Program analysis and specialisation using tree automata*

John P. Gallagher

Computer Science, Building 42.1, P.O. Box 260, Roskilde University, DK-4000 Denmark
Email: jpg@ruc.dk

Abstract. Static analysis of programs using regular tree grammars has been studied for more than 30 years, the earliest example being Reynolds' work on automatic derivation of data-type definitions from untyped functional programs. Recently the topic has attracted renewed attention, with applications in program specialisation, data flow analysis, shape analysis, mode and type inference, termination analysis and infinite state model checking.
There are several related viewpoints on analysis using regular tree grammars, including set constraints, abstract interpretation over tree automata domain, directed types, regular approximation and regular type inference.
The lectures will first summarise the relevant properties of finite tree automata, which provide a common foundation for these different viewpoints. It will then be shown how to construct an abstract interpretation over a domain of finite tree automata. Various program analyses based on this domain will be presented, such as "soft" type construction, checking of safety properties in infinite state systems, and derivation of term-size measures for termination analysis.
The lectures will also cover the construction of static analyses based on a given tree grammar capturing some properties of interest. It will be shown (for logic programs) how to build a precise analysis for a program based on an arbitrary regular tree grammar. This has applications in type and mode inference, binding time analysis for offline partial evaluation, and control of online partial evaluation.

1 Introduction and Definitions

Descriptions of sets of terms (or trees) can capture the "shape" of computational entities such as data structures, computation trees and proof trees. The field of finite tree automata provides fundamental notations and tools for defining and manipulating sets of terms. The aim of an analysis based on tree automata is to infer the approximate structure of such entities, allowing the analysis of

* Notes for the Summer School on Program Analysis and Transformation, DIKU, Copenhagen, June 2005. http://www.diku.dk/PAT2005/

many run-time properties such as descriptive types, termination, resource consumption, synchronisation and reachability of states. Tree automata can also be combined with constraints to increase their expressiveness.

In these notes we cover the following topics, some only briefly.

- The definitions of the main concepts concerning finite tree automata, their syntax and semantics.
- The relationship of finite tree automata to other common notations for specifying sets of terms.
- Key computational aspects such as decision procedures for emptiness, and operations on tree automata, especially determinisation.
- An outline of a static analysis framework for logic programs, in which a program is analysed with respect to a given tree automaton.
- Using tree automata to define program properties.
- Applications of tree automata in offline and online program specialisation.
- Applications in model checking and analysis of infinite-state systems.
- Scalability and complexity. Compact representation of determinised automata. Using BDDs to perform analyses based on tree automata.
- Deriving a tree automaton as an approximation of a program.

1.1 Fundamental Definitions

For the most part we use the notation and terminology from Comon *et al.* [12].

Let Σ be a set of function symbols. Each function symbol in Σ has a rank (arity) which is a natural number. Whenever we write an expression such as $f(t_1, \ldots, t_n)$, we assume that $f \in \Sigma$ and has arity n. We write f^n to indicate that function symbol f has arity n. If the arity of f is 0 we often write the term $f()$ as f and call f a *constant*.

The set of *ground terms* (or *trees*) Term_Σ associated with Σ is the least set containing the constants and all terms $f(t_1, \ldots, t_n)$ such that $t_1, \ldots, t_n$ are elements of Term_Σ and $f \in \Sigma$ has arity n.

Finite tree automata provide a means of finitely specifying possibly infinite sets of ground terms, just as finite automata specify sets of strings.

Definition 1. *A finite tree automaton (FTA) is 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 $f(q_1, \ldots, q_n) \to q$, where $f \in \Sigma$ and $q, q_1, \ldots, q_n \in Q$.*

FTAs can be "run" on terms in Term_Σ. To define what is meant by running an FTA we introduce *contexts*. Let $\mathcal{X}_n$ be set of n constants $x_1, \ldots, x_n$ not occurring in Σ. A term $t \in \mathsf{Term}_{\Sigma \cup \mathcal{X}_n}$ is *linear* if each x_i occurs at most once in t. A linear term $C \in \mathsf{Term}_{\Sigma \cup \mathcal{X}_n}$ is called a *context*, and the expression $C[t_1, \ldots, t_n]$ denotes the term in Term_Σ obtained by substituting t_i for x_i in C (for $1 \leq i \leq n$).

There is a *top-down move* $t \to t'$ for the FTA $\langle Q, Q_f, \Sigma, \Delta \rangle$, where $t, t' \in \mathsf{Term}_{\Sigma \cup Q}$ (here Q is considered as a set of constants), if and only if there exists

a context $C \in \mathsf{Term}_{\Sigma \cup \mathcal{X}_1}$, a transition $f(q_1, \ldots, q_n) \to q \in \Delta$, $t = C[q]$ and $t' = C[f(q_1, \ldots, q_n)]$. The top-down derivation relation $\xrightarrow{*}$ is the reflexive, transitive closure of $\to$.

Bottom-up moves and derivations are similarly defined, but the arrow is reversed. A term $t \in \mathsf{Term}_\Sigma$ is *accepted* by an FTA $\langle Q, Q_f, \Sigma, \Delta \rangle$ if $q_0 \xrightarrow{*} t$ (where the derivation is top-down) or $t \xrightarrow{*} q_0$ (where the derivation is bottom-up), where $q_0 \in Q_f$. 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.

Tree Automata and Regular Types In order to put this discussion in context, we draw attention immediately to the connection between regular types and tree automata. (We often refer to regular types simply as "types"). A type is simply regards as an accepting state of an automaton. Given an automaton $R = \langle Q, Q_f, \Sigma, \Delta \rangle$, and $q \in Q_f$, define the automaton R_q to be $\langle Q, \{q\}, \Sigma, \Delta \rangle$. The language $L(R_q)$ is the set of terms corresponding to type q. We say that a term *is of type* q, written $t : q$, if and only if $q \in L(R_q)$.

Example 1. In the following examples, let $\Sigma = \{[]^0, [\_|\_]^2, leaf^1, tree^2, 0^0, s^1\}$, and let $Q = \{list, listnat, nat, zero, one, bintree, any, list0, list1, list2\}$. We define the set Δ_{any} to be the following set of transitions.

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

- $Q_f = \{listnat\}$, $\Delta = \{[] \to listnat, [nat|listnat] \to listnat, 0 \to nat, s(nat) \to nat\}$. The type *listnat* is the set of lists of natural numbers in successor notation.
- $Q_f = \{list\}$, $\Delta = \Delta_{any} \cup \{[] \to list, [any|list] \to list\}$. The type *list* is the set of lists of arbitrary terms in Term_Σ.
- $Q_f = \{list2\}$, $\Delta = \{[] \to list0, [one|list0] \to list1, [zero|list1] \to list2, 0 \to zero, s(zero) \to one\}$. The type $list2$ is the set consisting of the single term $[0, s(0)]$.
- $Q_f = \{bintree\}$, $\Delta = \Delta_{any} \cup \{leaf(any) \to bintree, tree(bintree, bintree) \to bintree\}$. The type *bintree* is the set of binary trees whose leaves are any terms in Term_Σ.
- $Q_f = \{list1\}$, $\Delta = \{[] \to list1, [one|list1] \to list1, [zero|list0] \to list1, [] \to list0, [zero|list0] \to list0, 0 \to zero, s(zero) \to one\}$. The type $list1$ is the set of lists consisting of zero or more elements $s(0)$ followed by zero or more elements 0 (such as $[s(0), 0], [s(0), s(0), 0, 0, 0], [0, 0], [s(0)], \ldots$.

Example 2. Take the second automaton in the previous example:

- A bottom-up derivation of $[[0], 0]$ is

$$[[0], 0] \to [[0|any], 0] \to [[any|any], 0] \to [any, 0] \to [any, any] \\ \to [any, any|list] \to [any|list] \to list$$

– A top-down derivation of $[s(0)|0]$ fails after, e.g.

$$list \to [any|list] \to [s(any)|list] \to [s(0)|list]$$

since for any derivation to succeed there would have to be a transition $0 \to list$.

Deterministic and Non-deterministic Tree Automata There are two notions of non-determinism in tree automata: bottom-up and top-down. Both are highly relevant, in different ways, for types.

Definition 2. *A* bottom-up deterministic *finite tree automaton is one in which the set of transitions Δ contains no two transitions with the same left-hand-side.*

It can be shown that (so far as expressiveness is concerned) we can limit our attention to FTAs to *bottom-up deterministic* finite tree automata. For every FTA R there exists a bottom-up deterministic FTA R' such that $L(R) = L(R')$.

Bottom-up deterministic FTAs define disjoint types, and there are applications where it is desirable to obtain disjoint types. The transformation to bottom-up deterministic form can introduce an exponential number of new states, in the worst case. However, it is often useful and practical in the context of types. In Section 5.3 we discuss the determinisation process in more detail.

We will examine the process of determinisation in greater detail later.

Definition 3. *An automaton $R = \langle Q, Q_f, \Sigma, \Delta\rangle$ is called* complete *if it contains a transition $f(q_1, \ldots, q_n) \to q$ for all n-ary functions $f \in \Sigma$ and states $q_1, \ldots, q_n \in Q$.*

We may always extend an FTA $\langle Q, Q_f, \Sigma, \Delta\rangle$ to make it complete, by adding a new state $q^\flat$ to Q. Then add transitions of the form $f(q_1, \ldots, q_n) \to q^\flat$ for every combination of f and states $q_1, \ldots, q_n$ (including $q^\flat$) that does not appear in Δ. Note that a complete bottom-up deterministic finite tree automaton in which every state is an accepting state is one which partitions the set of terms into disjoint subsets (types), one for each state. This follows since in a bottom-up derivation there is exactly one transition that can be applied to each context of form $C[f(q_1, \ldots, q_n)]$. In such an automaton $q^\flat$ can be thought of as the error type, that is, the set of terms not accepted by any other type.

Example 3. Let $\Sigma = \{[]^0, [\_|\_]^2, 0^0\}$, and let $Q = \{list, listlist, any\}$. The set Δ_{any} is defined as before. let $Q_f = \{list, listlist\}$, $\Delta = \Delta_{any} \cup \{[] \to list, [any|list] \to list, [] \to listlist, [list|listlist] \to listlist, [listlist|listlist] \to listlist\}$. The type *list* is the set of lists of any terms, while the type *listlist* is the set of lists whose elements are of type *list* or *listlist*.

The automaton is not bottom-up deterministic; for example, three transitions have the same left-hand-side, namely, $[] \to list$, $[] \to listlist$ and $[] \to any$. So for example the term $[[0]]$ is accepted by $list, listlist$ and any. A determinization algorithm could be applied, yielding the following. Intuitively, we can think of q_1 as the type $any \cap list \cap listlist$, $q2$ as the type $(list \cap any) - listlist$, and q_3

as $any - (list \cup listlist)$. Thus q_1, q_2 and q_3 are disjoint. The automaton is given by $Q = \{q_1, q_2, q_3\}$, Σ as before, $Q_f = \{q_1, q_2\}$ and $\Delta = \{[] \to q_1, [q_1|q_1] \to q_1, [q_2|q_1] \to q_1, [q_1|q_2] \to q_2, [q_2|q_2] \to q_2, [q_3|q_2] \to q_2, [q_3|q_1] \to q_2, [q_2|q_3] \to q_3, [q_1|q_3] \to q_3, [q_3|q_3] \to q_3, 0 \to q_3\}$.

This automaton is also complete.

FTAs can be extended to allow ϵ-transitions, without altering their expressive power. An ϵ-transition is of the form $q \to q'$. Such transitions can be removed from Δ, after adding all transitions $f(q_1, \ldots, q_n) \to q'$ such that there is a transition $f_(q_1, \ldots, q_n) \to q$ in Δ, and there is a chain of ϵ-transitions $q \to \cdots \to q'$. We assume that ϵ-transitions have been removed unless otherwise stated.

A more restrictive kind of deterministic automaton can be defined, which is also highly relevant in the context of types.

Definition 4. *An FTA is* top-down deterministic *if it has no two transitions with both the same right-hand-side and the same function symbol on the left-hand-side (for example $f(q_1, q_2) \to q$ and $f(q_2, q_3) \to q$).*

When constructing a top-down derivation in a top-down deterministic automaton, there is thus at most one transition that can be used to construct a move for each leaf. Thus checking whether $t \in L(R)$ for such an automaton R can be done in $O(|t|)$ steps.

Top-down determinism introduces a loss in expressiveness. It is *not* the case that for each FTA R there is a top-down deterministic FTA R' such that $L(R) = L(R')$. Note that a top-down deterministic automaton can be transformed to an equivalent bottom-up deterministic automaton, as usual, but the result might not be top-down deterministic.

Example 4. Take the final automaton from Example 1. $Q_f = \{list1\}$, $\Delta = \{[] \to list1, [one|list1] \to list1, [zero|list0] \to list1, [] \to list0, [zero|list0] \to list0, 0 \to zero, s(zero) \to one\}$. This is not top-down deterministic, due to the presence of transitions $[one|list1] \to list1, [zero|list0] \to list1$. No top-down deterministic automaton can be defined that has the same language.

Now consider the automaton with transitions $\Delta_{any} \cup \{[] \to list, [any|list] \to list\}$. This is top-down deterministic, but not bottom-up deterministic (since $[] \to list$ and $[] \to any$ both occur). Determinizing (bottom-up) this automaton would result in one that is not top-down deterministic, since we would have disjoint types corresponding to $list$ and $q = any - list$. This would lead to transitions $[q|list] \to list$ and $[list|list] \to list$ which violates top-down non-determinism.

An automaton R which is not equivalent to any top-down deterministic automaton can be approximated by a top-down deterministic automaton R' such that $L(R) \subseteq L(R')$. Obviously we could use $R' = \langle \{any\}, \{any\}, \Sigma, \Delta_{any} \rangle$ since this is top-down deterministic, but we would like to find a top-down deterministic R' having the smallest possible language that includes $L(R)$.

Type systems for typed programming languages tend to insist on top-down deterministic types. In spite of the loss of expressiveness they are perceived as

somehow more "natural". Worst case computational behaviour is unchanged, so top-down determinism is not "simpler".

Operations on Finite Tree Automata Tree automata have a number of desirable properties and operations, which are at the heart of analysis algorithms. Let R, R_1, R_2 be FTAs.

- $t \in L(R)$ is decidable. The time required for the check is linear ($O(|t| + |R|)$ if R is bottom-up deterministic, and $O(|t|)$ if R is top-down deterministic) otherwise it is $O(|t| \times |R|)$.
- $L(R) = \emptyset$ is decidable. We write $\mathsf{empty}(R)$ and $\mathsf{nonempty}(R)$ for $L(R) = \emptyset$ and $L(R) \neq \emptyset$ respectively. Emptiness can be decided in time $O(|R|)$.
- $R_1 \times R_2$: the product automaton is defined as follows. Let $R_1 = \langle Q_1, Q_{f1}, \Sigma, \Delta_1 \rangle$ and $R_2 = \langle Q_2, Q_{f2}, \Sigma, \Delta_2 \rangle$. We assume that $Q_1 \cap Q_2$ is empty, and we can rename states to ensure this if necessary. The *product* automaton $R_1 \times R_2$ is defined as the automaton $\langle Q_1 \times Q_2, Q_{f1} \times Q_{f2}, \Sigma, \Delta_1 \times \Delta_2 \rangle$ where

$$\begin{aligned} \Delta_1 \times \Delta_2 = \{ f((q_1, q_1'), \ldots, (q_n, q_n')) \to (q, q') \mid \\ f(q_1, \ldots, q_n) \to q \in \Delta_1 \\ f(q_1', \ldots, q_n') \to q' \in \Delta_2 \} \end{aligned}$$

  The language accepted by $R_1 \times R_2$ is $L(R_1) \cap L(R_2)$. Note that if R_1 and R_2 are deterministic (either top-down or bottom-up), then so is $R_1 \times R_2$. The number of transitions in the product can often be reduced by generating only product transitions with a right-hand-side (q, q') that is reachable. (q, q') is reachable if and only if it is in $Q_{f1} \times Q_{f2}$, or it occurs on the left-hand-side of a transition whose right-hand-side is reachable. Eliminating unreachable states can often yield a significant reduction especially if R_1 and R_2 are top-down deterministic.
- $R_1 \cup R_2$: the union of automata R_1 and R_2 is obtained straightforwardly. Let $R_1 = \langle Q_1, Q_{f1}, \Sigma, \Delta_1 \rangle$ and $R_2 = \langle Q_2, Q_{f2}, \Sigma, \Delta_2 \rangle$. We assume again that $Q_1 \cap Q_2$ is empty. Then $R_1 \cup R_2 = \langle Q_1 \cup Q_2, Q_{f1} \cup Q_{f2}, \Sigma, \Delta_1 \cup \Delta_2 \rangle$. $L(R_1 \cup R_2) = L(R_1) \cup L(R_2)$. Note that determinacy is not preserved by this union operation. However, we may of course restore bottom-up determinacy by further transformation but there may not be any top-down deterministic automaton equivalent to $R_1 \cup R_2$.
- $\mathsf{complement}(R)$: the complement of R can be formed quite easily when R is a complete bottom-up deterministic automaton. In this case, simply replace the set of final states Q_f by $Q - Q_f$. R can always be converted to a complete bottom-up deterministic automaton, but at the possible cost of an exponential increase in the number of states and transitions. Note that the set of languages definable by top-down deterministic FTAs is not closed under complement.
- $R_1 \preceq R_2$: this holds if $\mathsf{nonempty}(R_1 \cap \mathsf{complement}(R_2))$, and the complexity can be derived from the emptiness check, intersection and complement operations. R_1 and R_2 are equivalent if $R_1 \preceq R_2$ and $R_2 \preceq R_1$.

– $\mathsf{minimize}(R)$: given any FTA R, we can construct another automaton with the same language, having the minimum number of states. The standard algorithm in the literature takes as input a bottom-up deterministic types.

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

1.2 Regular Tree Grammars

Regular tree grammars provide an alternative approach to defining sets of terms, focussing on the language *generated* by a given grammar, rather than the terms *recognised* by an automaton. However, the formal aspects, and the operations on grammars, mirror the corresponding treatment of automata, so we do not need to say much about regular tree grammars apart from noting the syntax and terminology.

Definition 5. *A regular tree grammar is a quadruple $\langle N, S, \Sigma, \Phi\rangle$ consists of a set Σ of terminal symbols, each with an arity, a set of non-terminals N, a distinguished non-terminal S called the start symbol, and a set Φ of productions $L \to R$, where L is a non-terminal and $R \in \mathsf{Term}_{\Sigma\cup N}$. The set of productions with the same left-hand-side L, say $\{L \to R_1, \ldots, L \to R_k\}$, may be written as $L \to R_1 \mid \cdots \mid R_k$.*

A regular tree grammar G generates terms by commencing with the start symbol S, and successively replacing some non-terminal L by R, where there is a production $L \to R$. The set of terms containing no non-terminals obtainable in this way is called the language of G, $L(G)$. Two grammars G_1 and G_2 are equivalent if $L(G_1) = L(G_2)$.

It can be shown that every regular tree grammar G is equivalent to another *normalised* grammar in which all productions have the form $L \to f(R_1, \ldots, R_n)$ $(n \geq 0)$ where $R_1, \ldots, R_n$ are non-terminals. Some new non-terminals might be introduced in the normalisation process.

Correspondence between regular tree grammars and FTAs There are various versions of the syntax of both FTAs and tree grammars; we have deliberately chosen versions to emphasise the close similarity between them. The notions are almost interchangeable from our point of view. A normalised regular tree grammar $\langle N, S, \Sigma, \Phi\rangle$ can be considered as alternative syntax for the FTA $\langle N, \{S\}, \Sigma, \Delta\rangle$, where $\Delta = \{f(q_1, \ldots, q_n) \to q \mid q \to f(q_1, \ldots, q_n) \in \Phi\}$. The FTA arising from a grammar contains only a single accepting state, corresponding to the start symbol of the grammar. The language generated by the grammar is the same as the language accepted by the FTA.

Similarly, given an FTA $\langle Q, Q_f, \Sigma, \Delta\rangle$, we can obtain a (non-normalised) tree grammar $\langle Q \cup \{q_S\}, q_S, \Sigma, \Phi\rangle$, where q_S is a new state not in Q, and the set of productions $\Phi = \{q_S \to q_f \mid q_f \in Q_f\} \cup \{q \to f(q_1, \ldots, q_n) \mid f(q_1, \ldots, q_n) \to q \in \Delta\}$. In this case the language generated by the grammar is the union of the sets of terms accepted by the states in Q_f. The proofs of these equivalences is based

on the close relationship between the generation of terms from the grammar and top-down derivations in the corresponding FTA.

Example 5. Let a regular tree grammar G be defined as

$$G = \langle \{list1, list2\}, list1, \{[]^0, [\_|\_]^2, 0^0, s^1\}, \Phi \rangle$$

where Φ is the following set of productions.

$$\begin{array}{l} list1 \rightarrow [] \mid [s(0)|list1] \mid [0|list0] \\ list0 \rightarrow [] \mid [0|list0] \end{array}$$

$L(G)$ is the set of lists consisting of zero or more elements $s(0)$ followed by zero or more elements 0 (such as $[s(0), 0], [s(0), s(0), 0, 0, 0], [0, 0], [s(0)], \ldots$. It is alternative syntax for the final FTA in Example 1. Normalising G would introduce non-terminals, say *one* and *zero*, to replace the occurrences of $s(0)$ and 0 respectively on the right-hand-sides of the productions above, along with the new productions $one \rightarrow s(zero)$ and $zero \rightarrow 0$.

1.3 Regular Unary Logic Programs

The use of logic programs to represent regular tree languages was introduced by Yardeni and Shapiro [50]. A unary or *monadic* definite logic program is a logic program in which all predicates are unary. A *regular unary logic (RUL)* program is one in which every clause is of the form

$$p(f(X_1, \ldots, X_n)) \leftarrow p_1(X_1), \ldots, p_n(X_n) \quad (n \geq 0)$$

where $X_1, \ldots, X_n$ are distinct variables. If $n = 0$ the clause is a unit clause represented $p(f^0) \leftarrow true$.

An RUL program is alternative syntax for an FTA, and hence for a regular tree grammar. Let P be an RUL program, and let the set of predicates in P be $\mathsf{preds}(P)$. Let Σ be the set of function symbols in the language of P. Define the FTA $\langle \mathsf{preds}(P), \mathsf{preds}(P), \Sigma, \Delta \rangle$ where $\Delta = \{f(p_1, \ldots, p_n) \rightarrow p \mid p(f(X_1, \ldots, X_n)) \leftarrow p_1(X_1), \ldots, p_n(X_n) \in P\}$.

The semantics of logic programs can be defined operationally, through (say) SLD-derivations, or denotationally, using models. The *success set* of a program P is the set of ground atomic formulas $p(t)$ such that $P \cup \{\leftarrow p(t)\}$ has an SLD-refutation [36], in other words, the ground goal $p(t)$ succeeds in P. An SLD-refutation of $\{\leftarrow p(t)\}$ in an RUL program mirrors a top-down derivation $p \stackrel{*}{\rightarrow} t$ in the FTA. Hence the success set of an RUL program captures the language recognised by the corresponding automaton.

The usual denotational semantics of a definite logic program P is given as the least Herbrand model of P. There are various ways to construct the least Herbrand model, but the most significant one from our point of view is as the least fixpoint of the *immediate consequence* function T_P. Write $c \in \mathsf{gnd}_\Sigma(P)$

to mean that c is a ground instance of some clause in P (that is, each variable of c is substituted by an element of Term_Σ). Let $I \subseteq \{p(t_1, \ldots, t_n) \mid p \in \mathsf{preds}(P), t_1, \ldots, t_n \in \mathsf{Term}_\Sigma\}$.

$$T_P(I) = \{H \mid \{B_1, \ldots, B_n\} \subseteq I, H \leftarrow B_1, \ldots, B_n \in \mathsf{gnd}_\Sigma(P)\}$$

The least fixed point of T_P, denoted $\mathsf{lfp}(T_P)$ exists and can be computed as $\bigcup\{T_P^n(\emptyset) \mid n > 0\}$. It is a standard result of logic program semantics that the success set of a logic program P is identical to $\mathsf{lfp}(T_P)$. Thus the language defined by an RUL program has a fixpoint definition. Similar fixpoint characterisations could be given for the languages of FTAs and regular tree grammars.

The RUL program formulation of regular tree languages has another dimension, since the class of RUL programs is equivalent in its expressive power to a wider class of logic programs called *proper unary* logic programs [17]. A proper unary logic program contains clauses $p(t) \leftarrow p_1(t_1), \ldots, p_n(t_n)$ such that t does not contain repeated variables, and for each t_i, $1 \leq i \leq n$, one of the following holds.

1. t_i is a subterm of t;
2. t_i has no variable in common with t;
3. t is a strict subterm of t_i, and no variable occurring in t occurs in t_i other than as a variable in the subterm t.

It was shown that every proper unary logic program P is equivalent to some RUL program P' (equivalence meaning that $\mathsf{preds}(P) \subseteq \mathsf{preds}(P')$ and their success sets, restricted to $\mathsf{preds}(P)$, are the same). Furthermore, an algorithm exists [17] which converts a proper unary logic program to a equivalent RUL program. This is significant for deriving descriptive types (see Section 6).

A further dimension to the RUL notation for FTAs is the possibility of adding constraints to transitions. Consider the clause

$$p(f(X_1, \ldots, X_n)) \leftarrow t_1(X_1), \ldots, t_n(X_n), c(X_1, \ldots, X_n)$$

where $c(X_1, \ldots, X_n)$ is a constraint over a domain with a decision procedure and a projection operation. For some classes of constraint, closure under intersection, union and complementation, and decidability properties such as emptiness, are retained under this extension. This provides an approach to extending the expressiveness of FTAs [12], [43].

1.4 Definite Set Constraints

A different approach to defining regular sets of terms was developed by Heintze and Jaffar, building on earlier work by Reynolds, Jones and Muchnick, and Mishra. This approach defines a set of *constraints* whose variables denote sets of terms. Given a set of such constraints, one seeks a solved form. The solved form for significant classes of set constraints, it turns out, is very similar to the form of transitions in FTAs or productions in regular grammars.

An initial intuition about the relation of set constraints to regular tree grammars and hence to FTAs can be obtained by considering the productions of a regular tree grammar as constraints. Let $\langle N, S, \Sigma, \Phi \rangle$ be a regular tree grammar. Introduce a set of variables $\mathcal{V} = \{X_n \mid n \in N\}$. The variables range over sets of terms in Term_Σ. An interpretation of the variables is a mapping $I : \mathcal{V} \to 2^{\mathsf{Term}_\Sigma}$. Given an interpretation I, we can extend it to $\mathsf{Term}_{\Sigma \cup \mathcal{V}}$, where $I(f(u_1, \ldots, u_n)) = \{f(t_1, \ldots, t_n) \mid t_i \in I(u_i), 1 \leq i \leq n\}$.

Then the set of productions Φ is read as the conjunction of constraints $\{X_L \supseteq R' \mid L \to R \in \Phi\}$, where R' is obtained from R by replacing each non-terminal by the corresponding variable. If we write a production as $L \to R_1 \mid \cdots \mid R_k$, it gives rise to the constraint $L \supseteq (R'_1 \cup \cdots \cup R'_k)$

Example 6. Consider the regular tree grammar in Example 5. It corresponds to the constraints

$$\begin{array}{l}(X_{list1} \supseteq ([] \ \cup \ [s(0)|X_{list1}] \ \cup \ [0|X_{list0}])) \ \wedge \\ (X_{list0} \supseteq ([] \ \cup \ [0|X_{list0}]))\end{array}$$

We seek a solution, or better, the least solution, to the set of constraints. (The interpretation which maps both X_{list1} and X_{list0} to Term_Σ is a solution, but not a very interesting one). In the case of constraints arising from a tree grammar G, it can be shown that the least solution assigns $L(G)$ to X_S, where S is the start symbol of the grammar.

The main problem of set constraints is to consider a wider class of constraints than the ones just considered, and develop algorithms for finding solutions for a given conjunction of constraints.

Various presentations of set constraints have been given; given a set of ranked function symbols Σ and a set of variables $\mathcal{V}$ a set expression (for our purposes) is of the form X, $\bot$, $f(e_1, \ldots, e_n)$, $f_i^{-1}(e_1, \ldots, e_n)$, $e_1 \cup e_2$, or $e_1 \cap e_2$, where $X \in \mathcal{V}$ and $e_1, e_2, \ldots, e_n$ are set expressions. A definite set constraint is of the form $X \supseteq e$ where $X \in \mathcal{V}$ and e is a set expression. If the constraint is $X \supseteq e$ where e has the form $f(X_1, \ldots, X_n)$, $(n \geq 0, X_i \in \mathcal{V}, 1 \leq i \leq n)$, it is in regular form. A set of set constraints in regular form is clearly alternative syntax for an FTA or regular tree grammar.

A conjunction of definite set constraints has a least solution which is a tuple of regular sets of terms (one for each variable occurring in the set of constraints). We can, of course, represent the solution as a tuple of tree grammars, a tuple of FTAs, or a set of set constraints in regular form. A number of algorithms for solving a conjunction of set constraints have been proposed.

1.5 Tree Languages: Summary

The main concept underlying type languages is that of a regular set of terms. This can be defined by several closely related formalisms, such as FTAs, regular tree grammars, RUL programs, and regular form set constraints. To these we could add type graphs [31, 47] which are data structures for tree grammars, having a graphical representation.

There is a number of useful operations, decidable properties and closure properties, available in all the formalisms. Restrictions, especially top-down determinism, play an important role in expressiveness of the formalisms and efficiency of the operations.

Another dimension, displayed by unary logic programs and set constraints, is the transformation of a wider class of expressions into regular form. Proper unary logic programs can be transformed to equivalent RUL programs, and definite set constraints can be transformed to equivalent regular set constraints. These procedures play a role in some of the algorithms for deriving descriptive types from programs.

2 Abstractions Based on Determinised Tree Automata

2.1 Analysis Based on Pre-Interpretations

We now define an analysis framework for logic programs. Bottom-up declarative semantics captures the set of logical consequences (or a model) of a program. The standard, or concrete semantics is based on the Herbrand pre-interpretation. The theoretical basis of this approach to static analysis of definite logic programs was set out in [4, 3] and [18]. We follow standard notation for logic programs [36].

Let P be a definite program and Σ the signature of its underlying language L. 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 ($n \geq 0$).

Correspondence of FTAs and Pre-Interpretations A pre-interpretation with a finite domain D over a signature Σ is equivalent to a complete bottom-up deterministic FTA over the same signature, as follows.

1. The domain D is the set of states of the FTA.
2. Let $\hat{f}$ be the function $D^n \to D$ assigned to $f \in \Sigma$ by the pre-interpretation. In the corresponding FTA there is a set of transitions $f(d_1, \ldots, d_n) \to d$, for each $d_1, \ldots, d_n, d$ such that $\hat{f}(d_1, \ldots, d_n) = d$. Conversely the transitions of a complete bottom-up deterministic FTA define a function [12].

Semantics parameterized by a pre-interpretation We quote some definitions from Chapter 1 of [36]. Let J be a pre-interpretation of L with domain D. Let V be a mapping assigning each variable in L to an element of D. A *term assignment* $T_J^V(t)$ is defined for each term t as follows:

1. $T_J^V(x) = V(x)$ for each variable x.
2. $T_J^V(f(t_1, \ldots, t_n)) = f'(T_J^V(t_1), \ldots, T_J^V(t_n)),\ (n \geq 0)$ for each non-variable term
   $f(t_1, \ldots, t_n)$, where f' is the function assigned by J to f.

Let J be a pre-interpretation of a language L, with domain D, and let p be an n-ary function symbol from L. Then a *domain atom* for J is any atom $p(d_1, \ldots, d_n)$ where $d_i \in D$, $1 \leq i \leq n$. Let $p(t_1, \ldots, t_n)$ be an atom. Then a *domain instance* of $p(t_1, \ldots, t_n)$ with respect to J and V is a domain atom $p(T_J^V(t_1), \ldots, T_J^V(t_n))$. Denote by $[A]_J$ the set of all domain instances of A with respect to J and some V.

The definition of domain instance extends naturally to formulas. In particular, let C be a clause. Denote by $[C]_J$ the set of all domain instances of the clause with respect to J.

Core bottom-up semantics function T_P^J The core bottom-up declarative semantics is parameterised by a pre-interpretation of the language of the program. Let P be a definite program, and J a pre-interpretation of the language of P. Let $Atom_J$ be the set of domain atoms with respect to J. The function $T_P^J : 2^{Atom_J} \to 2^{Atom_J}$ is defined as follows.

$$T_P^J(I) = \left\{ A' \;\middle|\; \begin{array}{l} A \leftarrow B_1, \ldots, B_n \in P \\ A' \leftarrow B'_1, \ldots, B'_n \in [A \leftarrow B_1, \ldots, B_n]_J \\ \{B'_1, \ldots, B'_n\} \subseteq I \end{array} \right\}$$

$\mathsf{M}^J[\![P]\!] = \mathsf{lfp}(T_P^J)$: $\mathsf{M}^J[\![P]\!]$ is the minimal model of P with pre-interpretation J.

Concrete Semantics The usual semantics is obtained by taking J to be the Herbrand pre-interpretation, which we call H. Thus $Atom_H$ is the Herbrand base of (the language of) P and $\mathsf{M}^H[\![P]\!]$ is the minimal Herbrand model of P.

The minimal Herbrand model consists of ground atoms. In order to capture information about the occurrence of variables, we extend the signature with an infinite set of extra constants $\mathcal{V} = \{v_0, v_1, v_2, \ldots\}$. The Herbrand pre-interpretation over the extended language is called HV. The model $\mathsf{M}^{HV}[\![P]\!]$ is our concrete semantics.

The elements of $\mathcal{V}$ do not occur in the program or goals, but can appear in atoms in the minimal model $\mathsf{M}^{HV}[\![P]\!]$. Let $\mathcal{C}(P)$ be the set of all atomic logical consequences of the program P, known as the Clark semantics [9]; that is, $\mathcal{C} = \{A \mid P \models \forall A\}$, where A is an atom. Then $\mathsf{M}^{HV}[\![P]\!]$ is isomorphic to $\mathcal{C}(P)$. More precisely, let Ω be some fixed bijective mapping from $\mathcal{V}$ to the variables in L. Let A be an atom; denote by $\Omega(A)$ the result of replacing any constant v_j in A by $\Omega(v_j)$. Then $A \in \mathsf{M}^{HV}[\![P]\!]$ iff $P \models \forall(\Omega(A))$. By taking the Clark semantics as our concrete semantics, we can construct abstractions capturing the occurrence of variables. This version of the concrete semantics is essentially the same as the one discussed in [18].

In our applications, we will always use pre-interpretations that map all elements of $\mathcal{V}$ onto the same domain element, say d_v. In effect, we do not distinguish between different variables. Thus, a pre-interpretation includes an infinite mapping $\{v_0 \mapsto d_v, v_1 \mapsto d_v, \ldots\}$. For such interpretations, we can take a simpler concrete semantics, in which the set of extra constants $\mathcal{V}$ contains just one

constant v instead of an infinite set of constants. Then pre-interpretations are defined which include a single mapping $\{v \mapsto d_v\}$ to interpret the extra constant.

Abstract Interpretations Let P be a program and J be a pre-interpretation. Let $Atom_J$ be the set of domain atoms with respect to J. The *concretisation function* $\gamma : 2^{Atom_J} \to 2^{Atom_{HV}}$ is defined as $\gamma(S) = \{\ A \mid [A]_J \subseteq S\ \}$

$\mathsf{M}^J[\![P]\!]$ is an abstraction of the atomic logical consequences of P, in the following sense.

Proposition 1. *Let P be a program with signature Σ, and $\mathcal{V}$ be a set of constants not in Σ (where $\mathcal{V}$ can be either infinite or finite). Let HV be the Herbrand interpretation over $\Sigma \cup \mathcal{V}$ and J be any pre-interpretation of $\Sigma \cup \mathcal{V}$. Then $\mathsf{M}^{HV}[\![P]\!] \subseteq \gamma(\mathsf{M}^J[\![P]\!])$.*

Thus, by defining pre-interpretations and computing the corresponding least model, we obtain safe approximations of the concrete semantics.

Condensing Domains The property of being a *condensing* domain [37] has to do with precision of goal-dependent and goal-independent analyses (top-down and bottom-up) over that domain. Goal-independent analysis over a condensing domain loses no precision compared with goal-dependent analysis; this has advantages since a single goal-independent analysis can be reused to analyse different goals (relatively efficiently) with the same precision as if the individual goals were analysed.

The abstract domain is 2^{Atom_J}, namely, sets of abstract atoms with respect to the domain of the pre-interpretation J, with set union as the upper bound operator. The conditions satisfied by a condensing domain are usually stated in terms of the abstract unification operation (namely that it should be idempotent and commutative) and the upper bound $\sqcup$ on the domain (which should satisfy the property $\gamma(X \sqcup Y) = \gamma(X) \cup \gamma(Y)$). The latter condition is clearly satisfied ($\sqcup = \cup$) in our domain). Abstract unification is not explicitly present in our framework. However, we argue informally that the declarative equivalent is the abstraction of the equality predicate $X = Y$. This is the set $\{d = d \mid d \in D_J\}$ where D_J is the domain of the pre-interpretation. This satisfies an idempotency property, since for example the clause $p(X, Y) \leftarrow X = Y, X = Y$ gives the same result as $p(X, Y) \leftarrow X = Y$. It also satisfies a relevant commutativity property, namely that the solution to the goal $q(X, Y), X = Y$ is the same as the solution to $q(X, Y)$, where each clause $q(X, Y) \leftarrow B$ is replaced by $q(X, Y) \leftarrow X = Y, B$. These are informal arguments, but we also note that the goal-independent analysis yields the *least*, that is, the most precise, model for the given pre-interpretation, which provides support for our claim that domains based on pre-interpretations are condensing.

3 Tree Automata for Binding-Time Analysis

Offline partial evaluation techniques rely on an annotated version of the source program to control the specialisation process. These annotations consist of *bind-*

ing types along with control indicators, which guide the specialisation and ensure the termination of the partial evaluation.

A binding type indicates something about the structure of an argument at specialisation time. The basic binding types are usually known as *static* and *dynamic* defined as follows.

- **static**: The argument is definitely known at specialisation time;
- **dynamic**: The argument is possibly unknown at specialisation time.

We will see that these binding types, and also more precise binding types can be defined by means of regular type declarations.

For example, an interpreter may use an environment that is a partially static data structure at partial evaluation time. To model the environment, e.g., as a list of static names mapped to dynamic variables we would use the following definition:

```
:- type binding = static / dynamic.
:- type list_env = []   |   [binding | list_env].
```

We discuss the definitions of `static` and `dynamic` in the next section.

We present an type-based abstract interpretation for propagating the binding types. This algorithm has been implemented as part of the LOGEN partial evaluation system.

Definition of Modes as Regular Types Instantiation modes can be coded as regular types. In other words, we claim that *modes are regular types*, and that this gives some new insight into the relation between modes and types. The set of ground terms over a given signature, for example, can be described using regular types, as can the set of non-ground terms, the set of variables, and the set of non-variable terms. Assume the signature $\Sigma = \{[], [\_|\_], s, 0\}$ with the usual arities, though clearly the definitions can be constructed for any signature. The definition of the types *static* and *dynamic* over $\Sigma \cup \{v\}$ are $static = 0 \mid [] \mid [static|static] \mid s(static)$ and $dynamic = v \mid 0 \mid [] \mid [static|static] \mid s(static)$ respectively. We can add the binding types representing $variable$ (var) and $non-variable$ ($nonvar$) as $var = v$ and $nonvar = 0 \mid [] \mid [dynamic|dynamic] \mid s(dynamic)$

Input states	Output states	Corresponding modes
g, var, any	{any,g}, {any,var}, {any}	ground, variable, non-ground-non-variable
g, any	{any,g}, {any}	ground, non-ground
var, any	{any,var}, {any}	variable, non-variable

Fig. 1. Mode pre-interpretations obtained from g, var and any

Using the determinization algorithm, we can derive other modes automatically. Abbreviating *static* as g (for ground), *dynamic* as *any*, Table 1 shows some disjoint modes obtained by determinisation of different combinations.

Different pre-interpretations are obtained by taking one or both of the modes g and var along with the type any, and then determinizing. The choices are summarised in Figure 1. We do not show the transitions, due to lack of space. To give one example, the mode *non-variable* in the determinized FTA computed from var and any is given by the transitions for $\{any\}$.

$$\{any\} = 0 \mid [] \mid [\{any\}|\{any\}] \mid [\{any, var\}|\{any\}] \mid [\{any\}|\{any, var\}] \mid [\{any, var\}|\{any, var\}] \mid s(\{any\}) \mid s(\{any, var\})$$

Let P be the naive reverse program shown below.

$$rev([], []). \qquad rev([X|U], W) \leftarrow rev(U, V), app(V, [X], W).$$
$$app([], Y, Y). \qquad app([X|U], V, [X|W]) \leftarrow app(U, V, W).$$

The result of computing the least model of P is summarised in Figure 2, with the abbreviations ground=g, variable=v, non-ground=ng, non-variable=nv and non-ground-non-variable=$ngnv$. An atom containing a variable X in the abstract model is an abbreviation for the collection of atoms obtained by replacing X by any element of the abstract domain. The analysis based on g and any is

Input types	Model
g, v, any	$\{rev(g,g), rev(ngnv, ngnv), app(g, var, ngnv), app(g, var, var),$ $app(g,g,g), app(g, ngnv, ngnv), app(ngnv, X, ngnv)\}$
g, any	$\{rev(g,g), rev(ng, ng), app(g, X, X), app(ng, X, ng)\}$
var, any	$\{rev(nv, nv), app(nv, X, X), app(nv, X, nv)\}$

Fig. 2. Abstract Models of Naive Reverse program

equivalent to the well-known Pos abstract domain [37], while that based on g, var and any is the **fgi** domain discussed in [18]. The presence of var in an argument indicates possible freeness, or alternatively, the absence of var indicates definite non-freeness. For example, the answers for rev are definitely not free, the first argument of app is not free, and if the second argument of app is not free then neither is the third.

Combining Modes with Other Types Consider the usual definition of lists, namely $list = []; [any|list]$. Now compute the pre-interpretation derived from the types $list$, any and g. Note that $list$, any and g intersect. The set of disjoint types is $\{\{any, ground\}, \{any, list\}, \{any, ground, list\}, \{any\}\}$ (abbreviated as $\{g, ngl, gl, ngnl\}$ corresponding to ground non-lists, non-ground lists, ground lists, and non-ground-non-lists respectively). The abstract model with respect to the pre-interpretation is

$\{rev(gl, gl), rev(ngl, ngl),$
$app(gl, X, X), app(ngl, ngnl, ngnl), app(ngl, gl, ngl), app(ngl, ngl, ngl)\}$

A given set of user types can be determinized together with types representing *static*, *dynamic* (that is, g and *any*) and *var*. *Call types* can be computed from the abstract model over the resulting pre-interpretation, for example using a query-answer transformation (magic sets). This is a standard approach to deriving call patterns; [11] gives a clear account and implementation strategy.

Let P be the following program for transposing a matrix.

$$
\begin{array}{ll}
transpose(Xs, []) \leftarrow & makerow([], [], []). \\
\quad nullrows(Xs). & makerow([[X|Xs]|Ys], [X|Xs1], [Xs|Zs]) \leftarrow \\
transpose(Xs, [Y|Ys]) \leftarrow & \quad makerow(Ys, Xs1, Zs). \\
\quad makerow(Xs, Y, Zs), & nullrows([]). \\
\quad transpose(Zs, Ys). & nullrows([[]|Ns]) \leftarrow nullrows(Ns).
\end{array}
$$

Let *row* and *matrix* be defined as $row = [] \mid [any|row]$ and $matrix = [] \mid [row|matrix]$ respectively. These are combined with the standard types g, *var* and *any*. Given an initial call of the form $transpose(matrix, any)$, BTA with respect to the disjoint types results in the information that every call to the predicates *makerow* and *transpose* has a matrix as first argument. More specifically, it is derived to have a type $\{any, matrix, row, g\}$ or $\{any, matrix, row\}$, meaning that it is either a ground or non-ground matrix. Note that any term of type *matrix* is also of type *row*. This BTA is optimal for this set of types.

4 Analysis of Program Properties Expressed as Tree Automata

Finite tree automata can capture properties of interest in a computational system (represented below as Horn clauses). In the examples below, the properties are then determinised and used to construct an analysis domain.

Infinite-State Model Checking The following example is from [42].

$$
\begin{array}{lll}
gen([0,1]). & trans1([0,1|T],[1,0|T]). & trans(X,Y) \leftarrow \\
gen([0|X]) \leftarrow gen(X). & trans1([H|T],[H|T1]) \leftarrow & \quad trans1(X,Y). \\
reachable(X) \leftarrow & \quad trans1(T,T1). & trans([1|X],[0|Y]) \leftarrow \\
\quad gen(X). & trans2([0],[1]). & \quad trans2(T,T1). \\
reachable(X) \leftarrow & trans2([H|T],[H|T1]) \leftarrow & \\
\quad reachable(Y), trans(Y,X). & \quad trans2(X,Y). &
\end{array}
$$

It is a simple model of a token ring transition system. A state of the system is a list of processes indicated by 0 and 1 where a 0 indicates a waiting process and a 1 indicates an active process. The initial state is defined by the predicate *gen* and the the predicate *reachable* defines the reachable states with respect to the transition predicate *trans*. The required property is that exactly one process is active in any state. The state space is infinite, since the number of processes (the length of the lists) is unbounded. Hence finite model checking techniques do not

suffice. The example was used in [6] to illustrate directional type inference for infinite-state model checking.

We define simple regular types defining the states. The set of "good" states in which there is exactly one 1 is *goodlist*. The type *zerolist* is the set of list of zeros. (Note that it is not necessary to give an explicit definition of a "bad" state).

$$\begin{array}{ll} one = 1 & goodlist = [zero|goodlist] \mid \ [one|zerolist] \\ zero = 0 & zerolist = [] \mid \ [zero|zerolist] \end{array}$$

Determinization of the given types along with *any* results in five states representing disjoint types: $\{any, one\}$, $\{any, zero\}$, the good lists $\{any, goodlist\}$, the lists of zeros $\{any, zerolist\}$ and all other terms $\{any\}$. We abbreviate these as *one*, *zero*, *goodlist*, *zerolist* and *other* respectively. The least model of the above program over this domain is as follows.

$gen(goodlist)$	$trans1(goodlist, goodlist), trans1(other, other)$
$trans2(other, other)$	$trans(goodlist, goodlist), trans(other, other)$
$trans2(goodlist, other)$	$reachable(goodlist)$
$trans2(goodlist, goodlist)$	

The key property of the model is the presence of $reachable(goodlist)$ (and the absence of other atoms for *reachable*), indicating that if a state is reachable then it is a *goodlist*. Note that the transitions will handle *other* states, but in the context in which they are invoked, only *goodlist* states are propagated. In contrast to the use of set constraints or directional type inference to solve this problem, no goal-directed analysis is necessary. Thus there is no need to define an "unsafe" state and show that it is unreachable.

In summary, the examples show that accurate mode analysis can be performed, and that modes can be combined with arbitrary user defined types. Types can be used to prove properties expressible by regular types. Note that no assumption needs to be made that programs are well-typed; the programmer does not have to associate types with particular argument positions.

Further examples of the use of tree automata for expressing and analysing properties of cryptographic protocols come from [24], [25] and [39].

5 Complexity and Scalability

5.1 Abstract Compilation of a Pre-Interpretation

The idea of abstract compilation was introduced first by Debray and Warren [16]. Operations on the abstract domain are coded as logic programs and added directly to the target program, which is then executed according to standard concrete semantics. The reason for this technique is to avoid some of the overhead of interpreting the abstract operations. The implementation method is to transform the program first, introducing equalities until every non-variable appears as the right-hand-side of an equality, and no nested functions occur. Secondly the program is transformed to an *abstract domain program* by interpreting these

equalities as the pre-interpretation function. 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 $\rightarrow$: $\{[] \rightarrow g, [g|g] \rightarrow g, [g|ng] \rightarrow ng, [ng|g] \rightarrow ng, [ng|ng] \rightarrow ng\}$. The least model of the transformed program together with the facts defining the pre-interpretation is then computed.

When a specific pre-interpretation J is added to $\bar{P}$, the result is a *domain program* for J, called $\bar{P}^J$. Clearly $\bar{P}^J$ has a different language than P, since the definition of $\rightarrow /2$ contains elements of the domain of interpretation. It can easily be shown that least model $\mathsf{M}^J[\![P]\!] = \mathsf{lfp}(T_P^J)$ is obtained by computing $\mathsf{lfp}(T_{\bar{P}^J})$, and then restricting to the predicates in P (that is, omitting the predicate $\rightarrow /2$ which was introduced in the abstract compilation). An example of the domain program for *append* and the pre-interpretation for variable/non-variable is shown below. (Note that don't care arguments are used in the definition of $\rightarrow /2$).

$app(U,Y,Y) \leftarrow \quad [] \rightarrow U.$ $\qquad$ $app(U,Y,V) \leftarrow app(X,Y,Z), [X|X] \rightarrow U, [X|Z] \rightarrow V.$
$v \rightarrow var. \quad [] \rightarrow nonvar.$ $\qquad$ $[\_|\_] \rightarrow nonvar.$

5.2 Efficient Computation of the Least Domain Model

The computation of the least model is an iterative fixpoint algorithm. The iterations of the basic fixpoint algorithm, which terminates when a fixed point is found, can be decomposed into a sequence of smaller fixpoint computations, one for each *strongly connected component (SCC)* of the program's predicate dependency graph. These can be computed in linear time [44]. In addition to the SCC optimisation, our implementation incorporates a variant of the *semi-naive* optimisation [46], which makes use of the information about new results on each iteration. A clause body containing predicates whose models have not changed on some iteration need not be processed on the next iteration.

The essential task in performing an analysis using finite pre-interpretations (complete deterministic tree automata) can be seen as computing the minimal model of a (definite) Datalog program [45], that is, a definite logic program containing no function symbols with non-zero arity. Such programs have finite models. Although there appear to be function symbols in the definition of the pre-interpretation rules $f(d_1, \ldots, d_n) \rightarrow d$, we can easily represent the rules 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 program, we are left with a Datalog program.

Efficient techniques for computing Datalog models have been studied extensively in research on deductive database systems [45], 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 [14].

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 [49].

Our previous experiments [20] 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 [20] 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 [48]. 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 [35]. 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 was exploited in model-checking [10], and in the logic programming context by [30]. 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 [29].

5.3 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. 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 7. Let $\Sigma = \{[]^0, [\_|\_]^2, 0^0\}$, and let $Q = \{list, listlist, any\}$. We define the set Δ_{any} in the usual way for Σ. Let $Q_f = \{list, listlist\}$, $\Delta = \{[] \to list, [any|list] \to list, [] \to listlist, [list|listlist] \to listlist\} \cup \Delta_{any}$. The transitions of the DFTA generated for this NFTA 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 $[\_|\_]^2$ in the full DFTA. There are other equivalent sets of product transitions, for example, $\Delta' = \{[] \to q_1, 0 \to q_3, [\{q_1, q_2\}|\{q_3\}] \to q_3, [\{q_1, q_2\}|\{q_2\}] \to q_2, [\{q_1, q_2\}|\{q_1\}] \to q_1, [\{q_3\}|\{q_3\}] \to q_3, [\{q_3\}|\{q_1\}] \to q_1, [\{q_3\}|\{q_2\}] \to q_2\}$.

A Determinisation Algorithm Generating Product Form The algorithm outlined in this section was based initially on the classical text-book algorithm [12]. 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.

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

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

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

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

The qmap function is as follows:

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

There is only one constant, $[]$, and $\mathcal{C} = \{\{any, list\}\}$. 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 $f^0 \to \{q \mid f^0 \to q \in \Delta\}$. For the example above, 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. The algorithm returns a more compact representation as a set of product transitions, namely:

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

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 f type *any* that are not of some other type.

6 Deriving an FTA Abstraction of a Program

We now turn to a different aspect of the use of finite tree automata in program analysis. We aim to derive a tree automaton that approximates a given program. Previously, we supplied a tree automaton and built an analysis based on it.

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 [17, 7], debugging [28], assisting compiler optimisations [31, 47], optimising a theorem prover [15], program specialisation [23], planning [5] and verification [7]. The first work in this area was by Reynolds [41]; other early

research was done by Jones and Muchnick [33, 32]. In the past decade two different approaches to deriving set expressions have been followed. One approach is based on abstract interpretation [31, 47, 19, 13, 38], and the other on solving set constraints derived from the program text [27, 17, 26, 2, 1, 34, 8, 40]. 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.

6.1 Abstract Domains of NFTAs

Let P be a definite logic program and $\mathsf{M}[P]$ its minimal Herbrand model. We will construct a set of NFTAs that forms the abstract domain for an abstract interpretation of P.

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.

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.

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 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 $f(Q_1, \ldots, Q_k) \to \{q_X\}$, 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 $\mathcal{Q}, \mathsf{Args}$ and $\mathcal{V}$ be disjoint infinite sets of identifiers. The mapping id_P is chosen to be any injective mapping $\mathsf{headterms}(P) \to \mathcal{Q}$. 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 $\mathcal{V}$. Let type and any be distinguished identifiers not in $\mathcal{Q} \cup \mathsf{Args} \cup \mathcal{V}$.

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 6. S^P_{var}

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

$\mathsf{S}^P_{var}(t) =$ *if t is a clause head, then* type
else if t is a variable, then $\mathsf{varid}(t)$
else $\mathsf{id}_P(t)$

Definition 7. S^P_{arg}

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

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

Example 9. Let P be the *append* program.

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

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.

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

Let $\mathcal{Q} = \{q_1, q_2, \ldots\}$; let id_P map the i^{th} element of $\mathsf{headterms}(P)$ (in the given order) to q_i; let $\mathsf{Args} = \{app_1, app_2, app_3\}$ and let argpos be the mapping such that $\mathsf{argpos}(\langle append, 1\rangle) = app_1, \ldots, \mathsf{argpos}(\langle append, 3\rangle) = app_3$; let $\mathcal{V} = \{a, b, c, d, \ldots\}$, and let $\mathsf{varid}(A) = a, \mathsf{varid}(B) = b$ etc. Then S^P_{var} is the following mapping.

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

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

$append([\,], A, A) \mapsto \mathsf{type}$	$append([B\|C], D, [B\|E]) \mapsto \mathsf{type}$	$D \mapsto app_2$
$[\,] \mapsto app_1$	$[B\|C] \mapsto app_1$	$[B\|E] \mapsto app_3$
$A \mapsto app_2$	$B \mapsto b$	$B \mapsto b$
$A \mapsto app_3$	$C \mapsto c$	$E \mapsto e$

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

6.2 The Abstract Domains for a Program

Given a program P and a mapping $\mathsf{S}^P : \mathsf{headterms}(P) \rightarrow \mathcal{Q} \cup \mathcal{V} \cup \{\mathsf{type}\}$ we define the set of NFTAs that make up the abstract domain. The set contains all the NFTAs whose states come from the powerset of the range of S^P.

Definition 8. *Abstract NFTA-Domains based on P and S^P*

Let P be a definite logic program, and let Σ be the set of function and predicate symbols in P. Let $R^P = \mathsf{range}(\mathsf{S}^P)$, and let $Q^P = 2^{R^P}$. Let Δ^P be the set of transitions $\{f_j^{n_j}(q_1, \ldots, q_{n_j}) \rightarrow q \mid f_j^{n_j} \in \Sigma, \{q_1, \ldots, q_{n_j}, q\} \subseteq Q^P\}$. Note that the states $q_1, \ldots, q_{n_j}$, and q are not elements of $\mathsf{range}(S^P)$, but rather sets of elements.

Then D_P is the following set of automata.

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

Note that $\mathsf{range}(\mathsf{S}^P)$ is finite, and hence D_P is 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. We have an equivalence relation $\equiv$ such that $R_1 \equiv R_2$ iff $L(R_1) \subseteq L(R_2) \wedge L(R_2) \subseteq L(R_1)$. The abstract domain $\mathcal{D}_P$ is the quotient set of D_P under $\equiv$, with the partial order induced by the subset order on the languages represented. I.e. let $d_1, d_2 \in \mathcal{D}_P$, then $d_1 \sqsubseteq d_2$ iff $R_1 \in d_1 \wedge R_2 \in d_2 \Rightarrow L(R_1) \subseteq L(R_2)$.

Define the concretisation functions $\gamma : \mathcal{D}_P \rightarrow 2^{\mathsf{Term}(\Sigma)}$, as $\gamma(d) = L(R)$, where $R \in d$ and $L(R)$ is the language of the NFTA R. γ is monotonic with respect to the partial orders on $\mathcal{D}_P$ and $2^{\mathsf{Term}(\Sigma)}$.

States that are sets containing more than one identifier represent products. For instance, in the transition $f(\{q_1, q_2\}, \{q_3\}) \rightarrow \{q\}$, 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 $R_{\{q_2\}}$.

6.3 Outline of Abstract Interpretation

A summary of the abstract interpretation proceeds as follows. A full description appears in [22]. The least automaton, namely the automaton with zero transitions, is the initial approximation.

On each iteration, the body of each clause is solved with respect to the current approximation, yielding a possible empty set of solutions. Each solution is an assignment of a state in Q^P to each occurrence of each variable occurring in the clause body. Secondly, assignments to variables occurring more than once in the body are checked for consistency. Namely, the product of the states assigned to repeated occurrences of a variable should be non-empty. Finally, the set of consistent assignments is projected onto the head variables. This yields a set of transitions (after eliminating ϵ-transitions) which is added to the current approximation.

The iterations continue until on some iteration no new transition is added. Clearly the number of iterations is bounded since there is a finite number of possible transitions, and the approximations monotonically increase.

Example 10. Let P be the *append* program. Assume we use the state mapping S^P_{var}. On the first iteration we have:

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

For the first clause, projection onto clause head $append([\,], A, A)$ 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. Solving the clause body $append(C, D, E)$ 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 S^P_{arg} generates the following sequence of results: (only the new transitions on each iteration are shown).

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

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. 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.

$append(app_1, app_2, app_3) \rightarrow \mathsf{type}$ $\quad [\,] \rightarrow app_1$ $\quad \mathsf{any} \rightarrow app_2$ $\quad \mathsf{any} \rightarrow app_3$
$[b|app_1] \rightarrow app_1$ $\quad [b|app_3] \rightarrow app_3$ $\quad \mathsf{any} \rightarrow b$

As argued in [13], there is a procedure for solving set constraints generated by a program which is isomorphic to the iterations of an abstract interpretation as shown above.

6.4 Abstract interpretation over Infinite-height Domains of FTAs

In the method outlined above, the abstract domain is program-specific; this is what allows the domain to be finite. Another approach is to define an infinite domain - the set of all FTAs over a given signature and an infinite set of states. We could also define the domain to be the set of all languages over a given signature representable by FTAs.

An abstract interpretation on such a domain requires a *widening* in order to ensure termination of the analysis. Typically, the approach generates FTAs that describe as accurately as possible the initial iterations of the concrete fixpoint semantics. As soon as the descriptions appear to be growing upboundedly, a widening is introduced corresponding to a recursion in the transitions of the automaton.

Example 11. Consider the *append* program again. The first iterations of the concrete bottom-up "immediate consequence" operator return the sets.

1. $\{append([], X, X)\}$
2. $\{append([], X, X), append([A], X, [A|X])\}$
3. $\{append([], X, X), append([A], X, [A|X]), append([A, B], X, [A, B|X]), \}$
4. $\cdots$

The successive terms can be described reasonably accurately by the following sets of transitions.

1. $R_1 = \{append(q_1, any, any) \rightarrow type, [] \rightarrow q_1\}$
2. $R_2 = R_1 \cup \{append(q_2, any, q_3) \rightarrow type, [any|q_1] \rightarrow q_2, [] \rightarrow q_1, [any|any] \rightarrow q_3\}$
3. $R_3 = R_2 \cup \{append(q_4, any, q_5) \rightarrow type, [any|q_2] \rightarrow q_4, [any|q_1] \rightarrow q_2, [] \rightarrow q_1, [any|any] \rightarrow q_3, [any|q_3] \rightarrow q_5\}$
4. $\cdots$

It can be seen that this sequence could be continued indefinitely, since each iteration extends the terms accepted by the first argument of *append*. There are various widening methods [31, 47, 19, 13, 38], which would in effect "notice" the growth of the first argument and introduce a recursive transition which is a fixpoint.

5. $R_4 = \{append(q_6, any, q_3) \rightarrow type, [] \rightarrow q_6, [any|q_6] \rightarrow q_6, [any|any] \rightarrow q_3\}$

The complications of this approach are offset by two factors:

- In principle, greater precision is possible, since any set of terms can be approximated arbitrarily closely by some FTA.
- The finite-height domain is determined by syntactic factors in the source program. For some applications (e.g. online program specialisation [21]) the program itself is being unfolded or otherwise transformed. The widening approach adapts more flexibly to such dynamic applications.

6.5 Deriving an FTA Approximation by Solving Constraints

As outlined in Section 1, procedures exist that transform formulas in some language into FTAs. Examples are definite set constraints [27] and proper unary clauses [17]. Another approach to deriving an FTA approximation of a program can thus be summarised as follows:

1. Given a program P with least model $M[P]$, first construct a set of formulas F_P which are definite set constraints [27] or proper unary clauses [17], such that F_P describes some superset of $M[P]$.
2. Solve the formulas F_P, obtaining an explicit solution represented as an FTA.

Cousot and Cousot [13] show that this method can also be presented as abstract interpretation over a program-specific, finite height domain, as outlined in Section 6.1. However the advantages and disadvantages from an algorithmic point of view of the various approaches is still a subject of research.

Acknowledgements

This article has drawn on papers written by the author in collaboration with several others: D.A. de Waal, D. Boulanger, H. Sağlam, K.S. Henriksen, G. Banda, J.C. Peralta and G, Puebla. Thanks are due to all these. In addition, the author has learned much on the topics of this article from discussions with others, notably M. Bruynooghe, W. Charatonik, M. Codish, P. Cousot, M. Hermenegildo, N. D. Jones, M. Leuschel, A. Podelski, E. Shapiro and all the participants of the European Framework 5 Project ASAP (IST-2001-38059) which supported some of this work.

References

1. Alexander Aiken. Set constraints: Results, applications, and future directions. In Alan Borning, editor, *Principles and Practice of Constraint Programming (PPCP 1994)*, volume 874 of *Springer-Verlag Lecture Notes in Computer Science*, pages 326–335, 1994.
2. Alexander Aiken and Edward L. Wimmers. Solving systems of set constraints (extended abstract). In *IEEE Symposium on Logic in Computer Science (LICS 1992)*, pages 329–340, 1992.

3. D. Boulanger and M. Bruynooghe. A systematic construction of abstract domains. In B. Le Charlier, editor, *Proc. First International Static Analysis Symposium, SAS'94*, volume 864 of *Springer-Verlag Lecture Notes in Computer Science*, pages 61–77, 1994.
4. D. Boulanger, M. Bruynooghe, and M. Denecker. Abstracting s-semantics using a model-theoretic approach. In M. Hermenegildo and J. Penjam, editors, *Proc. 6$^{\text{th}}$ International Symposium on Programming Language Implementation and Logic Programming, PLILP'94*, volume 844 of *Springer-Verlag Lecture Notes in Computer Science*, pages 432–446, 1994.
5. 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.
6. W. Charatonik. Directional type checking for logic programs: Beyond discriminative types. In G. Smolka, editor, *Programming Languages and Systems, 9th European Symposium on Programming, ESOP 2000*, volume 1782 of *Springer-Verlag Lecture Notes in Computer Science*, pages 72–87, 2000.
7. 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.
8. Witold Charatonik, Andreas Podelski, and Jean-Marc Talbot. Paths vs. trees in set-based program analysis. In Tom Reps, editor, *Proceedings of POPL'00: Principles of Programming Languages*, pages 330–338. ACM, ACM Press, January 2000.
9. K. Clark. Predicate logic as a computational formalism. Technical Report DOC 79/59, Imperial College, London, Department of Computing, 1979.
10. Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 2000.
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. S. Dawson, C. R. Ramakrishnan, and D. S. Warren. Practical program analysis using general purpose logic programming systemsa case study. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 17–126, May 1996.
15. D.A. de Waal and J. P. 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.
16. S Debray and D.S. Warren. Automatic mode inference for logic programs. *Journal of Logic Programming*, 5(3):207–229, 1988.
17. T. Frühwirth, E. Shapiro, M.Y. 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.

18. J. P. 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. MIT Press, 1995.
19. J. P. Gallagher and D.A. 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 K. S. Henriksen. Abstract domains based on regular types. In V. Lifschitz and B. Demoen, editors, *Proceedings of the International Conference on Logic Programming (ICLP'2004)*, volume 3132 of *Springer-Verlag Lecture Notes in Computer Science*, pages 27–42, 2004.
21. J. P. Gallagher and J. C. Peralta. Regular tree languages as an abstract domain in program specialisation. *Higher Order and Symbolic Computation*, 14(2,3):143–172, 2001.
22. J. P. Gallagher and G. Puebla. 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, January 2002.
23. John P. Gallagher and Julio 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.
24. Thomas Genet and Valérie Viet Triem Tong. Reachability analysis of term rewriting systems with Timbuk. In Robert Nieuwenhuis and Andrei Voronkov, editors, *LPAR*, volume 2250 of *Lecture Notes in Computer Science*, pages 695–706. Springer, 2001.
25. Jean Goubault-Larrecq. A method for automatic cryptographic protocol verification. In José D. P. Rolim, editor, *15 IPDPS 2000 Workshops, Cancun, Mexico, May 1-5, 2000, Proceedings*, volume 1800 of *Springer-Verlag Lecture Notes in Computer Science*, pages 977–984. Springer, 2000.
26. 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.
27. 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.
28. 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.
29. J. M. Howe and A. King. Positive Boolean Functions as Multiheaded Clauses. In P Codognet, editor, *International Conference on Logic Programming*, volume 2237 of *LNCS*, pages 120–134, November 2001.
30. Mizuho Iwaihara and Yusaku Inoue. Bottom-up evaluation of logic programs using binary decision diagrams. In Philip S. Yu and Arbee L. P. Chen, editors, *ICDE*, pages 467–474. IEEE Computer Society, 1995.
31. 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.

32. 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.
33. Neil D. Jones and Steven 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.
34. Dexter Kozen. Set constraints and logic programming. *Information and Computation*, 143(1):2–25, 1998.
35. J. Lind-Nielsen. BuDDy, a binary decision diagram package. `http://www.itu.dk/research/buddy`.
36. J.W. Lloyd. *Foundations of Logic Programming: 2nd Edition*. Springer-Verlag, 1987.
37. K. Marriott and H. Søndergaard. Bottom-up abstract interpretation of logic programs. In *Proceedings of the Fifth International Conference and Symposium on Logic Programming, Washington*, August 1988.
38. P. Mildner. *Type Domains for Abstract Interpretation: A Critical Study*. PhD thesis, Department of Computer Science, Uppsala University, 1999.
39. David Monniaux. Abstracting cryptographic protocols with tree automata. *Sci. Comput. Program.*, 47(2-3):177–202, 2003.
40. Andreas Podelski, Witold Charatonik, and Martin Müller. Set-based failure analysis for logic programs and concurrent constraint programs. In S. Doaitse Swierstra, editor, *Programming Languages and Systems, 8th European Symposium on Programming, ESOP'99*, volume 1576 of *LNCS*, pages 177–192, 1999.
41. John C. Reynolds. Automatic construction of data set definitions. In J. Morrell, editor, *Information Processing 68*, pages 456–461. North-Holland, 1969.
42. A. Roychoudhury, K. Narayan Kumar, C. R. Ramakrishnan, I. V. Ramakrishnan, and S. A. Smolka. Verification of parameterized systems using logic program transformations. In S. Graf and M. I. Schwartzbach, editors, *Tools and Algorithms for Construction and Analysis of Systems, 6th Int. Conf., TACAS 2000*, volume 1785 of *Springer-Verlag Lecture Notes in Computer Science*, pages 172–187, 2000.
43. H. Sağlam and J. P. Gallagher. Constrained regular approximation of logic programs. In N. Fuchs, editor, *Logic Program Synthesis and Transformation (LOPSTR'97)*, volume 1463 of *Springer-Verlag Lecture Notes in Computer Science*, 1998.
44. R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal of Computing*, 1(2):146–160, 1972.
45. J. D. Ullman. *Principles of Database and Knowledge-Base Systems, Vol. II*. Computer Science Press, Rockville, Md., 1989.
46. J.D. Ullman. Implementation of Logical Query Languages for Databases. *ACM Transactions on Database Systems*, 10(3), 1985.
47. 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.
48. J. Whaley, C. Unkel, and M. S. Lam. A bdd-based deductive database for program analysis, 2004. `http://suif.stanford.edu/bddbddb`.
49. John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In William Pugh and Craig Chambers, editors, *PLDI*, pages 131–144. ACM, 2004.
50. E. Yardeni and E.Y. Shapiro. A type system for logic programs. *Journal of Logic Programming*, 10(2):125–154, 1990.