Techniques for Scaling Up Analyses Based on Pre-interpretations

Gallagher, John Patrick; Henriksen, Kim Steen; Banda, Gourinath

Published in:
Logic Programming, 21st International Conference

Publication date:
2005

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

Citation for published version (APA):
Gallagher, J. P., Henriksen, K. S., & Banda, G. (2005). Techniques for Scaling Up Analyses Based on Pre-interpretations. In M. Gabbrielli, & G. Gupta (Eds.), *Logic Programming, 21st International Conference* (pp. 280-296). Springer.

Techniques for Scaling Up Analyses Based on Pre-interpretations*

John P. Gallagher**, Kim S. Henriksen, and Gourinath Banda

Computer Science, Building 42.1, P.O. Box 260,
Roskilde University, DK-4000, Denmark
{jpg, kimsh, gnbanda}@ruc.dk

Abstract. Any finite tree automaton (or regular type) can be used to construct an abstract interpretation of a logic program, by first determinising and completing the automaton to get a pre-interpretation of the language of the program. This has been shown to be a flexible and practical approach to building a variety of analyses, both generic (such as mode analysis) and program-specific (with respect to a type describing some particular property of interest). Previous work demonstrated the approach using pre-interpretations over small domains. In this paper we present techniques that allow the method to be applied to more complex pre-interpretations and larger programs. There are two main techniques presented: the first is a novel algorithm for determinising finite tree automata, yielding a compact "product" form of the transitions of the result automaton, that is often orders of magnitude smaller than an explicit representation of the automaton. Secondly, it is shown how this form (which is a representation of a pre-interpretation) can then be input directly to a BDD-based analyser of Datalog programs. We demonstrate through experiments that much more complex analyses become feasible.

1 Introduction and Motivation

In this paper we investigate the question of the scalability of logic program analyses based on pre-interpretations. This question is raised since pre-interpretations provide a general and flexible approach to specifying a variety of analyses, combining modes, types and other program specific properties. However, previous experiments [1,2,3] were limited to domains containing not more than four or five elements; furthermore for larger programs (especially those with predicates of high arity) experiments were restricted to even smaller domains. We discuss the reasons for this below. It was mentioned in earlier work that efficient representations of relations would be crucial to scalability.

An arbitrary regular type can be used to construct a pre-interpretation [2]. This contributes to the ease of specifying pre-interpretations, but adds another dimension to the complexity problem. A pre-interpretation can be orders of

* Work partially supported by European Framework 5 Project ASAP (IST-2001-38059).

** Partially supported by the CONTROL project funded by the Danish Natural Science Research Council, and the IT University of Copenhagen.

M. Gabbrielli and G. Gupta (Eds.): ICLP 2005, LNCS 3668, pp. 280–296, 2005.

magnitude larger than the regular type from which it is derived, when represented naively. This raises the question of whether the flexibility of this approach can be exploited for more complex analyses.

To summarise our conclusions, we show promising results for both aspects of the scalability problem. We give a new *determinisation algorithm* for finite tree automata, which returns the determinised automaton in a compact form. We then show how this compact form can be used directly in a *BDD*-based analyser for Datalog programs.

2 Preliminaries

In this section we recall those concepts pertaining to pre-interpretations and finite tree automata that concern us. We assume familiarity with standard logical concepts such as *interpretation*, *satisfiable* and *model* [4].

Pre-interpretations. Let P be a definite program and Σ the signature of its underlying language L; Σ is a set of ranked function and predicate symbols. A *pre-interpretation* of L consists of

1. a non-empty domain of interpretation D;
2. an assignment of an n-ary function $D^n \to D$ to each n-ary function symbol in Σ ($n \geq 0$).

A *domain atom* for a pre-interpretation J having domain D is an expression $p(d_1, \ldots, d_n)$ where p is an n-ary predicate symbol in Σ and $d_1, \ldots, d_n$ are elements of D. Let B_P^J be the set of domain atoms for pre-interpretation J and the signature of the language associated with program P. A model of a definite program P, based on pre-interpretation J, is some subset of B_P^J which satisfies P. A definite program has a least model for a given pre-interpretation. In particular, the least model for the Herbrand pre-interpretation is the usual declarative semantics of definite logic programs. The least model for a pre-interpretation J can be computed as the least fixpoint of a function $T_P^J : B_P^J \to B_P^J$.

A pre-interpretation assigns a domain element to each ground term in $\mathsf{Term}(\Sigma)$, its *denotation*. Let $f_J : D^n \to D$ be the function assigned to the n-ary function f by the pre-interpretation J. Then the denotation $\mathsf{Den}_J(t)$ of a term $t \in \mathsf{Term}(\Sigma)$ is defined as $\mathsf{Den}_J(f(t_1, \ldots, t_n)) = f_J(\mathsf{Den}(t_1), \ldots, \mathsf{Den}(t_n))$ ($n > 0$), and $\mathsf{Den}_J(t) = f_J(t)$ if t is a 0-ary function symbol.

Pre-interpretations and term properties. The domain elements of a pre-interpretation J define term properties. A domain element d corresponds to a term property p where for all terms t, $p(t)$ holds if and only if $\mathsf{Den}_J(t) = d$. Note that the properties defined by the elements of a pre-interpretation are disjoint and complete; each term has exactly one of the properties since it denotes exactly one domain element.

Abstract interpretation based on pre-interpretations. Static analysis of definite logic programs using (finite) pre-interpretations was set out in [5,6] and [1]. Earlier related ideas, not mentioning pre-interpretations, were developed by Corsini

et al. [7] and by Codish and Demoen [8]. We briefly summarise the approach; analysis consists of the construction of a pre-interpretation capturing some property of interest, followed by the computation of the least model with respect to that pre-interpretation. The implementation method is in three stages; let P be a program and J a pre-interpretation.

1. Represent J as a set of facts of the form $f(d_1, \ldots, d_n) \rightarrow d$, such that $f_J(d_1, \ldots, d_n) = d$, where f_J is the function assigned to f.
2. Transform P, introducing equalities until every non-variable appears in the left-hand-side of an equality, and no nested functions occur.
3. Convert to an *abstract domain program* by interpreting the introduced equalities as the pre-interpretation function. In practice this just means replacing the $=$ symbol by $\rightarrow$.

The stages of transformation are illustrated for a single clause below.

```
rev([X|Xs],Zs) :- rev(Xs,Ys), append(Ys,[X],Zs).
rev(U,Zs) :- rev(Xs,Ys), append(Ys,V,Zs), [X|Xs]=U, [X|W]=V, []=W.
rev(U,Zs) :- rev(Xs,Ys), append(Ys,V,Zs), [X|Xs]→U, [X|W]→V, []→W.
```

To continue the example, the pre-interpretation capturing the properties *ground* (`g`) and *non-ground* (`ng`) is given by the following facts defining the relation $\rightarrow$: $\{[] \rightarrow \mathtt{g}, [\mathtt{g}|\mathtt{g}] \rightarrow \mathtt{g}, [\mathtt{g}|\mathtt{ng}] \rightarrow \mathtt{ng}, [\mathtt{ng}|\mathtt{g}] \rightarrow \mathtt{ng}, [\mathtt{ng}|\mathtt{ng}] \rightarrow \mathtt{ng}\}$. The least model of the transformed program together with the facts defining the pre-interpretation is then computed.

Analysis based on pre-interpretations can be presented as an abstract interpretation [9]. The abstract domain $2^{B_P^J}$ is relational, capturing dependencies among arguments of a predicate, and condensing, implying that a bottom-up, goal-independent analysis yields results that lose no information with respect to goal-dependent analyses.

From regular types to pre-interpretations. In [2] it was shown that any term properties expressible by regular types could be transformed to a pre-interpretation, even if the properties were not disjoint. The process of constructing the pre-interpretation uses the algorithm for determinising a finite tree automaton. This means that one can start from term properties and build a pre-interpretation capturing those properties. More specifically, the pre-interpretation captures a set of disjoint properties derived from the original properties; for instance, given properties p_1, p_2 and p_3, the pre-interpretation might for example have domain elements corresponding to $p_1 \wedge p_3$, $p_2 \wedge p_3$, where these two properties were disjoint.

Determinisation of Finite Tree Automata. A *finite tree automaton* (FTA) is defined as a quadruple $\langle Q, Q_f, \Sigma, \Delta \rangle$, where Q is a finite set called *states*, $Q_f \subseteq Q$ is called the set of accepting (or final) states, Σ is a set of ranked function symbols and Δ is a set of *transitions*. Each element of Δ is of the form $f(q_1, \ldots, q_n) \rightarrow q$, where $f \in \Sigma$ and $q, q_1, \ldots, q_n \in Q$. We write f^n to indicate that function symbol f has arity n. and we often write the term $f^0()$ as f and call f a *constant*. Term_Σ is the set of *ground terms* (or *trees*) constructed from Σ in the usual way.

An FTA can be "run" on terms in Term_Σ; the details are omitted here, except to say that a successful run of a term and an FTA is one in which the term is *accepted* by one of the final states the FTA. Implicitly, a tree automaton R defines a set of terms, that is, a tree language, denoted $L(R)$, as the set of all terms that it accepts.

As far as expressiveness is concerned we can limit our attention to FTAs in which the set of transitions Δ contains no two transitions with the same left-hand-side. These are called *bottom-up deterministic* finite tree automata (DFTAs). For every FTA R there exists a bottom-up deterministic FTA R' such that $L(R) = L(R')$. A term can be accepted by at most one final state of a DFTA.

An automaton $R = \langle Q, Q_f, \Sigma, \Delta \rangle$ is called *complete* if for all n-ary functions $f \in \Sigma$ and states $q_1, \ldots, q_n \in Q$, $f(q_1, \ldots, q_n) \to q \in \Delta$.

A complete DFTA in which every state is an accepting state partitions the set of terms into disjoint subsets, one for each state, since every term is accepted by exactly one state.

Example 1. Let $\Sigma = \{[]^0, [\_|\_]^2, 0^0\}$, and let $Q = \{list, listlist, any\}$. We define the set Δ_{any}, for a given Σ, to be the following set of transitions.

$$\{f(\overbrace{any, \ldots, any}^{n \text{ times}}) \to any \mid f^n \in \Sigma\}$$

Let $Q_f = \{list, listlist\}$, $\Delta = \{[] \to list, [any|list] \to list, [] \to listlist, [list|listlist] \to listlist\} \cup \Delta_{any}$. The state (or regular type) *list* accepts terms in the set of lists of any terms, while the state *listlist* accepts terms in the set of lists whose elements are themselves lists. Clearly *listlist* is contained in *list*, which is contained in *any*.

The automaton is not bottom-up deterministic; a determinisation algorithm yields the DFTA $\langle Q', Q'_f, \Sigma, \Delta' \rangle$, where $Q' = \{q_1, q_2, q_3\}$, $Q'_f = \{q_1, q_2\}$ and $\Delta' = \{[] \to q_1, [q_1|q_1] \to q_1, [q_2|q_1] \to q_1, [q_1|q_2] \to q_2, [q_2|q_2] \to q_2, [q_3|q_2] \to q_2, [q_3|q_1] \to q_2, [q_2|q_3] \to q_3, [q_1|q_3] \to q_3, [q_3|q_3] \to q_3, 0 \to q_3\}$. q_1 corresponds

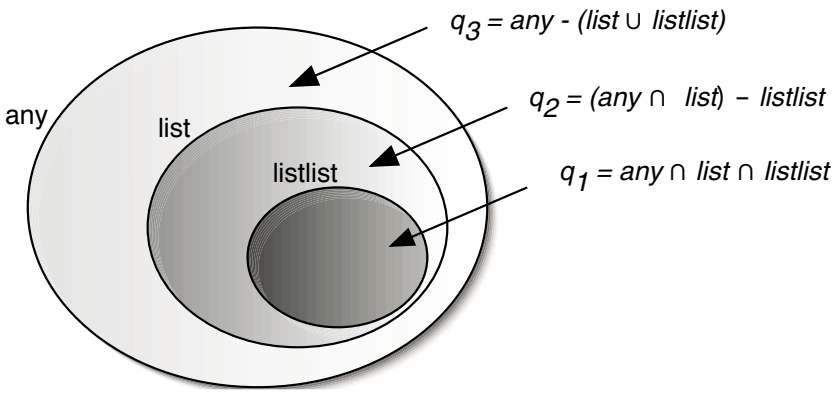

Fig. 1. The original types and the disjoint types from Example 1

to the set $any \cap list \cap listlist$, $q2$ to the set $(list \cap any) - listlist$, and q_3 to $any - (list \cup listlist)$. Thus q_1, q_2 and q_3 accept disjoint sets of terms. The original regular types and the disjoint types are shown in Figure 1. This automaton is also complete. In fact, any DFTA obtained from an FTA whose transitions include Δ_{any} (for the appropriate signature) is complete. □

3 An Algorithm for Determinisation

Product representation sets of transitions. The determinisation algorithm described below generates an automaton whose transitions are represented in product form, as described below, which is a more compact form and leads to a correspondingly more efficient determinisation algorithm. The main difference from the textbook algorithm is the form of the output, and in the explicit use of indices for efficient searching of the set of transitions. A *product transition* is of the form $f(Q_1, \ldots, Q_n) \to q$ where $Q_1, \ldots, Q_n$ are sets of states and q is a state. This product transition denotes the set of transitions $\{f(q_1, \ldots, q_n) \to q \mid q_1 \in Q_1, \ldots, q_n \in Q_n\}$. Thus $\prod_{i=1\ldots n} |Q_i|$ transitions are represented by a single product transition.

Example 2. The transitions of the DFTA generated in Example 1 can be represented in product transition form as follows. $\Delta' = \{[] \to q_1, 0 \to q_3, [\{q_1, q_2, q_3\}|\{q_3\}] \to q_3, [\{q_1, q_2\}|\{q_2\}] \to q_2, [\{q_1, q_2, q_3\}|\{q_1\}] \to q_1, [\{q_3\}|\{q_2\}] \to q_2\}$. Thus 4 product transitions replace the 9 transitions for $[\_|\_]^2$ shown in Example 1. There are other equivalent sets of product transitions, for example, $\Delta' = \{0 \to q_3, [\{q_1, q_2\}|\{q_3\}] \to q_3, [\{q_3\}|\{q_3\}] \to q_3, [\{q_1, q_2\}|\{q_2\}] \to q_2, [\{q_3\}|\{q_2\}] \to q_2[\{q_1, q_2\}|\{q_1\}] \to q_1, [\{q_3\}|\{q_1\}] \to q_1, [] \to q_1\}$.

3.1 A Determinisation Algorithm Generating Product Form

The algorithm developed in this section was based initially on the classical textbook algorithm [10]. It differs firstly by introducing an index structure to avoid traversing the complete set of transitions in each iteration of the algorithm, and secondly by noting that the algorithm only needs to compute explicitly the set of states of the determinised automaton. The set of transitions can be represented implicitly in the algorithm and generated later if required from the determinised states and the implicit form. However, in our approach the implicit form is close to product transition form and we will use this form directly. Hence, we never need to compute the full set of transitions and this is a major saving of computation. Let $\langle Q, Q_f, \Sigma, \Delta \rangle$ be an FTA. Consider the following functions.

- $\mathsf{qmap}_\Delta : (Q \times \Sigma \times \mathcal{N}) \to 2^\Delta$
  $\mathsf{qmap}_\Delta(q, f^n, j) = \{f(q_1, \ldots, q_n) \to q_0 \in \Delta \mid q = q_j\}$ for $1 \leq j \leq n$.
- $\mathsf{Qmap}_\Delta : (2^Q \times \Sigma \times \mathcal{N}) \to 2^\Delta$
  $\mathsf{Qmap}_\Delta(Q', f^n, j) = \bigcup\{\mathsf{qmap}_\Delta(q, f^n, j) \mid q \in Q'\}$.
- $\mathsf{states}_\Delta : 2^\Delta \to 2^Q$
  $\mathsf{states}_\Delta(\Delta') = \{q_0 \mid f(q_1, \ldots, q_n) \to q_0 \in \Delta'\}$.

- $\mathsf{fmap}_\Delta : \Sigma \times \mathcal{N} \times 2^{2^Q} \to 2^{2^\Delta}$
  $\mathsf{fmap}_\Delta(f^n, j, \mathcal{D}) = \{\mathsf{Qmap}_\Delta(Q', f^n, j) \mid Q' \in \mathcal{D}\} \setminus \emptyset$, for $1 \leq j \leq n$.
- $\mathsf{C} : 2^Q$
  $\mathsf{C} = \{\{q \mid f^0 \to q \in \Delta\} \mid f^0 \in \Sigma\}$
- $\mathsf{F}_\Delta : 2^{2^Q} \to 2^{2^Q}$
  $\mathsf{F}_\Delta(\mathcal{D}') = \mathsf{C} \cup \{\mathsf{states}_\Delta(\Delta_1 \cap \cdots \cap \Delta_n) \mid f^n \in \Sigma, \Delta_1 \in \mathsf{fmap}_\Delta(f^n, 1, \mathcal{D}'), \ldots, \Delta_n \in \mathsf{fmap}_\Delta(f^n, n, \mathcal{D}')\} \setminus \emptyset$

The subscript Δ is omitted in the context of some fixed FTA. The function qmap_Δ is an index on Δ, recording the set of transitions that contain a given state q at a given position in its left-hand-side. Qmap_Δ is the same index lifted to sets of states.

The algorithm finds the least set $\mathcal{D} \in 2^{2^Q}$ such that $\mathcal{D} = \mathsf{F}(\mathcal{D})$. The set $\mathcal{D}$ is computed by a fixpoint iteration as follows.

initialise $i = 0$; $\mathcal{D}_0 = \emptyset$
repeat $\mathcal{D}_{i+1} = \mathsf{F}(\mathcal{D}_i)$; $i = i + 1$ **until** $\mathcal{D}_i = \mathcal{D}_{i-1}$

It can be shown that the sequence $\mathcal{D}_0, \mathcal{D}_1, \mathcal{D}_2, \ldots$ increases monotonically (with respect to the subset ordering on 2^{2^Q}) and clearly there exists some i such that $\mathcal{D}_{i-1} = \mathcal{D}_i$ since Q is finite.

Example 3. Consider the following regular types (FTA transitions), in which each transition has been labelled to identify it conveniently. We have $Q = \{any, list\}$ and $\Delta = \{t_1, \ldots, t_5\}$.

$t_1 : [] \to list$ $\quad$ $t_3 : [] \to any$
$t_2 : [any|list] \to list$ $\quad$ $t_4 : [any|any] \to any$
$t_5 : f(any, any) \to any$

The qmap function is as follows:

$\mathsf{qmap}(list, cons, 1) = \emptyset$ $\quad$ $\mathsf{qmap}(list, cons, 2) = \{t_2\}$ $\quad$ $\mathsf{qmap}(list, f, 1) = \emptyset$
$\mathsf{qmap}(list, f, 2) = \emptyset$ $\quad$ $\mathsf{qmap}(any, cons, 1) = \{t_2, t_4\}$ $\quad$ $\mathsf{qmap}(any, cons, 2) = \upsilon\{t_4\}$
$\mathsf{qmap}(any, f, 1) = \{t_5\}$ $\quad$ $\mathsf{qmap}(any, f, 2) = \{t_5\}$

There is only one constant, $[]$, and $\mathsf{C} = \{\{any, list\}\}$. Initialise $\mathcal{D}_0 = \emptyset$; the iterations of the algorithm produce the following values.

1. $\mathcal{D}_1 = \{\{any, list\}\}$
2. $\mathcal{D}_2 = \{\{any, list\}, \{any\}\}$
3. $\mathcal{D}_2 = \mathcal{D}_3$ □

The determinised automaton can be constructed from the fixpoint $\mathcal{D}$ and Qmap. The set of states $\mathcal{Q}$ is $\mathcal{D}$ itself. The set of final states $\mathcal{Q}_f$ is $\{Q' \mid Q' \in \mathcal{Q}, Q' \cap Q_f \neq \emptyset\}$. The set of transitions is

$$\{f(Q_1, \ldots, Q_n) \to \mathsf{states}(\mathsf{Qmap}(Q_1, f, 1) \cap \cdots \cap \mathsf{Qmap}(Q_n, f, n)) \mid f^n \in \Sigma, Q_1 \in \mathcal{Q}, \ldots, Q_n \in \mathcal{Q}\}$$

The transition for each constant f^0 is $f^0 \to \{q \mid f^0 \to q \in \Delta\}$. Continuing Example 3, we obtain

$[] \to \{any, list\}$
$[\{any\}|\{any, list\}] \to \mathsf{states}(\mathsf{Qmap}(\{any\}, cons, 1) \cap \mathsf{Qmap}(\{any, list\}, cons, 2))$
$\quad \to \mathsf{states}(\{t_2, t_4\} \cap \{t_2, t_4\})$
$\quad \to \{any, list\}$
$[\{any\} \mid \{any\}] \to \mathsf{states}(\mathsf{Qmap}(\{any\}, cons, 1) \cap \mathsf{Qmap}(\{any\}, cons, 2))$
$\quad \to \mathsf{states}(\{t_2, t_4\} \cap \{t_4\})$
$\quad \to \{any\}$
$f(\{any\}, \{any\}) \to \mathsf{states}(\mathsf{Qmap}(\{any\}, f, 1) \cap \mathsf{Qmap}(\{any\}, f, 2))$
$\quad \to \mathsf{states}(\{t_5\} \cap \{t_5\})$
$\quad \to \{any\}$
and so on.

There are nine transitions in this small example. As we will see we can also obtain a more compact representation as a set of product transitions.

Implementation of the Algorithm. The function qmap is computed once at the start of the algorithm in time $O(|\Delta|)$, and it can be stored as a hash-table, which allows the computation of $\mathsf{qmap}(q, f, j)$ in constant time. The value of $\mathsf{Qmap}(Q', f, j)$ can thus be computed in $O(|Q|)$. $\mathsf{states}(\Delta')$ can be computed in $O(|\Delta|)$ after construction of a suitable index to the transitions.

The function fmap is maintained as a table, called ftable. As described above, the algorithm computes a sequence $\emptyset, \mathsf{F}(\emptyset), \mathsf{F}^2(\emptyset), \ldots$, where $\mathcal{D}_i = \mathsf{F}^i(\emptyset)$. Let $\mathcal{D}_i$ and $\mathcal{D}_{i+1}$ be successive values of the sequence. At the $i + 1^{th}$ stage of the algorithm values of the form $\mathsf{fmap}(f, j, \mathcal{D}_{i+1})$ are computed for each f and j. We use the property that $\mathsf{fmap}(f, j, \mathcal{D}_{i+1}) = \mathsf{fmap}(f, j, \mathcal{D}_i) \cup \mathsf{fmap}(f, j, (\mathcal{D}_{i+1} \setminus \mathcal{D}_i))$. The table entry $\mathsf{ftable}(f^n, j)$ holds the values of $\mathsf{fmap}(f, j, \mathcal{D}_i)$ on the i^{th} iteration of the algorithm. Hence on the next iteration only the new values of fmap, that is, $\mathsf{fmap}(f, j, (\mathcal{D}_{i+1} \setminus \mathcal{D}_i))$, need to be added to $\mathsf{ftable}(f, j)$.

The evaluation of the function F can also be optimised taking into account the newly computed values of fmap. Assuming the existence of the ftable, define a function F' as

$$\mathsf{F}'(\mathcal{D}_{new}) = \{\mathsf{states}(\Delta_1 \cap \cdots \cap \Delta_n) \mid \begin{array}{l} f^n \in \Sigma, \\ \Delta_1 \in \mathsf{ftable}(f^n, 1), \\ \ldots, \\ \Delta_j \in \mathsf{fmap}(f^n, j, \mathcal{D}_{new}), \\ \ldots, \\ \Delta_n \in \mathsf{ftable}(f^n, n), \\ 1 \le j \le n\} \setminus \emptyset \end{array}$$

Thus for each tuple $\Delta_1, \ldots, \Delta_n$, at least one component of the tuple must be chosen from $\mathcal{D}_{new}$, ensuring that each tuple $\Delta_1, \ldots, \Delta_n$ needs to be considered only once for each f^n during the execution of the algorithm. After performing these optimisations the algorithm can be summarised as follows.

$\mathcal{D} = \mathsf{C};\ \mathcal{D}_{new} = \mathcal{D};$
for $f^n \in \Sigma$
  for $j = 1$ **to** n
    $\mathsf{ftable}(f^n, j) = \emptyset$
  endfor
endfor
repeat
  $\mathcal{D}_{old} = \mathcal{D};$
  for $f^n \in \Sigma$
   for $j = 1$ **to** n
      $\mathsf{ftable}(f^n, j) = \mathsf{ftable}(f^n, j) \cup \mathsf{fmap}(f^n, j, \mathcal{D}_{new})$
   endfor
  endfor
  $\mathcal{D} = \mathcal{D} \cup \mathsf{F}'(\mathcal{D}_{new});$
  $\mathcal{D}_{new} = \mathcal{D} \setminus \mathcal{D}_{old}$
until $\mathcal{D}_{new} = \emptyset$

Complexity. For each $f^n \in \Sigma$, the computation time is dominated by the number of tuples $Q_1, \ldots, Q_n$ that have to be considered during the computation of F. This is $\prod_{i=1\ldots n} |\mathsf{fmap}(f, i, \mathcal{D})|$. The maximum size of $|\mathsf{fmap}(f, i, \mathcal{D})|$ is the number of possible right-hand-sides in the determinised transitions for a f, say k_f. This is 2^Q in the worst case, but in practice it is often much smaller. The number of tuples is in fact closely related to the set of product transitions generated as follows. As can be seen from Figure 2, this is usually much smaller than the set of transitions in the DFTA.

Let $f^n \in \Sigma$ and let $\mathcal{D}$ be the set of sets of states computed as the fixpoint in the algorithm. Then the set of product transitions for f^n $(n > 0)$ is

$$\{f(\mathsf{fmap}^{-1}(\Delta_1, f^n, 1), \ldots, \mathsf{fmap}^{-1}(\Delta_n, f^n, n)) \to \mathsf{states}(\Delta_1 \cap \cdots \cap \Delta_n) \mid \Delta_1 \in \mathsf{fmap}(f^n, 1, \mathcal{D}), \ldots, \Delta_n \in \mathsf{fmap}(f^n, n, \mathcal{D})\}$$

where $\mathsf{fmap}^{-1}(\Delta', f^n, i) = \{Q' \mid \mathsf{Qmap}(Q', f^n, i) = \Delta', Q' \in \mathcal{D}\}$. $\mathsf{fmap}^{-1}(\Delta', f^n, i)$ can be computed and stored during the evaluation of $\mathsf{fmap}(f^n, i, \mathcal{D})$. For the example above, the final values of the fmap function are

$$\mathsf{fmap}(cons, 1, \mathcal{D}) = \{\{t_2, t_4\}\} \qquad \mathsf{fmap}(cons, 2, \mathcal{D}) = \{\{t_2, t_4\}, \{t_4\}\}$$
$$\mathsf{fmap}(f, 1, \mathcal{D}) = \{\{t_5\}\} \qquad \mathsf{fmap}(f, 2, \mathcal{D}) = \{\{t_5\}\}$$

The values of fmap^{-1} are:

$$\mathsf{fmap}^{-1}(\{t_2, t_4\}, cons, 1) = \{\{any, list\}, \{list\}\} \quad \mathsf{fmap}^{-1}(\{t_2, t_4\}, cons, 2) = \{\{any, list\}\}$$
$$\mathsf{fmap}^{-1}(\{t_4\}, cons, 2) = \{\{any\}\} \quad \mathsf{fmap}^{-1}(\{t_5\}, f, 1) = \{\{any, list\}, \{list\}\}$$

From these values we obtain the following product transitions (including the transition for the constant []).

$$[\{\{any\}, \{any, list\}\} | \{\{any, list\}\}] \to \{any, list\}$$
$$[\{\{any\}, \{any, list\}\} | \{\{any\}\}] \to \{any\}$$
$$f(\{\{any\}, \{any, list\}\}, \{\{any\}, \{any, list\}\}) \to \{any\}$$
$$[] \to \{any, list\}$$

The two states $\{any\}$ and $\{any, list\}$ denote non-lists and lists respectively. The determinised automaton is a pre-interpretation over this two-element domain. In general, a state $\{q_1, \ldots, q_k\}$ in a determinised automaton represents those terms in the intersection of the original states $q_1, \ldots, q_k$, and not in any other state. Thus $\{any\}$ always stands for terms that are of type any that are not of some other type.

4 Computing Models of Datalog Programs

The essential task in performing an analysis using pre-interpretations is to compute the minimal Herbrand model of a (definite) Datalog program [11]. A definite Datalog program is a set of Horn clauses containing no function symbols with arity greater than zero. The Herbrand models of such programs are finite. In the abstract domain programs defined in Section 2, a pre-interpretation was represented by a set of facts (unit clauses) of the form $(f(d_1, \ldots, d_n) \rightarrow d) \leftarrow true$. Although there are function symbols occuring in such facts, we can easily represent the facts using a separate predicate for each function symbol; say pre_f is the relation corresponding to f. Then all atoms of form $f(d_1, \ldots, d_n) \rightarrow d$ would be represented as the function-free atom $pre_f(d_1, \ldots, d_n, d)$ instead. Since function symbols occur nowhere else in the abstract domain program, we are left with a Datalog program.

Efficient techniques for computing Datalog models have been studied extensively in research on deductive database systems [11], and indeed, many techniques (especially algorithms for computing joins) from the field of relational databases are also relevant. In the logic programming context, facts containing variables are also allowed; tabulation and subsumption techniques have been applied in a Datalog model evaluation system for program analysis [12].

The analysis method based on pre-interpretations is of course independent of which technique is used for computing the model of the Datalog program. Having transformed the analysis task to that of computing a Datalog program model, we are free to choose the best method available. We do not give a detailed account of the various techniques here, but remark only that current techniques allow very large Datalog programs to be handled [13].

Our previous experiments [2] used a Prolog implementation, which though it incorporated many optimisations such as computing SCCs and the semi-naive strategy, did not scale well in certain dimensions. In particular, programs containing predicates of high arity (such as the Aquarius compiler benchmark, which has some predicates with arity greater than 25) could not be analysed for domains with size greater than three. The number of possible tuples of arity n with a domain of size m is m^n, so this limitation is almost certain to apply to any tuple-based representation. It was pointed out in [2] that improved representations of finite relations was a key factor in scaling up to larger domains.

Computing Datalog models using BDDs. Our current work uses the BDD-based solver `bddbddb` developed by Whaley [14]. This tool computes the model of a Datalog program, and provides facilities for querying Datalog programs. It

is written in Java and can link to established BDD libraries using the Java Native Interface (JNI). Our experiments were conducted using `bddbddb` linked to the BuDDy package [15]. We wrote a front end to translate our abstract logic programs and pre-interpretations into the form required by `bddbddb`.

The possibility of using Boolean functions to represent finite relations[1] was exploited in model-checking [16]. Assume that a relation over D^n is to be represented, where D contains m elements. Then we code the m elements using $k = \lceil log_2(m) \rceil$ bits and introduce $n.k$ Boolean variables $x_{1,1}, \ldots, x_{1,k}, x_{2,1}, \ldots, x_{n,1}, \ldots, x_{n,k}$. A tuple in the relation is then a conjunction $x_{1,1} = b_{1,1} \wedge \ldots \wedge, x_{n,k} = b_{n,k}$ where $b_{i,1} \cdots b_{i,k}$ is the encoding of the i^{th} component of the tuple. A finite relation is thus a disjunction of such conjunctions. BDDs allow very large relations, translated in this way into Boolean formulas, to be represented compactly (though variable ordering is critical, and there are some relations that admit no compact representation).

In a BDD-based evaluation of a Datalog program, the solution of each predicate is thus represented as a Boolean formula (in BDD form) and the relational operations required to compute the model can be translated into operations on BDDs. For example, if we are solving the conjunction $p(A, B), q(B, C)$ we take the Boolean formulas representing the current solutions of p and q, say F_p and F_q and build a new BDD representing the formula $F_p \wedge F_q \wedge x_{2,1} = y_{1,1} \wedge \ldots \wedge x_{2,k} = y_{1,k}$ where $x_{1,1}, \ldots, x_{1,k}, x_{2,1}, \ldots, x_{2,k}$ and $y_{1,1}, \ldots, y_{1,k}, y_{2,1}, \ldots, y_{2,k}$ are the Boolean variables representing the respective arguments of p and q.

Representing and manipulating Boolean formulas is a very active research field and there are other techniques besides BDDs that are competitive. In logic-program analyses, multi-headed clauses have demonstrated good performance when compared to BDDs, for example [17].

5 From Product Representations to Datalog

The determinisation algorithm in Section 3 returns transitions in product form. Though this saves computation, we still need to represent the product form as a Datalog program, so that we can exploit techniques such as BDD-based evaluation of the model.

Consider a product transition $f(\{a, b\}, \{c, d, e\}) \rightarrow q$. As before, we can introduce a predicate for each function to replace the arrow relation, obtaining $pre_f(\{a, b\}, \{c, d, e\}, q)$. To represent this as a clause we could write the following.

$$pre_f(X, Y, q) \leftarrow member(X, [a, b]), member(Y, [c, d, e]).$$

To convert to Datalog we need only introduce a specialised *member* predicate for each set that occurs as an argument in a product transition. In the above case we obtain:

[1] We are indebted to Peter Stuckey for drawing our attention to the fact that BDD-based approaches could be applied to arbitrary Datalog programs.

$$pre_f(X,Y,q) \leftarrow m_1(X), m_2(Y). \quad m_2(c) \leftarrow true.$$
$$m_1(a) \leftarrow true. \quad m_2(d) \leftarrow true.$$
$$m_1(b) \leftarrow true. \quad m_2(e) \leftarrow true.$$

As a further optimisation, if some product transition has for some argument a set containing all of the determinised states, we may simply replace that argument by an anonymous variable (a "don't care" argument). Also, singleton sets $\{q\}$ can be replaced by q instead of introducing a deterministic *member* call. For the transitions produced from Example 3, the set of determinised states was $\{\{any\}, \{any, list\}\}$. (We can write these states as constants q_1, q_2 respectively). The product transitions are

$$[\{q_1, q_2\}|\{q_2\}] \rightarrow q_2$$
$$[\{q_1, q_2\}|\{q_1\}] \rightarrow q_1$$
$$f(\{q_1, q_2\}, \{q_1, q_2\}) \rightarrow q_1$$
$$[] \rightarrow q_2$$

The Datalog program is thus

$$pre_{cons}(\_, q_2, q_2) \leftarrow true.$$
$$pre_{cons}(\_, q_1, q_1) \leftarrow true.$$
$$pre_f(\_, \_, q_1) \leftarrow true.$$
$$pre_{nil}(q_2) \leftarrow true.$$

Introduction of don't care arguments is certainly important for tuple-based representations but probably not for BDD-based approaches. In any case it does no harm in the latter case.

6 Experiments

We now summarise the analysis procedure. The procedure takes two inputs: a program P to be analysed and a set of regular type definitions R expressing term properties of interest. The procedure then follows these steps.

1. Augment the types with a standard type *any* over the signature of the program, and determinise yielding transitions R_d in product form.
2. Transform P to an abstract domain program P_a (using flattened predicates pre_f to denote the pre-interpretation of function f as explained in the previous section).
3. Transform R_d to a suitable Datalog representation R_{dat}, again using the pre_f representation, together with the specialised *member* predicates for the product transitions (and optionally introduce don't care arguments).
4. Transform $P_a \cup R_{dat}$ to the syntax required by `bddbddb` and compute its least model.

`bddbddb` provides facilities for querying specific predicates rather than computing the whole model, which may be more useful in certain applications, especially

those where we are simply interested in whether a predicate has any solution at all. However, we simply computed the whole model in the experiments. All the experiments were carried out using a machine equipped with a Pentium IV 2.8GHz processor with Hyper Threading enabled, 512MB RAM, with Linux installed. The determinisation algorithm is implemented in Ciao-Prolog, and the `bddbddb` tool is implemented in Java, with a JNI interface to the BuDDy BDD package, which is implemented in C.

Experiments on determinisation. Figure 2 shows a few experimental results just illustrating the effect of the determinisation algorithm. For each input FTA, the table shows the number of states Q and transitions Δ, followed by the number of states in the output DFTA, Q_d. Three measures of the set of transitions are shown. First the total number of transitions Δ_d, followed by the size of the set of product transitions generated by the algorithm $\Delta_\prod$. Thirdly we show the size of another set of product transitions Δ_{dc} that is generated by locating "don't care" arguments. The final column is the time in seconds to compute the product form Δ_{dc} (which is almost identical to the time to compute $\Delta_\prod$).

The most important observation is the significant reduction in size of $\Delta_\prod$ and Δ_{dc} compared to Δ_d. Note also that the set of states in the DFTA can actually be less than the set of states in the input FTA, as in the `dnf` example. This is because, as is typical in automatically generated FTAs, there are many equivalent states in the input, and this redundancy is removed in the DFTA.

	FTA		DFTA				
Name	Q	Δ	Q_d	Δ_d	$\Delta_\prod$	Δ_{dc}	secs
`chr`	21	64	57	118837	242	86	0.09
`dnf`	105	803	46	6567	168	141	0.57
`mat1`	6	10	6	39	8	8	0.01
`mat2`	3	8	3	12	9	7	0.01
`ring`	5	12	5	30	14	11	0.01
`pic`	8	270	8	4989	274	280	0.15

Fig. 2. Determinisation results

The input FTAs are `chr`, a set of regular types for analysing a CHR transition system; `dnf`, the regular type inferred automatically by the abstract interpretation over DFTAs described in [18]; `mat1`, a set of types for an off-line binding time analysis of a matrix transposition program; `mat2`, the regular types from Example 1 augmented by two extra function symbols; `ring`, the regular types describing states in the token-ring analysis problem [2]; and `pic`, a set of regular types expressing properties of a PIC processor emulator. We were unable to determinise the `chr`, `dnf` or `pic` examples using an available toolkit for handling tree automata, Timbuk[2] [19].

[2] The author of Timbuk confirmed that the implementation followed the textbook algorithm and no special effort to optimise it had been made.

Experiments on model computation. We now describe some experiments with analyses that use both determinisation and model computation.

We performed three general kinds of experiment. Firstly, we analysed two larger standard benchmarks using general-purpose domains including groundness, and list types. The results shown are for the Aquarius compiler and the Chat parser. One domain (`dom1`) has four elements (ground-lists, non-ground-lists, ground-non-lists, non-ground-non-lists) but this is more complex than the two-element domains (such as POS [20]) reported previously for analysis of these programs [21,17]. Another (`dom2`) includes a fifth element (variable) as well as the ones mentioned above (and therefore the binary encoding requires three bits per element). This caused a much more complex analysis for the Aquarius compiler (see Figure 3). Secondly, we took an example of automatically generated regular types from a program (`dnf`) using the type inference system described in [18] and re-analysed the program with a pre-interpretation based on those types `dnftype`. The point of doing this is that further precision can be gained, since the type inference analysis is not relational, but derives an independent type for each variable of the program. Using analysis with a pre-interpretation, dependencies among the arguments can be derived. Thirdly, we analysed a program using some program-specific types `colours` written by the user. The purpose is to check that required properties hold. In our case the program analysed is a Coloured Petri Net emulator, implemented in Prolog, for the task scheduler of an operating system kernel for real-time embedded systems [22]. The user types describe the types (colours) of the tokens in the net.

None of these examples could be handled by our previous analyser employing a tuple-representation of the least model. In the case of the larger pre-interpretations, the results show that the product representation allows pre-interpretations that would have enormous numbers of transitions if written out in full.

For each experiment in Figure 3, the following information is reported: the name of the program (Prog) and the number of clauses it contains (Clauses); the name of the pre-interpretation (Domain); the number of states in the original FTA (Q); the number of transitions in the FTA (Δ); the number of states in the determinised automaton (Q_d); the number of transitions in the full determinised automaton (Δ_d), which is shown in brackets as this is not actually computed - it is just shown to underline the impracticality of computing this; the number of product transitions ($\Delta_{\prod}$); and finally the time taken, split into the pre-processing time and the actual model computation. The pre-processing is shown separately since `bddbddb` can be considerably optimised in this respect[3], and should in fact be linear in the size of the program.

Variable ordering can be critical to the effectiveness of BDDs. In the experiments we used the default textual order of variables occurring in the program, and this was satisfactory except for the `aquarius` program with `dom2`, which was unable to complete in one hour. `bddbddb` has various heuristics for selecting variable order but we have not yet succeeded in exploiting these effectively. An-

[3] Personal communication from the developer of `bddbddb`.

Prog	Clauses	Domain	Q	Δ	Q_d	(Δ_d)	$\Delta_{\prod}$	Pre-Process	Analyse
`aquarius`	4192	`dom1`	3	1933	4	(1130118)	1951	68.8s	3.0s
`aquarius`	4192	`dom2`	4	1934	5	(10054302)	1951	70.0s	1h+
`chat`	515	`dom1`	3	655	4	(20067)	433	1.6s	0.2s
`chat`	515	`dom2`	4	656	5	(86803)	433	1.6s	2.8s
`dnf`	33	`dnftype`	105	803	46	(6567)	141	0.5s	58.0s
`petri`	66	`colours`	16	65	16	(268436271)	89	1.2s	1.5s

Fig. 3. Experimental results for Model Computation

other aspect of the variable ordering issue is the binary encoding of the domain elements. For instance,[4] given domain elements $\{a, b, c, d\}$, with the encoding $a = 00, b = 01, c = 10, d = 11$, the relation $\{p(b), p(c)\}$ requires two BDD nodes, while the relation $\{p(a), p(b)\}$ can be represented with a single node. The situation is reversed with the encoding $a = 10, b = 00, c = 01, d = 11$.

7 Related Work and Conclusions

Analysis based on pre-interpretations was introduced some time ago [5,6,1]. Earlier related approaches were put forward [7,8]. Scalability of these approaches was not really investigated, except in the case of Boolean domains, where BDDs [21] and other representations [17] were applied.

Tree automata are increasingly being applied in static analysis e.g. [23,24,25,26,18,19]. It is well known that an arbitrary finite tree automaton (FTA) can be transformed to an equivalent bottom-up deterministic tree automaton (DFTA). Many important operations and properties of tree automata are stated in terms of DFTAs [10]. However, the transformation to deterministic form can result in an explosion of states and transitions, and so some previous attempts to use DFTAs directly in static analysis reported problems with scalability [25,27]. The possibility of using a product representation does not seem to have been investigated before, though other means of compressing tree automata have been studied [28].

Dawson *et al.* [12] described an approach to program analysis (for various target languages) using logic programs to express semantic properties. Computation in a Datalog program is fundamental to the approach. Their implementation uses optimisations such as tabling and subsumption, but presumably relies on a tuple-based representation of the model and hence scalability for large relations must be an issue. Whaley *et al.* [14] obtained very promising results, with more evidence of scalability, again using Datalog programs to represent properties, but using BDDs to represent relations. Iwaihara *et al.* [29] presented two different approaches for using BDDs to compute models of Datalog programs, including the one used in `bddbddb`. In future work we plan to compare other binary encodings of relations.

[4] This example was provided by one of the anonymous referees.

Conclusions. We have described two techniques for handling larger pre-interpretations and applying them to analyse larger programs. Firstly, we presented a novel determinisation algorithm for finite tree automata, which yields a compact representation of the result. This makes it possible to build pre-interpretations from regular types, that are much more complex than those described previously [2]. Secondly, we showed how analysis based on pre-interpretations can be computed using BDD-based methods (or any other technique able to compute models of Datalog programs). Such methods have proven their scalability in other domains, especially model-checking, and there is a reasonable hope of achieving greater scalability for logic program analysis using these techniques.

Much work is required, especially in investigating strategies for improving BDD-based computations, particularly variable orderings, but also strategies for solving clause bodies, where the order of solution of body atoms, and the early elimination of local variables, can have a significant effect.

Acknowledgements

We wish to thank Peter Stuckey for suggesting the use of BDDs for computing models of Datalog programs, and for other related discussions. John Whaley provided great assistance with the `bddbddb` tool. We also thank the partners in the ASAP project for discussions and feedback on related topics. An abstract presenting the determinisation algorithm was presented at the NSAD Workshop in Paris, January 2005, and useful comments were received from Laurent Mauborgne and other attendees at the workshop. The ICLP referees gave valuable suggestions for improving the paper.

References

1. Gallagher, J.P., Boulanger, D., Sağlam, H.: Practical model-based static analysis for definite logic programs. In Lloyd, J.W., ed.: Proc. of International Logic Programming Symposium, MIT Press (1995) 351–365
2. Gallagher, J.P., Henriksen, K.S.: Abstract domains based on regular types. In Lifschitz, V., Demoen, B., eds.: Proceedings of the International Conference on Logic Programming (ICLP'2004). Volume 3132 of Springer-Verlag Lecture Notes in Computer Science. (2004) 27–42
3. Craig, S., Gallagher, J.P., Leuschel, M., Henriksen, K.S.: Fully automatic binding time analysis for Prolog. In Etalle, S., ed.: Pre-Proceedings, 14th International Workshop on Logic-Based Program Synthesis and Transformation, LOPSTR 2004, Verona, August 2004. (2004) 61–70
4. Lloyd, J.: Foundations of Logic Programming: 2nd Edition. Springer-Verlag (1987)
5. Boulanger, D., Bruynooghe, M., Denecker, M.: Abstracting *s*-semantics using a model-theoretic approach. In Hermenegildo, M., Penjam, J., eds.: Proc. 6th International Symposium on Programming Language Implementation and Logic Programming, PLILP'94. Volume 844 of Springer-Verlag Lecture Notes in Computer Science. (1994) 432–446

6. Boulanger, D., Bruynooghe, M.: A systematic construction of abstract domains. In Le Charlier, B., ed.: Proc. First International Static Analysis Symposium, SAS'94. Volume 864 of Springer-Verlag Lecture Notes in Computer Science. (1994) 61–77
7. Corsini, M.M., Musumbu, K., Rauzy, A., Le Charlier, B.: Efficient bottom-up abstract interpretation of prolog by means of constraint solving over symbolic finite domains. In Bruynooghe, M., Penjam, J., eds.: Programming Language Implementation and Logic Programming, 5th International Symposium, PLILP'93. Volume 714 of Springer-Verlag Lecture Notes in Computer Science. (1994) 75 – 91
8. Codish, M., Demoen, B.: Analysing logic programs using "Prop"-ositional logic programs and a magic wand. In Miller, D., ed.: Proceedings of the 1993 International Symposium on Logic Programming, Vancouver, MIT Press (1993)
9. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of the 4th ACM Symposium on Principles of Programming Languages, Los Angeles. (1977) 238–252
10. Comon, H., Dauchet, M., Gilleron, R., Jacquemard, F., Lugiez, D., Tison, S., Tommasi, M.: Tree Automata Techniques and Applications. http://www.grappa.univ-lille3.fr/tata (1999)
11. Ullman, J.: Principles of Knowledge and Database Systems; Volume 1. Computer Science Press (1988)
12. Dawson, S., Ramakrishnan, C.R., Warren, D.S.: Practical program analysis using general purpose logic programming systemsa case study. In: Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation. (May 1996) 17–126
13. Whaley, J., Lam, M.S.: Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In Pugh, W., Chambers, C., eds.: PLDI, ACM (2004) 131–144
14. Whaley, J., Unkel, C., Lam, M.S.: A bdd-based deductive database for program analysis (2004) `http://bddbddb.sourceforge.net/`.
15. Lind-Nielsen, J.: BuDDy, a binary decision diagram package (2004) `http://sourceforge.net/projects/buddy`.
16. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press (2000)
17. Howe, J.M., King, A.: Positive Boolean Functions as Multiheaded Clauses. In Codognet, P., ed.: International Conference on Logic Programming. Volume 2237 of LNCS. (2001) 120–134
18. Gallagher, J.P., Puebla, G.: Abstract Interpretation over Non-Deterministic Finite Tree Automata for Set-Based Analysis of Logic Programs. In: Fourth International Symposium on Practical Aspects of Declarative Languages (PADL'02). LNCS (2002)
19. Genet, T., Tong, V.V.T.: Reachability analysis of term rewriting systems with Timbuk. In Nieuwenhuis, R., Voronkov, A., eds.: LPAR. Volume 2250 of Lecture Notes in Computer Science., Springer (2001) 695–706
20. Marriott, K., Søndergaard, H.: Bottom-up abstract interpretation of logic programs. In: Proceedings of the Fifth International Conference and Symposium on Logic Programming, Washington. (1988)
21. Schachte, P.: Precise and Efficient Static Analysis of Logic Programs. PhD thesis, Dept. of Computer Science, The University of Melbourne, Australia (1999)
22. Banda, G.: Scalable real-time kernel for small embedded systems. Master's thesis, Southern Univ. of Denmark, Sønderborg (2003)

23. Charatonik, W., Podelski, A.: Set-based analysis of reactive infinite-state systems. In Steffen, B., ed.: 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)
24. Goubault-Larrecq, J.: A method for automatic cryptographic protocol verification. In Rolim, J.D.P., ed.: 15 IPDPS 2000 Workshops, Cancun, Mexico, May 1-5, 2000, Proceedings. Volume 1800 of Springer-Verlag Lecture Notes in Computer Science., Springer (2000) 977–984
25. Monniaux, D.: Abstracting cryptographic protocols with tree automata. Sci. Comput. Program. **47(2-3)** (2003) 177–202
26. Comon, H., Kozen, D., Seidl, H., Vardi, M.: Applications of Tree Automata in Rewriting, Logic and Programming. Schloß Dagstuhl Seminar 9743, `http://www.informatik.uni-trier.de/~seidl/Trees.html` (October 20-24, 1997)
27. Heintze, N.: Using bottom-up tree automaton to solve definite set constraints. Unpublished. Presentation at Schloß Dagstuhl Seminar 9743, `http://www.informatik.uni-trier.de/~seidl/Trees.html` (1997)
28. Börstler, J., Möncke, U., Wilhelm, R.: Table compression for tree automata. ACM Trans. Program. Lang. Syst. **13** (1991) 295–314
29. Iwaihara, M., Inoue, Y.: Bottom-up evaluation of logic programs using binary decision diagrams. In Yu, P.S., Chen, A.L.P., eds.: ICDE, IEEE Computer Society (1995) 467–474