definitions define two different mappings from clause head positions to states.

Definition 1. S^P_{var}

Let P be a definite program. The function $\mathsf{S}^P_{var} : \mathsf{headterms}(P) \rightarrow \mathsf{Q} \cup \mathsf{V} \cup \{\mathsf{type}\}$ is defined as follows.

$$
\begin{array}{rl}
\mathsf{S}^P_{var}(t) = & \textit{if } t \textit{ is a clause head, then } \mathsf{type} \\
& \textit{else if } t \textit{ is a variable, then } \mathsf{varid}(t) \\
& \textit{else } \mathsf{id}_P(t)
\end{array}
$$

Definition 2. S^P_{arg}

Let P be a definite program. The function $\mathsf{S}^P_{arg} : \mathsf{headterms}(P) \rightarrow \mathsf{Q} \cup \mathsf{Args} \cup \mathsf{V} \cup \{\mathsf{type}\}$ is defined as follows.

$$
\begin{array}{rl}
\mathsf{S}^P_{var}(t) = & \textit{if } t \textit{ is a clause head, then } \mathsf{type} \\
& \textit{else if } t \textit{ occurs as argument } j \textit{ of predicate } p, \textit{ then } \mathsf{argpos}(\langle p, j\rangle) \\
& \textit{else if } t \textit{ is a variable, then } \mathsf{varid}(t) \\
& \textit{else } \mathsf{id}_P(t)
\end{array}
$$

Example 1. Let P be the *append* program.

$$
\mathit{append}([\,], A, A) \leftarrow \mathit{true} \qquad \mathit{append}([B|C], D, [B|E]) \leftarrow \mathit{append}(C, D, E)
$$

Taking them in textual order $\mathsf{headterms}(P)$ is the following set. We can imagine the different occurrences of the same term (such as A) to be subscripted to indicate their positions, but we omit this extra notation.

$$
\{\mathit{append}([\,], A, A), [\,], A, A, \mathit{append}([B|C], D, [B|E]), [B|C], B, C, D, [B|E], B, E\}.
$$

Let $\mathsf{Q} = \{q_1, q_2, \ldots\}$; let id_P map the i^{th} element of $\mathsf{headterms}(P)$ (in the given order) to q_i; let $\mathsf{Args} = \{\mathit{app}_1, \mathit{app}_2, \mathit{app}_3\}$ and let argpos be the obvious mapping into this set; let $\mathsf{V} = \{a, b, c, d, \ldots\}$, and let $\mathsf{varid}(A) = a, \mathsf{varid}(B) = b$ etc. Then S^P_{var} is the following mapping.

$$
\begin{array}{lll}
\mathit{append}([\,], A, A) \mapsto \mathsf{type} & \mathit{append}([B|C], D, [B|E]) \mapsto \mathsf{type} & D \mapsto d \\
[\,] \mapsto q_2 & [B|C] \mapsto q_6 & [B|E] \mapsto q_{10} \\
A \mapsto a & B \mapsto b & B \mapsto b \\
A \mapsto a & C \mapsto c & E \mapsto e
\end{array}
$$

The mapping S^P_{arg} is given as follows.

$$
\begin{array}{lll}
\mathit{append}([\,], A, A) \mapsto \mathsf{type} & \mathit{append}([B|C], D, [B|E]) \mapsto \mathsf{type} & D \mapsto \mathit{app}_2 \\
[\,] \mapsto \mathit{app}_1 & [B|C] \mapsto \mathit{app}_1 & [B|E] \mapsto \mathit{app}_3 \\
A \mapsto \mathit{app}_2 & B \mapsto b & B \mapsto b \\
A \mapsto \mathit{app}_3 & C \mapsto c & E \mapsto e
\end{array}
$$

It can be seen that S^P_{var} distinguishes more states than S^P_{var}, and hence will lead to a finer-grained analysis.

4.1 The Abstract Domains

We now define two sets of NFTAs. The variable-based domain is the more fine-grained, and is intended to capture a separate set of terms for each position in each clause head. The argument-based domain only captures one set corresponding to each argument of a predicate.

Define $\Delta^{\Sigma}_{\mathsf{any}}$ to be the set of transitions $\{f_j^{n_j}(\{\mathsf{any}\},\ldots,\{\mathsf{any}\}) \to \{\mathsf{any}\} \mid f_j^{n_j} \in \Sigma\}$, where Σ is a finite set of function symbols. Every element in Term_Σ is accepted by the NFTA $\langle\{\mathsf{any}\},\{\mathsf{any}\},\Sigma,\Delta^{\Sigma}_{\mathsf{any}}\rangle$. The state $\{\mathsf{any}\}$, though it can be regarded as if it were an ordinary state, is treated specially for efficiency reasons. In particular, we do not eliminate ϵ-transitions of the form $\{\mathsf{any}\} \to q$.

Definition 3. *Variable-Based and Argument-Based Domains*
Let P be a definite logic program, and let Σ be the set of function and predicate symbols in P. Let $R_d^P = \mathsf{range}(S_d^P)$, $d \in \{var, arg\}$ and let $Q_d^P = 2^{R_d^P}$. Let Δ_d^P be the set of transitions $\{f_j^{n_j}(q_1,\ldots,q_{n_j}) \to q \mid f_j^{n_j} \in \Sigma, \{q_1,\ldots,q_{n_j},q\} \subseteq Q_d^P\}$. Note that the states $q_1,\ldots,q_{n_j}$, and q are not elements of $\mathsf{range}(S_d^P)$, but rather sets of elements.

Then the variable-based *domain for P, called D_P^{var} is the following set of automata.*

$$\{\langle Q_{var}^P, \{\mathsf{type}\}, \Sigma, \Delta' \cup \Delta^{\Sigma}_{\mathsf{any}}\rangle \mid \Delta' \subseteq \Delta_{var}^P\}$$

The argument-based *domain for P, called D_P^{arg} is the following set of automata.*

$$\{\langle Q_{arg}^P, \{\mathsf{type}\}, \Sigma, \Delta' \cup \Delta^{\Sigma}_{\mathsf{any}}\rangle \mid \Delta' \subseteq \Delta_{arg}^P\}$$

In the above definition, it can be seen that the two domains D_P^{var} and D_P^{arg} differ only in the choice of the set of states of the automata, which are determined by the range of the S_{var}^P and S_{arg}^P functions respectively. Note that $\mathsf{range}(\mathsf{S}_{var}^P)$ and $\mathsf{range}(\mathsf{S}_{arg}^P)$ are finite, and hence the domains D_P^{var} and D_P^{arg} are finite.

Let $R_1 = \langle Q, \{\mathsf{type}\}, \Sigma, \Delta_1\rangle$ and $R_2 = \langle Q, \{\mathsf{type}\}, \Sigma, \Delta_2\rangle$ be two elements of D_P^d, $d \in \{var, arg\}$. We have a partial order $\sqsubseteq$ such that $R_1 \sqsubseteq R_2$ iff $\Delta_1 \subseteq \Delta_2$. The minimal element R_d^{min} is $\langle Q_d^P, \{\mathsf{type}\}, \Sigma, \emptyset\rangle$, and the maximal element R_d^{max} is $\langle Q_d^P, \{\mathsf{type}\}, \Sigma, \Delta_d^P \cup \Delta^{\Sigma}_{\mathsf{any}}\rangle$, $d \in \{var, arg\}$, and we have complete lattices $(D_d^P, \{\mathsf{type}\}, R_d^{min}, R_d^{max})$.

Define the concretisation functions $\gamma_d : D_P^d \to D_P$, $d \in \{var, arg\}$, as $\gamma_d(R) = L(R)$, where $L(R)$ is the language of the NFTA R. γ_d is monotonic with respect to the partial orders on D_P^d and D_P.

States that are sets containing more than one identifier represent products. For instance, in the transition $f(\{q_1,q_2\},\{q_3\}) \to \{q\}$, the state $\{q_1,q_2\}$ represents the product state. The set of terms accepted by $R_{\{q_1,q_2\}}$ is the product of $R_{\{q_1\}}$ and $R_{\{q_2\}}$. When representing an automaton, we write down only the transitions whose right hand side is a singleton, and the transitions for the products are not explicitly included. For convenience we will often refer to a singleton state $\{q\}$ simply as q, especially in examples.

5 Abstract Semantic Operations

We now proceed to define the operations reduce, $\mathsf{project}$, and $\bigsqcup$ for the variable-based and argument-based interpretations. As for the abstract domains, we define operations parameterised where necessary by a variable d that stands for either var or arg.

The reduce operation takes a clause body B and an element R of D_P^d. For convenience in presenting the operation, we use the RUL representation of R, that is, a transition $f(q_1,\ldots,q_{n_j}) \to q$ in R is represented in the form $q(f_j^{n_j}(x_1,\ldots,x_{n_j})) \leftarrow q_1(x_1),\ldots,q_{n_j}(x_{n_j})$. Let B be a clause body $p_1(\bar{t}_1),\ldots,p_m(\bar{t}_m)$: then $\mathsf{type}(B)$ is the conjunction $\mathsf{type}(p_1(\bar{t}_1)),\ldots,\mathsf{type}(p_m(\bar{t}_m))$.

Definition 4. reduce

Let P be a definite program, B be a clause body in P, and $R \in D_P^d$ be an NFTA, with transitions Δ represented as an RUL program. Let τ be an SLD-tree for $\Delta \cup \{\leftarrow \mathsf{type}(B)\}$. Then define $\mathsf{reduce}(B, R) = \{E_1, \ldots, E_r\}$, where $\leftarrow E_1, \ldots, \leftarrow E_r$ is the set of all goals from τ, satisfying the conditions that

(i) *$\leftarrow E_i$ is the first goal on its branch of τ that contains no function symbols, for $0 \leq i \leq r$;*
(ii) *for each set of predicates in E_i all of which have the same argument, say $\{q'_1, \ldots, q'_p\}$, $\mathsf{nonempty}(R_{\bar{q}})$ holds, where $\bar{q} = q'_1 \times \cdots \times q'_p$, for $0 \leq i \leq r$.*

The idea of reduce is to "solve" a clause body with respect to an NFTA. We can think of it as "partially evaluating" the clause body (after transforming it by the type operation) using the transitions of the NFTA, until all the predicate and function symbols in B have been eliminated. The order of selection of literals in the construction of the SLD tree does not affect the values of $\{E_1, \ldots, E_r\}$. If there are k function symbols in B, then exactly k resolution steps are required to remove them, since each transition (RUL clause) contains exactly one function symbol in its left hand side, and no function symbol can be introduced by a resolution step, since all the head variables of RUL clauses are distinct, and each head variable occurs exactly once in the body. We then have to perform an emptiness check on the product of the automata corresponding to repeated variables.

The $\mathsf{project}_d$ operation ($d \in \{var, arg\}$) takes a clause head H and one of the conjunctions E returned by the reduce operation. It returns a set of transitions.

Definition 5. $\mathsf{project}_d$

Let P be a definite program, $H \leftarrow B$ be a clause in P, $R \in D_P^d$ be an NFTA, and $E \in \mathsf{reduce}(B, R)$. Then $\mathsf{project}_d(H, E)$ is a set of transitions defined as follows.

$$
\begin{array}{lll}
\mathsf{project}_d(H, E) = & & \\
& \{f(\{q_1\}, \ldots, \{q_n\}) \to \{q\} \mid & f(t_1, \ldots, t_n) \text{ is a subterm of } H, \\
& & \mathsf{S}_d^P(f(t_1, \ldots, t_n)) = q, \\
& & \mathsf{S}_d^P(t_i) = q_i, 1 \leq i \leq n\} \\
& \bigcup & \\
& \{q' \to \{q\} \mid & x \text{ is a variable in } H, \\
& & \mathsf{S}_d^P(x) = q, \\
& & q' = \mathsf{restrict}(E, x)\}
\end{array}
$$

The subsidiary function $\mathsf{restrict}(E, x)$ returns $\{\mathsf{any}\}$, if x does not occur in E, otherwise it returns $\{q_1, \ldots, q_m\} \setminus \{\mathsf{any}\}$, if $q_1(x), \ldots, q_m(x)$ are the occurrences of predicates with argument x in E.

The abstract interpretation is completed by defining $\bigsqcup_P^d(S)$ (where S is a set of sets of transitions) to be the NFTA $\langle Q_d^P, \{\mathsf{type}\}, \Sigma, \Delta\rangle$ where $\Delta = \mathsf{elim}_\epsilon(\bigcup S) \cup \Delta_{\mathsf{any}}^{\Sigma}$. Thus the result of $\bigsqcup_d^P(S)$ is an element of D_P^d. Finally, define the abstract interpretation to be $\mathsf{lfp}(T_P^d)$, where

$$T_P^d(R) = \bigsqcup_P^d \{\mathsf{project}_d(H, \theta) \mid H \leftarrow B \in P, \theta \in \mathsf{reduce}(B, R)\}.$$

As noted as the end of Section 4, we do not represent product automata explicitly. However, when eliminating ϵ-transitions of the form $\{q_1, \ldots, q_n\} \to \{q\}$, we have to calculate the product corresponding to $\{q_1, \ldots, q_n\}$, in order to derive the transitions with right hand side $\{q\}$.

Example 2. Let P be the *append* program. In the first application of T^P_{var} we have:

$$\mathsf{reduce}(true, R_{min}) = \{true\} \qquad \mathsf{reduce}(append(C, D, E), R_{min}) = \emptyset.$$

For the first clause, $\mathsf{project}_{var}(append([\,], A, A)$ gives these transitions.

$$append(q_2, a, a) \to \mathsf{type} \qquad [\,] \to q_2 \qquad \mathsf{any} \to a$$

No transitions are returned from the second clause. On the second iteration, the first clause returns the same result. reduce applied to $append(C, D, E)$ returns the conjunction $(q_2(C), a(D), a(E))$, since we can unfold $append(C, D, E)$ using the transition (in RUL form) $\mathsf{type}(append(X, Y, Z)) \leftarrow q_2(X), a(Y), a(Z)$ obtained on the first step. Thus $\mathsf{project}$ gives the following transitions for the second clause head.

$$\begin{array}{llll}
append(q_6, d, q_{10}) \to \mathsf{type} & [b|c] \to q_6 & [b|e] \to q_{10} & q_2 \to c \\
a \to d & a \to e & \mathsf{any} \to b &
\end{array}$$

Adding these to the results of the first iteration and eliminating ϵ-transitions we obtain the following.

$$\begin{array}{llll}
append(q_6, d, q_{10}) \to \mathsf{type} & [b|c] \to q_6 & [b|e] \to q_{10} & [\,] \to c \\
\mathsf{any} \to d & \mathsf{any} \to e & \mathsf{any} \to b &
\end{array}$$

The third iteration yields the following new transitions, after eliminating ϵ-transitions.

$$[b|c] \to c \qquad [b|e] \to e$$

No new transitions are added on the fourth iteration, thus the least fixed point has been reached.

The argument-based approximation generates the following sequence of results: (only the new transitions on each iteration are shown).

$$\begin{array}{llllll}
(1) & append(app_1, app_2, app_3) \to \mathsf{type} & [\,] \to app_1 & \mathsf{any} \to app_2 & \mathsf{any} \to app_3 & \\
(2) & [b|c] \to app_1 & [\,] \to c & [b|e] \to app_3 & \mathsf{any} \to e & \mathsf{any} \to b \\
(3) & [b|c] \to c & [b|e] \to e & & &
\end{array}$$

Considering the first argument of *append*, we can see that the variable-based analysis is more precise. For instance, the term $append([a], [\,], [\,])$ is accepted by the second automaton but not by the first. This is because the two clauses of the *append* program are distinguished in the first, with two states (q_2 and q_6) describing the first argument in the two clauses respectively. A single state app_1 describes the first argument in the argument-based analysis. However, in this case (though not always), the precision of the variable-based analysis could be recovered from the argument-based analysis. We will discuss this further in Section 8. Further, note that the derived automata are not minimal in the number of states. For example the states c and e could be eliminated in the argument-based analysis, giving an equivalent more compact result.

$$\begin{array}{llll}
append(app_1, app_2, app_3) \to \mathsf{type} & [\,] \to app_1 & \mathsf{any} \to app_2 & \mathsf{any} \to app_3 \\
[b|app_1] \to app_1 & [b|app_3] \to app_3 & \mathsf{any} \to b &
\end{array}$$

5.1 Soundness of the Abstract Interpretations

The convergence of the sequence depends on the monotonicity of T_P^{var} and T_P^{arg} respectively, and the finiteness of the domains D_P^{var} and D_P^{arg}. Space does not permit a detailed proof of monotonicity, but it follows from the monotonicity of reduce in its second argument.

To show the soundness of the analyses requires proving that $\mathsf{lfp}(T_P) \subseteq \gamma_d(\mathsf{lfp}(T_P^d))$, $d \in \{var, arg\}$. Again, only a brief justification can be given here. The result follows in the framework of abstract interpretation [13] after showing that for all $R \in D_d^P$, $T_P(\gamma_d(R)) \subseteq \gamma_d(T_P^d(R))$. Informally, if t can be "generated" by applying T_P to the set of atoms accepted by automaton R (that is, $\gamma_d(R)$), then we can show that t is accepted by the automaton "generated" by applying T_P^d to R.

6 Implementation Aspects

We have implemented both the variable-based and the argument-based analyses. They share the same core semantics, and the code differs only in the part implementing the $\mathsf{project}$ operators, which takes into account the different relationships between program points and automata states.

6.1 Domain-Independent Optimisations

The presentation in Section 5 is naive from the implementation point of view, as it suggests that the sequence of approximations converging to the fixed point is computed by applying T_P^{var} (or T_P^{arg}) repeatedly to the complete accumulated result.

Various domain-independent optimisations are well known and have been applied in our implementation. We followed the pattern of our previous work on bottom-up analysis of logic programs [19, 18, 17]. The most important optimisations are the decomposition into strongly connected components (SCCs) of the predicate dependency graph of the program being analysed, and a variant of the "semi-naive" optimisation.

There are other domain-independent optimisations that could be included, such as the "chaotic iteration strategy" of Bourdoncle [3], and "eager evaluation" [36].

6.2 Domain-Dependent Optimisations

The operation $\bigsqcup_P$ for the two interpretations is defined as the union of sets of transitions, followed by the elimination of ϵ-transitions. This accords with the partial order on the domains, and has a conceptual simplicity. The successive applications of T_P^d simply keep on adding transitions until no new ones are generated. However, many redundant transitions can be generated, and the number of transitions is the major factor in the cost of expensive operations such as computing products of automata.

Thus in our implementation of $\bigsqcup_P$ the redundant transitions are removed from the automata. In the example in Section 5, the transition $[b|e] \to e$ can be removed from the variable-based analysis, and the transitions $[b|e] \to app_3$ and $[b|e] \to e$ from the argument-based analysis.

This optimisation implies that the sequence of automata generated in the sequence does not necessarily monotonically increase with respect to the partial order on the domain, since transitions can be removed as well as added. Convergence is still guaranteed due to the finiteness of the domain (and we take care not to introduce the same transition more than once). Soundness is obviously preserved since $\gamma_d(R) = \gamma_d(R')$ if R differs from R' only in the presence of redundant transitions. Alternatively, we could use the standard technique of constructing a domain and partial order on the domain, based on equivalence classes of automata with respect to the equivalence relation $\cong$. Clearly removing redundant transitions from an automaton yields an element of the same equivalence class.

6.3 Checking Non-Emptiness of Product Automata

Our experiments show that large numbers of states and transitions can be generated from user-written programs, as can be seen from Tables 1 and 2. It is therefore essential to implement the basic domain operations as efficiently as possible. In particular, the check for emptiness within the reduce operation is critical. Non-emptiness of an automaton can be checked in time linear in the size of the automaton, but we are required to check the emptiness of product automata, which is EXPTIME-complete [12].

We store the non-empty products that arise during the analysis as a table of tuples $\langle q'_1, \ldots, q'_p \rangle$. Suppose that during the reduce operation we have to check $\mathsf{nonempty}(R_{\bar{q}})$ where $\bar{q} = q'_1 \times \cdots \times q'_p$. We first check to see whether $R_{\bar{q}}$ has already been shown to be non-empty, that is, whether $\langle q'_1, \ldots, q'_p \rangle$ is already tabulated. If so, then the monotonicity of T_P^d implies that it is still non-empty *even if the definitions of* $q'_1, \ldots q'_p$ *have changed since non-emptiness was established.* To check non-emptiness of a product that has not yet been shown to be non-empty, we must first compute the transitions in the product. However, the table of non-empty products can be exploited again. As described by Comon *et al.* the non-emptiness check involves treating each transition $f(q_1, \ldots, q_n) \to q$ as a propositional formula $q_1 \wedge \ldots \wedge q_n \to q$. Non-emptiness of an automaton R_s reduces to checking that s follows from the set of propositional Horn formulas obtained from the transitions of R_s. For each such formula derived from the product automaton we can strike out any q_j that is already known to be non-empty (since in the propositional form it is already *true*).

Example 3. The use of the table of non-empty products is illustrated by the analysis of the naive reverse program.

$$rev([\,],[\,]) \leftarrow true \qquad rev([A|B],C) \leftarrow rev(B,D), append(D,[A],C)$$

The definition of *append* is as before, and assume that it has already been analysed (as the lowest SCC component) using the argument-based interpretation. The first iteration on *rev* yields transitions

$$rev(rev_1, rev_2) \to \mathsf{type} \qquad [\,] \to rev_1 \qquad [\,] \to rev_2$$

The next iteration applies reduce to the body of the second clause for *rev*. This requires checking the non-emptiness of the product $rev_2 \times app_1$ due to the repeated variable D. Computing the product of rev_2 and app_1 we obtain the propositional formula $true \to (rev_2 \times app_1)$, hence $rev_2 \times app_1$ is non-empty. Thus the following transitions are generated.

$$[a|b] \to rev_1 \qquad \mathsf{any} \to rev_2 \qquad \mathsf{any} \to a \qquad [\,] \to b$$

On the third iteration, we again must check non-emptiness of $rev_2 \times app_1$ but since it is already known to be non-empty we do not need to recompute the product. Note that the product is in fact larger than on the first iteration. The final transition to be added is $[a|b] \rightarrow b$.

We use a balanced 2-3-4 tree structure (that is, a B-tree of order 4) to store the transitions and the table of non-empty products. In the tree of transitions, the primary key is the state on the right-hand-side of the transition; within each record we use the function symbol on the left of the transition as a secondary key.

The elimination of unnecessary states, as illustrated in Example 2, trades off in general with an increase in the number of transitions. The choice of whether to eliminate is thus in general a heuristic matter. We adopt the following strategy. Any state $\{q\}$ that is defined by a single ϵ-transition $q' \rightarrow \{q\}$ (before the elimination of ϵ-transitions in the $\bigsqcup$ operation) is eliminated and replaced by q' wherever it occurs. Thus we eliminate states without increasing the number of transitions. We can keep a list of such eliminated states during the analysis, in case we need to access information about the program point (which will always be a variable in a clause head) represented by $\{q\}$. The analysis results given by q' can be applied to that program point.

For goal-dependent analysis we used "query-answer" transformations, related to "magic-set" transformations, to achieve a goal-dependent analysis in a bottom-up semantic framework [11, 15, 19]. This is a fairly crude but easily implemented technique for goal-directed analysis. Techniques such as "induced magic" [10] would doubtless improve performance.

7 Experiments

Some of the potential applications of set-constraint-based analysis were mentioned in Section 1. Our experiments were selected to show a range of different kinds of analysis, ranging from goal-independent type inference to planning and verification problems.

The implementation was developed in Ciao-Prolog [5]. The experiments were run in SICStus Prolog v. 3.8.6 under Solaris using a machine with two Ultrasparc processors running at 200 MHz.

Table 1 shows the results for goal-independent analysis, and Table 2 gives the results of analysing the program with respect to a goal. The first group of benchmarks consists of a standard set of test programs widely available. To these we added the Aquarius compiler of Van Roy [35]. The second set of benchmarks are planning programs, which we obtained from [4]. For these, there is a given goal, and the aim of the analysis is to show that the goal has no solution. For these, an indication is provided ($\surd$) as to whether the analysis did prove the failure of the goal (the "F" column in Table 2). The variable-based analysis is more precise over these examples, showing failure in several cases where the argument-based analysis cannot.

The programs in the first group of benchmarks do not always have a clear entry point, and sometimes contain dead code with respect to the apparent entry point, so the significance of the goal-dependent analyses is variable. The goal-dependent result for the Aquarius compiler in particular seems meaningless. They are all included for completeness. A "-" indicates that the analysis did not terminate in the resources available.

			Variable-Based		Argument-Based	
Program	Clauses	Preds	Transitions	Time (secs)	Transitions	Time (secs)
cs_r	109	37	462	0.56	245	0.26
disj_r	80	43	220	0.23	132	0.17
gabriel	45	20	165	0.18	82	0.08
kalah	88	45	297	0.30	176	0.19
peep	227	22	832	1.19	279	0.57
pg	18	10	62	0.07	32	0.04
plan	29	16	118	0.09	67	0.08
press	155	50	627	0.97	302	0.35
qsort	6	3	31	0.04	11	0.01
queens	9	5	29	0.03	15	0.02
read	161	43	438	0.55	186	0.42
aquarius	4192	1471	20075	41.46	7464	14.30
odd_even	4	3	8	0.01	5	0.01
wicked_oe	5	4	10	0.01	10	0.01
appendlast	5	3	22	0.01	16	0.01
reverselast	5	3	17	0.02	13	0.01
nreverselast	7	4	30	0.03	21	0.02
schedule	13	7	62	0.04	40	0.05
multisetl	6	4	14	0.02	11	0.01
multiseto	8	2	13	0.02	8	0.02
blockpair2o	16	4	88	0.04	77	0.05
blockpair3o	16	4	105	0.07	55	0.06
blockpair2l	15	6	117	0.07	104	0.02
blockpair3l	15	6	134	0.11	112	0.06
blocksol	14	6	109	0.07	98	0.05

Table 1. Results for Goal-Independent Analysis

The results show that the argument-based interpretation is faster than the variable-based interpretation. Both execution time and the number of transitions in the final result is typically approximately halved in the argument-based interpretation.

8 Discussion and Conclusions

The results in Section 7 show that the argument-based interpretation is faster than the variable-based interpretation. Although there is a loss of precision associated with the argument-based interpretation, it can often be regained. Simply apply the T_P^{var} function to the result of the argument-based analysis. That is, compute $T_P^{var}(\mathsf{lfp}(T_P^{arg}))$. This projects the results of the argument-based analysis onto the domain of the variable-based analysis, producing a separate result for each position in the clause heads. In general, $\mathsf{lfp}(T_P^{var}) \subseteq T_P^{var}(\mathsf{lfp}(T_P^{arg}))$, but we have not yet made a detailed comparison of the relative precision of the two analyses. For many programs, the two are identical. To increase precision further, we could compute the limit (or any finite prefix) of the finite decreasing sequence $A, T_P^{var}(A), T_P^{var}(T_P^{var}(A)), \ldots$, where $A = \mathsf{lfp}(T_P^{arg})$).

			Variable-Based			Argument-Based		
Program	Clauses	Preds	Transitions	Time (secs)		Transitions	Time (secs)	F
cs_r	225	74	971	3.29		372	1.03	
disj_r	154	86	360	1.18		254	0.89	
gabriel	83	40	203	0.50		108	0.29	
kalah	171	90	67	0.27		50	0.28	
peep	318	44	888	2.91		393	1.1	
pg	34	20	88	0.36		74	0.17	
plan	58	32	152	0.37		57	0.16	
press	278	100	838	4.16		554	1.93	
qsort	13	6	47	0.10		16	0.04	
queens	18	10	40	0.06		24	0.07	
read	352	86	236	1.23		125	0.98	
aquarius	9122	2942	129	23.21		32	25.38	
					F			F
odd_even	9	6	14	0.02	√	9	0.02	√
wicked_oe	13	8	15	0.03	√	15	0.03	√
appendlast	10	6	25	0.03	√	18	0.02	√
reverselast	10	6	31	0.05	√	23	0.03	×
nreverselast	14	8	48	0.08	×	36	0.06	×
schedule	25	14	62	0.16	√	47	0.13	√
multisetl	12	8	23	0.04	√	20	0.07	×
multiseto	16	4	67	0.26	√	36	0.12	√
blockpair2o	62	14	-	-	×	-	-	×
blockpair3o	62	14	-	-	×	-	-	×
blockpair2l	26	12	276	2.4	√	250	1.12	×
blockpair3l	26	12	223	2.62	√	258	1.11	×
blocksol	24	12	223	1.85		206	0.65	

Table 2. Results for Goal-Dependent Analysis

8.1 Comparison With Type Inference by Abstract Interpretation

Comparing our analyses with other abstract interpretations over type domains [25, 34, 19, 13, 30], the main difference is that all previous work is based on deterministic types. That is, a type may have have at most one "case" for each function symbol. These correspond roughly to deterministic finite tree automata, and as noted in Section 2, these have less expressive power than NFTAs. For example, it is not possible to represent the set of lists terminating in the element a using deterministic automata. The relative precision non-deterministic regular types compared to deterministic ones is discussed by Podelski and Charatonik [7]. The other aspect of existing type analyses based on abstract interpretations is that they are defined on an infinite domain, and so require a widening in order for the analysis to terminate. Mildner [30] has made a detailed comparison of various widenings in the literature.

The use of an infinite domain of NFTAs along with a widening is in principle more precise than our approach, since widening can be delayed an arbitrary number of iterations. The widenings that appear in the literature do not give more precision; our goal-dependent analysis produces the same accuracy as the examples discussed by Van Hentenryck *et al.*, including those "that require the widening to be rather sophisticated" [34]. However, existing abstract interpretations are based on deterministic types; the combination of non-deterministic types and widening has not been investigated, to our knowledge.

In summary, the method we presented seems to compare favourably, both in precision and efficiency, to all other type inference abstract interpreters known to us. For applications such as planning and verification, the extra precision of non-deterministic types over deterministic ones is significant.

8.2 Comparison with Set-Constraint Analysis

The variable-based analysis can be compared with set-constraint analysis [22, 21] via the monadic approximation of a program presented by Frühwirth *et al.* [16]. The minimal model of the monadic program is equivalent to the solution of the set-constraints for the program. Our $\mathsf{project}_{var}$ operator can be seen as performing the monadic transformation dynamically during the analysis. We claim that our variable-based analysis computes the minimal model of the corresponding monadic program (our $\mathsf{project}_{var}$ operator mimics the monadic transformation), and thus can be seen as a method of solving set-constraints for logic programs.

The monadic transformation is attractive from the point of view of presenting set-constraint analysis, but direct use of the monadic transformation in the implementation of the analysis seems to be inadvisable. The transformation produces one copy of each clause body for every variable in its head. Solving these separately would be very inefficient. The somewhat awkward “pretend” variable, that is introduced in the monadic transformation of clauses with ground heads, is avoided in our approach.

We do not have an implementation of set-constraint solving against which to compare our implementation. Judging by our experiments and results reported in the literature, our approach is a practical alternative to set-constraint-solving algorithms. However, it does not seem likely that there are any inherent advantages in our approach to solving set-constraints. The main interest comes from combining set constraints with other analyses, in the framework of abstract interpretation.

8.3 Complexity and Scalability

Charatonik and Podelski remark that the worst-case complexity of set-based analysis is seldom encountered since types in user-written programs tend to be relatively small [7]. This does indeed seem to be true for “type analysis” applications of set constraints. However, for verification and planning problems, the types can grow very large since they can be combinatorial combinations of initial states present in the top goal. For instance, some of the planning problems discussed by Bruynooghe *et al.* [4] contain a procedure for checking equality of multisets. The procedure generates all permutations of the elements of one of the multisets. Set-based analysis is precise enough to generate a type containing all the permutations too, when the input sets are given. The two planning problems “blockpair2o” and ”blockpair3o” were too complex for our implementation and ran out of memory. In summary, the precision of set-based analysis is sometimes too good to be practical, and coarser domains or widening operators may be needed in the abstract interpretations. A coarser domain, such as one containing deterministic automata only, could be used for more intractable examples. Introduction of widenings is arguably more systematic and conceptually easier in the abstract interpretation approach than in the original framework of set constraints. The generic correctness conditions for widening operators are established, and the invention of widenings follows a pattern of identifying invariant parts of the approximations from one iteration to the next.

8.4 Future Work

An advantage of our approach to set constraint analysis is that it can be incorporated into existing abstract interpretation frameworks such as PLAI [6, 31] which forms part of the Ciao-Prolog pre-processor [24]. The aims of integrating set-constraint analysis into PLAI are to allow combination with other abstract domains, especially numerical approximations like convex hulls, and to have access to features of PLAI such as incremental analysis [23]. The pre-processor already includes a type analyser, and the greater precision available from set-based-analysis would increase its scope. To implement an abstract interpretation for a given domain in PLAI, a small number of domain-dependent operations have to be provided, such as abstract unification and projection. The transitions of the automata would be carried around the AND-OR tree of PLAI as "abstract substitutions". We can see no difficulty in principle in performing the integration, and this is the next stage in our research.

In conclusion, we have demonstrated that abstract interpretation over NFTAs for set-based analysis of logic programs is feasible, and we argue that there are conceptual and practical advaantages in following this approach. Future research will focus on integrating the analysis into a generic abstract interpretation framework, combining it with other abstract interpretations.

Acknowledgements

This work was performed while the first author was on study leave at the Universidad Politécnica de Madrid. The second author is supported in part by Spanish CICYT project TIC 99-1151 "EDIPIA". Some of the ideas presented here (in particular the tabulation of the non-empty products) were originally inspired by discussions between the first author and Andreas Podelski in January 1998. We thank Julio Peralta and the anonymous referees for their helpful comments.

References

1. A. Aiken. Set constraints: Results, applications, and future directions. In A. Borning, editor, *Principles and Practice of Constraint Programming (PPCP 1994)*, volume 874 of *Springer-Verlag Lecture Notes in Computer Science*, pages 326–335. Springer Verlag, 1994.
2. A. Aiken and E. L. Wimmers. Solving systems of set constraints (extended abstract). In *IEEE Symposium on Logic in Computer Science (LICS 1992)*, pages 329–340, 1992.
3. F. Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Formal Methods in Programming and their Applications*, volume 735 of *Springer-Verlag Lecture Notes in Computer Science*, pages 123–141, 1993.
4. M. Bruynooghe, H. Vandecasteele, D. A. de Waal, and M. Denecker. Detecting unsolvable queries for definite logic programs. *Journal of Functional and Logic Programming*, Special Issue 2, 1999.
5. F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López-García, and G. Puebla. The Ciao prolog system. reference manual. Technical Report CLIP3/97.1, School of Computer Science, Technical University of Madrid (UPM), August 1997. Available from http://www.clip.dia.fi.upm.es/.

6. F. Bueno, M. G. de la Banda, and M. Hermenegildo. Effectiveness of Abstract Interpretation in Automatic Parallelization: A Case Study in Logic Programming. *ACM Transactions on Programming Languages and Systems*, 21(2):189–238, March 1999.
7. W. Charatonik and A. Podelski. Directional type inference for logic programs. In G. Levi, editor, *Proceedings of the International Symposium on Static Analysis (SAS'98), Pisa, September 14 - 16, 1998*, volume 1503 of *Springer LNCS*, pages 278–294. Springer-Verlag, 1998.
8. W. Charatonik and A. Podelski. Set-based analysis of reactive infinite-state systems. In B. Steffen, editor, *Proc. of TACAS'98, Tools and Algorithms for Construction and Analysis of Systems, 4th International Conference, TACAS '98*, volume 1384 of *Springer-Verlag Lecture Notes in Computer Science*, 1998.
9. W. Charatonik, A. Podelski, and J.-M. Talbot. Paths vs. trees in set-based program analysis. In T. Reps, editor, *Proceedings of POPL'00: Principles of Programming Languages*, pages 330–338. ACM, ACM Press, January 2000.
10. M. Codish. Efficient goal directed bottom-up evaluation of logic programs. *Journal of Logic Programming*, 38(3):355–370, 1999.
11. M. Codish and B. Demoen. Analysing logic programs using "Prop"-ositional logic programs and a magic wand. In D. Miller, editor, *Proceedings of the 1993 International Symposium on Logic Programming, Vancouver*. MIT Press, 1993.
12. H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. *Tree Automata Techniques and Applications*. http://www.grappa.univ-lille3.fr/tata, 1999.
13. P. Cousot and R. Cousot. Formal language, grammar and set-constraint-based program analysis by abstract interpretation. In *Proceedings of the Seventh ACM Conference on Functional Programming Languages and Computer Architecture*, pages 170–181, La Jolla, California, 25–28 June 1995. ACM Press, New York, NY.
14. D. de Waal and J. Gallagher. The applicability of logic program analysis and transformation to theorem proving. In *Proceedings of the 12th International Conference on Automated Deduction (CADE-12), Nancy*, 1994.
15. S. Debray and R. Ramakrishnan. Abstract Interpretation of Logic Programs Using Magic Transformations. *Journal of Logic Programming*, 18:149–176, 1994.
16. T. Frühwirth, E. Shapiro, M. Vardi, and E. Yardeni. Logic programs as types for logic programs. In *Proceedings of the IEEE Symposium on Logic in Computer Science, Amsterdam*, July 1991.
17. J. Gallagher. A bottom-up analysis toolkit. In *Proceedings of the Workshop on Analysis of Logic Languages (WAILL); Eilat, Israel; (also Technical Report CSTR-95-016, Department of Computer Science, University of Bristol, July 1995)*, June 1995.
18. J. Gallagher, D. Boulanger, and H. Sağlam. Practical model-based static analysis for definite logic programs. In J. W. Lloyd, editor, *Proc. of International Logic Programming Symposium*, pages 351–365, 1995.
19. J. Gallagher and D. de Waal. Fast and precise regular approximation of logic programs. In P. Van Hentenryck, editor, *Proceedings of the International Conference on Logic Programming (ICLP'94), Santa Margherita Ligure, Italy*. MIT Press, 1994.
20. J. P. Gallagher and J. C. Peralta. Using regular approximations for generalisation during partial evaluation. In *Proceedings of the 2000 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'2000), Boston, Mass., (ed. J. Lawall)*, pages 44–51. ACM Press, January 2000.
21. N. Heintze. Practical aspects of set based analysis. In K. Apt, editor, *Proceedings of the Joint International Symposium and Conference on Logic Programming*, pages 765–769. MIT Press, 1992.
22. N. Heintze and J. Jaffar. A Finite Presentation Theorem for Approximating Logic Programs. In *Proceedings of the 17th Annual ACM Symposium on Principles of Programming Languages, San Francisco*, pages 197–209. ACM Press, 1990.

23. M. Hermenegildo, G. Puebla, K. Marriott, and P. Stuckey. Incremental Analysis of Constraint Logic Programs. *ACM Transactions on Programming Languages and Systems*, 22(2):187–223, March 2000.
24. M. V. Hermenegildo, F. Bueno, G. Puebla, and P. López. Program analysis, debugging, and optimization using the Ciao system preprocessor. In D. De Schreye, editor, *Proceedings of ICLP 1999: International Conference on Logic Programming, Las Cruces, New Mexico, USA*, pages 52–66. MIT Press, 1999.
25. G. Janssens and M. Bruynooghe. Deriving descriptions of possible values of program variables by means of abstract interpretation. *Journal of Logic Programming*, 13(2-3):205–258, July 1992.
26. N. Jones. Flow analysis of lazy higher order functional programs. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*. Ellis-Horwood, 1987.
27. N. D. Jones and S. S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *Conference Record of the Ninth Symposium on Principles of Programming Languages*, pages 66–74. ACM Press, 1982.
28. D. Kozen. Set constraints and logic programming. *Information and Computation*, 143(1):2–25, 1998.
29. J. Lloyd. *Foundations of Logic Programming: 2nd Edition*. Springer-Verlag, 1987.
30. P. Mildner. *Type Domains for Abstract Interpretation: A Critical Study*. PhD thesis, Department of Computer Science, Uppsala University, 1999.
31. . Muthukumar and M. Hermenegildo. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *Journal of Logic Programming*, 13(2 and 3):315–347, July 1992.
32. A. Podelski, W. Charatonik, and M. Müller. Set-based failure analysis for logic programs and concurrent constraint programs. In S. D. Swierstra, editor, *Programming Languages and Systems, 8th European Symposium on Programming, ESOP'99*, volume 1576 of *LNCS*, pages 177–192. Springer-Verlag, 1999.
33. J. C. Reynolds. Automatic construction of data set definitions. In J. Morrell, editor, *Information Processing 68*, pages 456–461. North-Holland, 1969.
34. P. Van Hentenryck, A. Cortesi, and B. Le Charlier. Type analysis of prolog using type graphs. *Journal of Logic Programming*, 22(3):179–210, 1994.
35. P. Van Roy and A. M. Despain. High-performance logic programming with the Aquarius Prolog compiler. *IEEE Computer*, 25(1):54–68, 1992.
36. J. Wunderwald. Memoing evaluation by source-to-source transformation. In M. Proietti, editor, *Logic Program Synthesis and Transformation (LOPSTR'95)*, volume 1048 of *Springer-Verlag Lecture Notes in Computer Science*, pages 17–32, 1995.