Approximating Term Rewriting Systems: A Horn Clause Specification and Its Implementation*

John P. Gallagher and Mads Rosendahl

Computer Science, Building 42.2, Roskilde University, DK-4000 Denmark
{jpg,madsr}@ruc.dk

Abstract. We present a technique for approximating the set of reachable terms of a given term rewriting system starting from a given initial regular set of terms. The technique is based on previous work by other authors with the same goal, and yields a finite tree automaton recognising an over-approximation of the set of reachable terms. Our contributions are, firstly, to use Horn clauses to specify the transitions of a possibly infinite-state tree automaton defining (at least) the reachable terms. Apart from being a clear specification, the Horn clause model is the basis for further automatic approximations using standard logic program analysis techniques, yielding finite-state tree automata. The approximations are applied in two stages: first a regular approximation of the model of the given Horn clauses is constructed, and secondly a more precise relational abstraction is built using the first approximation. The analysis uses efficient representations based on BDDs, leading to more scalable implementations. We report on preliminary experimental results.

1 Introduction

We consider the problem of automatically approximating the set of reachable terms of a term rewriting system (TRS) given a set of initial terms. The problem has been studied in several contexts such as flow analysis of higher-order functional languages [20], cryptographic protocol analysis [16] and more generally the static analysis of any programming language whose operational semantics is expressed as a TRS [2]. The applications have in common the goal of proving properties of the set of all reachable terms of the TRS, which is infinite in general. It is obviously sufficient to show that the required property holds in an over-approximation of the reachable set. Safety properties, namely assertions that some given terms are not reachable, form an important class of properties that can proved using over-approximations.

For practical application of this principle it is necessary to describe the over-approximations in some decidable formalism so that the properties can be effectively checked. Regular tree languages, described by tree grammars or finite tree

* Work supported by the Danish Natural Science Research Council project *SAFT: Static Analysis Using Finite Tree Automata*.

I. Cervesato, H. Veith, and A. Voronkov (Eds.): LPAR 2008, LNCS 5330, pp. 682–696, 2008.

automata, provide a suitable formalism. Thus we focus on the specific problem of deriving a regular tree language containing the set of reachable terms of a TRS, starting from a possibly infinite set of initial terms also expressed as a regular tree language. Our approach builds on work by Feuillade *et al.* [9] and also by Jones [20] and Jones and Andersen [21] (an updated version of [20]).

In Section 2 we review the basic notions concerning term rewriting systems and regular tree languages expressed as finite tree automata. Following this, Section 3 contains a Horn clause specification of a possibly infinite tree automaton over-approximating the reachable terms of a given TRS and initial set. The problem is thus shifted to computing the model of this set of Horn clauses, or some approximation of the model. In Section 4, methods of approximating Horn clause models are outlined, drawing on research in the abstract interpretation of logic programs. Section 5 explains how BDD-based methods can be used to compute the (approximate) models, enhancing the scalability of the approach. Some initial experiments are reported. Finally, Section 6 contains a discussion of related work and concludes.

2 Term Rewriting Systems and Their Approximation

A *term rewriting system* (TRS for short) is formed from a non-empty finite signature Σ and a denumerable set of variables $\mathcal{V}$. $\mathsf{Term}(\Sigma \cup \mathcal{V})$ denotes the smallest set containing $\mathcal{V}$ such that $f(t_1, \ldots, t_n) \in \mathsf{Term}(\Sigma \cup \mathcal{V})$ wherever $t_1, \ldots, t_n \in \mathsf{Term}(\Sigma \cup \mathcal{V})$ and $f \in \Sigma$ has arity n. $\mathsf{Term}(\Sigma)$ denotes the subset of $\mathsf{Term}(\Sigma \cup \mathcal{V})$ containing only variable-free (ground) terms. The set of variables in a term t is denoted $\mathsf{vars}(t)$. A *substitution* is a function $\mathcal{V} \to \mathsf{Term}(\Sigma \cup \mathcal{V})$; substitutions are naturally extended to apply to $\mathsf{Term}(\Sigma \cup \mathcal{V})$. We write substitution application in postfix form – $t\theta$ stands for the application of substitution θ to t. A term t' is a *ground instance* of term t if t' is ground and $t' = t\theta$ for some substitution θ. As for notation, variable names from $\mathcal{V}$ will start with a capital letter and elements of Σ will start with lower-case letters, or numbers.

A term rewriting system is a set of rules of the form $l \to r$, where $l, r \in \mathsf{Term}(\Sigma \cup \mathcal{V})$. In a rule $l \to r$, l is called the left-hand-side (lhs) and r the right-hand-side (rhs). We consider here TRSs formed from a finite set of rules, satisfying the conditions $l \notin \mathcal{V}$ and $\mathsf{vars}(r) \subseteq \mathsf{vars}(l)$. A left- (resp. right-) linear TRS is one in which no variable occurs more than once in the lhs (resp. rhs) of a rule. In this paper we consider only left-linear TRSs.

The operational semantics of a TRS is defined as the reflexive, transitive closure of a relation $\Rightarrow$ over $\mathsf{Term}(\Sigma) \times \mathsf{Term}(\Sigma)$. The $\Rightarrow$ relation captures the concept of a "rewrite step". Intuitively, $t_1 \Rightarrow t_2$ for $t_1, t_2 \in \mathsf{Term}(\Sigma)$ holds if there is a subterm of t_1 that is a ground instance of the lhs of some rewrite rule; t_2 is the result of replacing that subterm by the corresponding instance of the rhs. More precisely, define a *(ground) context* to be a term from $\mathsf{Term}(\Sigma \cup \{\bullet\})$ containing exactly one occurrence of $\bullet$. Let c be a context; define $c[t] \in \mathsf{Term}(\Sigma)$ to be the term resulting from replacing $\bullet$ by $t \in \mathsf{Term}(\Sigma)$. Then given a TRS, we define the relation $\Rightarrow$ as the set of pairs of the form $c[l\theta] \Rightarrow c[r\theta]$ where c is a context, $l \to r$ is a rule in the TRS and there is a substitution θ such that $l\theta$

is a ground instance of l (and hence $r\theta$ is also ground). The reflexive, transitive closure of $\Rightarrow$ is denoted $\Rightarrow^*$. Given a set of ground terms S and a TRS, the set of terms reachable from S is defined as $\mathsf{reach}(S) = \{t \mid s \in S, s \Rightarrow^* t\}$.

We are not concerned here with important aspects of TRSs such as confluence, rewrite strategies, or the distinction between constructors and defined functions. Details on TRSs can be found in the literature, for example in [8].

2.1 Finite Tree Automata

Finite tree automata (FTAs) can be seen as a restricted class of rewriting system. An FTA with signature Σ is a finite set of ground rewrite rules over an extended signature $\Sigma \cup \mathcal{Q}$ where $\mathcal{Q}$ is a set of constants (unary function symbols) disjoint from Σ, called *states*. A set $\mathcal{Q}_f \subseteq \mathcal{Q}$ is called the set of *accepting* or *final* states. Each rule is of the form $f(q_1, \ldots, q_n) \to q$ where $q_1, \ldots, q_n, q \in \mathcal{Q}$ and $f \in \Sigma$ has arity n. The relation $\Rightarrow^*$ is defined as for TRSs in general but we are interested mainly in pairs $t \Rightarrow^* q$ where $t \in \mathsf{Term}(\Sigma)$ and $q \in \mathcal{Q}_f$. In such a case we say that there is a successful *run* of the FTA for term t, or that t is *accepted* by the FTA. For a given FTA A we define $\mathcal{L}(A)$, the *term language* of A, to be $\{t \in \mathsf{Term}(\Sigma) \mid t \text{ is accepted by } A\}$.

The main point of interest of FTAs is that they define so-called *regular tree languages*, which have a number of desirable computational properties. Given an FTA A it is decidable whether $\mathcal{L}(A)$ is empty, finite or infinite, and whether a given term $t \in \mathcal{L}(A)$. Furthermore, regular tree languages are closed under union, intersection and complementation; given FTAs A_1 and A_2, we can construct an FTA $A_1 \cup A_2$ such that $\mathcal{L}(A_1 \cup A_2) = \mathcal{L}(A_1) \cup \mathcal{L}(A_2)$, and similarly for the intersection and complementation operations.

A *bottom-up deterministic* FTA, or DFTA, is one in which no two rules have the same lhs. A *complete* FTA is one in which there is a rule $f(q_1, \ldots, q_n) \to q$ for every $f \in \Sigma$ of arity n, and states $q_1, \ldots, q_n \in \mathcal{Q}$. It can be shown that for every FTA there is a complete DFTA (usually with a different set of states) accepting the same language. Further information about FTAs and regular tree languages can be found in the literature, for example in [6].

2.2 Approximation of TRSs

The set of reachable states of a TRS is not in general a regular tree language. For instance consider the TRS containing a single rule $f(X, Y) \to f(g(X), h(Y))$. The set $\mathsf{reach}(\{f(a, b)\})$ is $\{f(g^k(a), h^k(b)) \mid k > 0\}$, and it can be shown that there is no FTA that accepts precisely this set of terms. However, given a TRS and an initial set of terms S there is always an FTA A such that $\mathcal{L}(A) \supseteq \mathsf{reach}(S)$. In such a situation $\mathcal{L}(A)$ is called an *over-approximation* of $\mathsf{reach}(S)$. There could of course be more than one possible FTA defining an over-approximation and generally we prefer the most *precise*, that is, smallest over-approximations. Here, as usual in static analysis problems, there is a trade-off between complexity and precision; the more precise the approximation, the more expensive it tends to be to construct it.

There are a number of cases where an FTA gives a perfectly precise approximation, that is, there exists an FTA A such that $\mathcal{L}(A) = \mathsf{reach}(S)$. These cases

Term Rewriting System	FTA defining initial terms
$plus(0, X) \to X.$	$even(qpo) \to qf.$
$plus(s(X), Y) \to s(plus(X, Y)).$	$even(qpe) \to qf.$
$even(0) \to true.$	$s(qeven) \to qodd.$
$even(s(0)) \to false.$	$s(qodd) \to qeven.$
$even(s(X)) \to odd(X).$	$plus(qodd, qodd) \to qpo.$
$odd(0) \to false.$	$plus(qeven, qeven) \to qpe.$
$odd(s(0)) \to true.$	$0 \to qeven.$
$odd(s(X)) \to even(X).$	(Accepting state is qf)

Fig. 1. A TRS and an FTA (from [9]) defining the initial terms in Example 1

are often characterised by syntactic conditions on the TRS and the initial set S. Discussion of various classes for which this holds is contained in [19,6,9].

Example 1. To illustrate these concepts we use the TRS in Figure 1 taken from [9]. The TRS defines the operation *plus* on natural numbers in successor notation (i.e. n represented by $s^n(0)$), and the predicates *even* and *odd* on natural numbers. The FTA (call it A) defines the set of "calls" $even(plus(n_1, n_2))$ where n_1, n_2 are either both even or both odd. It can be checked that, for example $even(plus(s(0), s(s(s(0))))) \Rightarrow^* qf$ in the FTA, meaning that it is contained in the set of initial terms, and that $even(plus(s(0), s(s(s(0))))) \Rightarrow^* true$ in the TRS. The property to be proved here is that $false \notin \mathsf{reach}(\mathcal{L}(A))$, in other words, that the sum of two even or two odd numbers is not odd.

This example happens to be one where the set $\mathsf{reach}(\mathcal{L}(A))$ is precisely expressible as an FTA. The procedure given in [9] can compute this FTA and check that *false* is not accepted by it, thus proving the required property. (As we will see our method can also achieve this).

3 Horn Clause Specification of Reachable Term Approximations

In this section we present a procedure for constructing an over-approximating FTA, given a term rewriting system R and an initial set of terms S. We assume that R is left-linear and that S is regular tree language specified by an FTA.

Our procedure is based initially on the one given in [9] and is also inspired by [20]. However unlike these works we express the procedure as a set of Horn clauses in such as way as to allow the construction of a tree automaton with an infinite set of states. In other words, the Horn clauses have a possible infinite model. Following this, we draw on existing techniques for approximating Horn clause models to produce an FTA approximation. We argue that this is a flexible and scalable approach, which exploits advances made in the analysis of logic programs.

3.1 Definite Horn Clauses

We first define definite Horn clauses and their semantics. Definite Horn clauses form a fragment of first-order logic. Let Σ be a set of function symbols, $\mathcal{V}$ a

set of variables and $\mathcal{P}$ be a set of predicate symbols. An *atomic formula* is an expression of the form $p(t_1, \ldots, t_n)$ where $p \in \mathcal{P}$ is an n-ary predicate symbol and $t_1, \ldots, t_n \in \mathsf{Term}(\Sigma \cup \mathcal{V})$. A definite Horn clause is a logical formula of the form $\forall((u_1 \wedge \ldots \wedge u_m) \rightarrow u)$ $(m \geq 0)$ where $u_1, \ldots, u_m, u$ are atomic formulas. If $m = 0$ the clause is called a *fact* or *unit clause*. A fact is often written as $true \rightarrow u$. The symbol $\forall$ indicates that all variables occurring in the clause are universally quantified over the whole formula.

From now on we will write Horn clauses "backwards" as is the convention in logic programs, and use a comma instead of the conjunction $\wedge$. The universal quantifiers are also implicit. Thus a Horn clause is written as $u \leftarrow u_1, \ldots, u_m$ or $u \leftarrow true$ in the case of facts.

The semantics of a set of Horn clauses is obtained using the usual notions of interpretation and model from classical logic. An *interpretation* is defined by (i) a domain of interpretation which is a non-empty set D; (ii) a *pre-interpretation* which is a function mapping each n-ary function $f \in \Sigma$ to a function $D^n \rightarrow D$; and (iii) a predicate interpretation which assigns to each n-ary predicate in $\mathcal{P}$ a relation in D^n. A model of a set of Horn clauses is an interpretation that satisfies each clause (using the usual notion of "satisfies" based on the meanings of the logic connectives and quantifiers; see for example [22] or any textbook on logic).

The *Herbrand interpretation* is of special significance. The domain D of a Herbrand interpretation is $\mathsf{Term}(\Sigma)$; the pre-interpretation maps each n-ary function f to the function $\hat{f} : \mathsf{Term}(\Sigma)^n \rightarrow \mathsf{Term}(\Sigma)$ defined by $\hat{f}(t_1, \ldots, t_n) = f(t_1, \ldots, t_n)$. A Herbrand interpretation of the predicates assigns each n-ary predicate a relation in $\mathsf{Term}(\Sigma)^n$. Such a relation can be conveniently represented as a set of ground atomic formulas $p(t_1, \ldots, t_n)$ where $t_1, \ldots, t_n \in \mathsf{Term}(\Sigma)$. There exists a *least* Herbrand model (which may be infinite) of a set of definite Horn clauses, which is the most concrete interpretation; it contains exactly those ground atomic formulas that are true in every interpretation.

As will be discussed in Section 4, models other than Herbrand models are of interest for the purpose of abstracting the meaning of a set of Horn clauses.

Horn Clause representation of FTAs A simple Horn Clause representation of an FTA with signature Σ and set of states $\mathcal{Q}$ can be obtained by regarding the rewrite arrow $\rightarrow$ as a binary predicate and $\Sigma \cup \mathcal{Q}$ as the signature of the Horn clause language. An FTA rule $f(q_1, \ldots, q_n) \rightarrow q$ is thus a unit Horn clause or fact written as $(f(q_1, \ldots, q_n) \rightarrow q) \leftarrow true$.

3.2 Tree Automata Approximation of Reachable Terms

We now return to the problem of computing an FTA over-approximating the set of reachable terms of a left-linear TRS. The approach taken in [9] is based on the notion of *completion*. Intuitively, this works as follows. Let $l \rightarrow r$ be a rule in the TRS and suppose that I is the FTA defining the set of initial terms. Then the FTA A defining $\mathsf{reach}(\mathcal{L}(I))$ has to be such that

(i) $\mathcal{L}(A) \supseteq \mathcal{L}(I)$ and

(ii) if there is some ground instance of l, say $l\theta$, where θ is a substitution mapping variables to FTA states, such that $l\theta \Rightarrow^* q$ in A, for some FTA (not necessarily final) state q then $r\theta \Rightarrow^* q$ should also hold in A.

We illustrate the principle referring to Example 1. Consider the second rule of the TRS, $plus(s(X), Y) \rightarrow s(plus(X, Y))$. There exists a substitution θ, namely $\{X = qeven, Y = qodd\}$ such that $plus(s(X), Y)\theta \Rightarrow^* qpo$ using the initial FTA. Hence by requirement (ii) the FTA for the reachable terms should be such that $s(plus(X, Y))\theta \Rightarrow^* qpo$. This could be ensured by adding the rules $plus(qeven, qodd) \rightarrow q_0, s(q_0) \rightarrow qpo$, for some state q_0.

We now show how we can construct a set of Horn clauses H_A capturing requirements (i) and (ii). The Horn clauses will be such that their minimal Herbrand model will be a set of atomic formulas $f(q_1, \ldots, q_n) \rightarrow q$ defining the rules of the required FTA. First of all, H_A contains the clauses corresponding to the rules in I, which ensures condition (i) above.

Secondly, consider requirement (ii). Given a TRS rule $l \rightarrow r$, we construct Horn clauses as follows. Define the operation $\mathsf{flatlhs}(t \rightarrow Y)$ as follows, where $t \in \mathsf{Term}(\Sigma \cup \mathcal{V})$ and $Y \in \mathcal{V}$:

- $\mathsf{flatlhs}(t \rightarrow Y) = \{t \rightarrow Y\}$, if t has no non-variable proper subterms;
- $\mathsf{flatlhs}(t \rightarrow Y) = \mathsf{flatlhs}(t' \rightarrow Y) \cup \{f(X_1, \ldots, X_n) \rightarrow Y'\}$, if $f(X_1, \ldots, X_n)$ $(n \geq 0)$ is a proper subterm of t whose arguments are all variables, Y' is a fresh variable and t' is the result of replacing the subterm $f(X_1, \ldots, X_n)$ by Y' in t.

Example 2. Consider the rule $even(s(0)) \rightarrow false$. Then $\mathsf{flatlhs}(even(s(0)) \rightarrow Y_0)$ is $\{0 \rightarrow Y_1, s(Y_1) \rightarrow Y_2, even(Y_2) \rightarrow Y_0\}$.

The following claim establishes that the flattened form can be used to check the condition that there exists a substitution such that $l\theta \Rightarrow^* q$.

Claim. Let t be a linear term, A an FTA and Y a variable not occurring in t. Then there is a substitution θ mapping variables of t to FTA states such that $t\theta \Rightarrow^* q$ for some FTA state q if and only if the conjunction $\wedge(\mathsf{flatlhs}(t \rightarrow Y))$ is satisfiable in the set of FTA rules.

Now we come to the critical aspect; how to ensure that $r\theta \Rightarrow^* q$ whenever $l\theta \Rightarrow^* q$. The key question is how to choose the states in the intermediate steps of the derivation that is to be constructed (e.g. the state q_0 above). This question is discussed in detail in [9]. Essentially, in order not to lose precision, a new set of states should be created unique to that derivation, that is, depending only on the rule $l \rightarrow r$ and the substitution θ. However, when applied repeatedly to the same rule this strategy can lead to the creation of an infinite number of new states. An *abstraction* function is introduced in [9], whose purpose is to specify how to introduce a finite set of states in order to satisfy requirement (ii). The question of finding the "right" abstraction function becomes crucial.

Example 3. Consider the TRS rule $f(X) \rightarrow f(g(X))$. Suppose the initial FTA contains a rule $f(q_0) \rightarrow q_1$. Applying the completion principle, we find the

substitution $\theta_0 = \{X = q_0\}$ such that $f(X)\theta_0 \Rightarrow^* q_1$. To build the required derivation $f(g(q_0)) \Rightarrow^* q_1$ we create a new state q_2 and add the rules $g(q_0) \to q_2$ and $f(q_2) \to q_1$. Applying the completion again we find a substitution $\theta_1 = \{X = q_2\}$ such that $f(X)\theta_1 \Rightarrow^* q_1$, and then try to construct a derivation $f(g(q_2)) \Rightarrow^* q_1$. If we choose the same intermediate state q_2 we add unintended derivations (such as $g(g(q_2)) \Rightarrow^* q_2$). Therefore we pick a fresh state q_3 and add the rules $g(q_2) \to q_3$ and $f(q_3) \to q_1$. Clearly this process continues indefinitely and an infinite number of states would be added. A solution using an abstraction function, as in [9], might indeed pick the same state q_2 in both completion steps, but at the cost of possibly adding non-reachable terms.

Our approach is different; we defer the decision on how to abstract the states and define a construction that can introduce an infinite number of states. Given a TRS rule $l \to r$, associate with each occurrence of a non-variable proper subterm w of r a unique function symbol, say q_w, whose arity is the number of distinct variables within w. Define $\mathsf{flatrhs}(t \to Y)$ as follows.

- $\mathsf{flatrhs}(t \to Y) = \{t \to Y\}$, if t has no non-variable proper subterms except possibly "state" terms of form $q_{w'}(\bar{Z})$;
- $\mathsf{flatrhs}(t \to Y) = \mathsf{flatlhs}(t' \to Y) \cup \{f(s_1, \ldots, s_n) \to q_w(\bar{Z})\}$, if $f(s_1, \ldots, s_n)$ $(n \geq 0)$ is a proper subterm of t whose arguments s_i are all either variables or state terms, q_w is the function symbol associated with that subterm, $\bar{Z}$ is the tuple of distinct variables in the subterm, and t' is the result of replacing the subterm $f(s_1, \ldots, s_n)$ by $q_w(\bar{Z})$ in t.

The terms $q_w(\bar{Z})$ represent newly created FTA states. The arguments $\bar{Z}$ are substituted during the completion step and thus the states are unique to their position in r and the substitution θ associated with a completion step.

Example 4. Consider the rule in Example 3 above. Associate the unary function q_2 with the subterm $g(X)$ of the rhs. Then $\mathsf{flatrhs}(f(g(X)) \to Y_0)$ is $\{g(X) \to q_2(X), f(q_2(X)) \to Y_0\}$.

Now, the completion step simply adds new rules corresponding to the flattened form of the rhs.

Example 5. Again, considering Example 3, the first step adds the rules $g(q_0) \to q_2(q_0)$ and $f(q_2(q_0)) \to q_1$. The second step adds rules $g(q_2(q_0)) \to q_2(q_2(q_0))$ and $f(q_2(q_2(q_0))) \to q_1$, and so on.

If the rhs of a rule is a variable, the procedure flatrhs does not apply. In such cases, we replace the rule with a set of rules, one for each n-ary function f in Σ, in which the variable is substituted throughout in both lhs and rhs by a term $f(Z_1, \ldots, Z_n)$, where $Z_1, \ldots, Z_n$ are fresh distinct variables.

Example 6. The rule $plus(0, X) \to X$ in Figure 1 is replaced by the rules $plus(0, 0) \to 0$, $plus(0, s(Z)) \to s(Z)$, $plus(0, plus(Z_1, Z_2)) \to plus(Z_1, Z_2)$, and so on for each function in the signature.

```
odd(B)->D          :- 0->A, odd(B)->C, plus(A,C)->D.
false->C           :- 0->A, false->B, plus(A,B)->C.
true->C            :- 0->A, true->B, plus(A,B)->C.
even(B)->D         :- 0->A, even(B)->C, plus(A,C)->D.
s(B)->D            :- 0->A, s(B)->C, plus(A,C)->D.
0->C               :- 0->A, 0->B, plus(A,B)->C.
plus(B,C)->E       :- 0->A, plus(B,C)->D, plus(A,D)->E.
plus(A,C)->q0(A,C) :- s(A)->B, plus(B,C)->D.
s(q0(A,C))->D      :- s(A)->B, plus(B,C)->D.
true->B            :- 0->A, even(A)->B.
false->C           :- 0->A, s(A)->B, even(B)->C.
odd(A)->C          :- s(A)->B, even(B)->C.
false->B           :- 0->A, odd(A)->B.
true->C            :- 0->A, s(A)->B, odd(B)->C.
even(A)->C         :- s(A)->B, odd(B)->C.

%Initial FTA rules
even(qpo)->qf          :- true.      plus(qodd,qodd)->qpo    :- true.
even(qpe)->qf          :- true.      plus(qeven,qeven)->qpe  :- true.
s(qeven)->qodd         :- true.      0->qeven                :- true.
s(qodd)->qeven         :- true.
```

Fig. 2. Horn Clauses for the TRS and FTA in Figure 1

The Horn clauses specifying the completion can now be constructed. For each rule $l \to r$ in the TRS (with variable rhs rules replaced as just shown) let Q be a variable not occurring in the rule. Construct the set of Horn clauses $\{H \leftarrow \wedge(\mathsf{flatlhs}(l \to Q)) \mid H \in \mathsf{flatrhs}(r \to Q)\}$.

Example 7. Consider the rule $plus(s(X), Y) \to s(plus(X, Y))$ from Example 1. Let q_0 be the unique function symbol associated with the subterm $plus(X, Y)$ of the rhs. Then the two Horn clauses for the rule are:

$$(plus(X, Y) \to q_0(X, Y)) \leftarrow (s(X) \to Q_1), (plus(Q_1, Y) \to Q).$$
$$(s(q_0(X, Y)) \to Q) \leftarrow (s(X) \to Q_1), (plus(Q_1, Y) \to Q).$$

These rules, together with all the others for the TRS and FTA in Figure 1, are shown in Figure 2. Note that the first seven clauses correspond to the instances of the rule $plus(0, X) \to X$.

The minimal Herbrand model of a Horn clause program constructed in this way contains a set of tree automaton rules. If the model is finite it represents an FTA A, whose states are teh states occurring in the rules and whose final states are the same as those of the initial FTA I (which is a subset of A). In this case we claim that $\mathcal{L}(A) \supseteq \mathsf{reach}(\mathcal{L}(I))$. The Horn clause program directly captures the requirements (i) and (ii) discussed at the start of Section 3.2. Furthermore, conditions under which $\mathcal{L}(A) = \mathsf{reach}(\mathcal{L}(I))$ can be obtained using the formal techniques in [9] as a guide, but this is beyond the scope of this paper. Our primary interest is in handling the case where the model is infinite and therefore not an FTA.

Even where A is infinite, the $\Rightarrow^*$ and the set $\mathcal{L}(A)$ are defined; $\mathcal{L}(A)$ contains $\mathsf{reach}(\mathcal{L}(I))$, though $\mathcal{L}(A)$ is no longer a regular language. The problem of reasoning about the reachable terms $\mathsf{reach}(\mathcal{L}(I))$ is now shifted to the problem of reasoning about $\mathcal{L}(A)$. In our approach, we use static analysis techniques developed for approximating Horn clauses models.

4 Tree Automata Approximations of Horn Clause Models

The minimal Herbrand model of a definite Horn clause program P can be computed as the least fixed point of an immediate consequences operator T_P; the least fixed point is the limit of the sequence $\{T_P^n(\emptyset)\}$, $n = 0, 1, 2, \ldots$ [22]. In general this sequence does not terminate because the model of P is infinite.

As it happens the program in Figure 2 has a finite model and hence the sequence converges to a limit in a finite number of steps. Furthermore the minimal Herbrand model thus obtained contains no rule with lhs *false*, thus proving the required property since *false* cannot possibly be reachable.

However, in general the fixpoint construction does not terminate, as already noted. In this case we turn to general techniques for approximating the models of Horn clauses, developed in the field of static analysis of logic programs. Such methods are mostly based on abstract interpretation [7].

4.1 FTA Approximation of the Minimal Model

The problem of computing regular tree language approximations of logic programs has its roots in automatic type inference [23,5,18]. Later, various authors developed techniques for computing tree grammar approximations of the model of a logic program [17,10,12,26,27,15]. Approaches based on solving set constraints [17] can be contrasted with those computing an abstract interpretation over a domain of tree grammars, but all of these have in common that they derive a tree grammar in some form, whose language over-approximates the model of the analysed program. The most precise of these techniques, such as [17,10,15] compute a general FTA (rather than a top-down deterministic tree grammar).

These techniques (specifically, the implementation based on [15]) have been applied to our Horn clause programs defining the reachable terms in a TRS. This yields an FTA describing a superset of the model of the Horn clause program. More precisely, let P be a Horn clause program and $M[\![P]\!]$ be the minimal Herbrand model of P. We derive an FTA A_P such that $\mathcal{L}(A_P) \supseteq M[\![P]\!]$.

In our case, $M[\![P]\!]$ contains the rules of a tree automaton and the approximating FTA A_P describes a superset of $M[\![P]\!]$; that is, $\mathcal{L}(A_P)$ is a set of *rules*. (It is important not to confuse the FTA derived by regular tree approximation of the Horn clause model from the tree automaton represented by the model itself). A_P could in principle be used itself in order to reason about reachability properties, but the possibilities here seem to be limited to checking whether rules of a certain syntactic form are present in the model. The main purpose of the approximation A_P is as a stepping stone for deriving a more precise approximation and obtaining an FTA approximation of $\mathsf{reach}(\mathcal{L}(I))$.

Term Rewriting System

$plus(0, X) \rightarrow X.$	$even(s(X)) \rightarrow odd(X).$
$plus(s(X), Y) \rightarrow s(plus(X, Y)).$	$odd(0) \rightarrow false.$
$times(0, X) \rightarrow 0.$	$odd(s(0)) \rightarrow true.$
$times(s(X), Y) \rightarrow plus(Y, times(X, Y)).$	$odd(s(X)) \rightarrow even(X).$
$square(X) \rightarrow times(X, X).$	$even(square(X)) \rightarrow odd(square(s(X))).$
$even(0) \rightarrow true.$	$odd(square(X)) \rightarrow even(square(s(X))).$
$even(s(0)) \rightarrow false.$	

FTA defining initial terms

$even(s1) \rightarrow s0.$	$square(s2) \rightarrow s1.$
$0 \rightarrow s2.$	

Fig. 3. Another TRS and an FTA (from [9])

4.2 Relational Abstract Interpretation Based on an FTA

In [13] it was shown that an arbitrary FTA could be used to construct an abstract interpretation of a logic program. The process has the following steps: (i) construct an equivalent *complete deterministic FTA* (or DFTA) from the given FTA; define a finite *pre-interpretation* whose domain is the set of states of the DFTA; (iii) compute the least model with respect to that pre-interpretation using a terminating iterative fixpoint computation. The general approach of using pre-interpretations as abstractions was developed in [3,4] and a practical framework and experiments establishing its practicality were described in [11].

The abstract models derived from FTAs in this way are *relational abstractions*; they are optimal in the sense that they are the most precise models based on the derived pre-interpretation. The computed model is in general more precise than the language of the original FTA from which the pre-interpretation is constructed. A simple example illustrates the kind of precision gain. Given the standard program for appending lists

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

the FTA approximation methods cited yield the rules $\{append(list, any, any) \rightarrow append\_type, [] \rightarrow list, [any|list] \rightarrow list\}$ together with some rules defining *any*. It can be seen that no information is returned about the second and third arguments of *append*. The pre-interpretation obtained by determinising this FTA has a domain $\{list, nonlist\}$ with interpretation $\mathcal{D}$ given by

$$\mathcal{D}([]) = list, \mathcal{D}([list|list]) = \mathcal{D}([nonlist|list]) = list,$$
$$\mathcal{D}([list|nonlist] = \mathcal{D}([nonlist|nonlist] = nonlist.$$

The minimal model with respect to this pre-interpretation is the relation

$$\{append(list, list, list), append(list, nonlist, nonlist)\}$$

which is more precise, containing relational information on the dependency between the second and third arguments. Note that no further information is

```
odd(B)->D             :- 0->A, odd(B)->C, plus(A,C)->D.
false->C              :- 0->A, false->B, plus(A,B)->C.
true->C               :- 0->A, true->B, plus(A,B)->C.
even(B)->D            :- 0->A, even(B)->C, plus(A,C)->D.
square(B)->D          :- 0->A, square(B)->C, plus(A,C)->D.
times(B,C)->E         :- 0->A, times(B,C)->D, plus(A,D)->E.
s(B)->D               :- 0->A, s(B)->C, plus(A,C)->D.
0->C                  :- 0->A, 0->B, plus(A,B)->C.
plus(B,C)->E          :- 0->A, plus(B,C)->D, plus(A,D)->E.
plus(A,C)->q0(A,C)    :- s(A)->B, plus(B,C)->D.
s(q0(A,C))->D         :- s(A)->B, plus(B,C)->D.
0->C                  :- 0->A, times(A,B)->C.
times(A,C)->q1(A,C)   :- s(A)->B, times(B,C)->D.
plus(C,q1(A,C))->D    :- s(A)->B, times(B,C)->D.
times(A,A)->B         :- square(A)->B.
true->B               :- 0->A, even(A)->B.
false->C              :- 0->A, s(A)->B, even(B)->C.
odd(A)->C             :- s(A)->B, even(B)->C.
false->B              :- 0->A, odd(A)->B.
true->C               :- 0->A, s(A)->B, odd(B)->C.
even(A)->C            :- s(A)->B, odd(B)->C.
s(A)->q2(A)           :- square(A)->B, even(B)->C.
square(q2(A))->q3(A)  :- square(A)->B, even(B)->C.
odd(q3(A))->C         :- square(A)->B, even(B)->C.
s(A)->q4(A)           :- square(A)->B, odd(B)->C.
square(q4(A))->q5(A)  :- square(A)->B, odd(B)->C.
even(q5(A))->C        :- square(A)->B, odd(B)->C.
even(s1)->s0          :- true.
square(s2)->s1        :- true.
0->s2                 :- true.
```

Fig. 4. Horn Clauses for the TRS and FTA in Figure 3

supplied beyond the original FTA; the analysis just exploits more fully the information available in the FTA.

Let P be a Horn clause program constructed from a TRS, as shown in Section 3. A pre-interpretation $\mathcal{D}$ maps each state of the possibly infinite automaton in the model $M[\![P]\!]$ onto one of the finite number of domain elements of $\mathcal{D}$. Let $f(q_1,\ldots,q_n) \to q \in M[\![P]\!]$. $\mathcal{D}$ is extended to rules as follows: $\mathcal{D}(f(q_1,\ldots,q_n) \to q) = f(\mathcal{D}(q_1),\ldots,\mathcal{D}(q_n)) \to \mathcal{D}(q)$. Define the FTA $M^{\mathcal{D}}[\![P]\!] = \{\mathcal{D}(l \to r) \mid l \to r \in M[\![P]\!]\}$. It is trivial to show that $\mathcal{L}(M^{\mathcal{D}}[\![P]\!]) \supseteq \mathcal{L}(M[\![P]\!])$, and hence $\mathcal{L}(M^{\mathcal{D}}[\![P]\!]) \supseteq \mathsf{reach}(\mathcal{L}(I))$.

Example 8. Consider the example in Figure 3, also taken from [9]. This example is intended to show that the square of any even (resp. odd) number is even (resp. odd). In order to prove this it is sufficient to show that *false* is not reachable from the given initial FTA. Applying the Horn clause construction we obtain the program in Figure 4. The model of this program is not finite. We compute

an FTA approximation of this program using the method of [15]. The output FTA is automatically refined by splitting all non-recursive states as follows. Let q be a non-recursive state having rules $l_1 \to q, \ldots, l_k \to q$ $(k > 1)$ in the FTA. Introduce fresh states $q^1, \ldots, q^k$ and replace the rules for q with rules $l_1 \to q^1, \ldots, l_k \to q^k$. Then replace each rule containing q by k copies with q replaced by $q^1, \ldots, q^k$ respectively. This transformation (which is optional) preserves the language of the automaton and gives a more fine-grained abstraction. Following this we construct a pre-interpretation containing 53 elements which are the states of the determinised transformed FTA and compute the abstract model of the Horn clause program. The model contains no rule $\mathit{false} \to s0$, thus proving the required property. Compare this automatic approach (no information is supplied beyond the TRS and the initial FTA) with the semi-automatic method described in [9] in which an abstraction is introduced on the fly with no obvious motivation for the particular abstraction chosen.

Other approximation techniques are available apart from those illustrated in Example 8. For example, a pre-interpretation $\mathcal{D}_{pos}$ can be defined as $\mathcal{D}_{pos}(q(\_, \ldots, \_) = q$; this simply abstracts each state by the identifier of the position in the TRS where it originated, ignoring the substitution θ. The result is closely related to the approximation obtained by Jones [20].

5 BDD-Based Implementation

In [14] it was shown that static analysis of logic programs using abstraction based on FTAs could be implemented using BDD-based techniques. The main components of this method are (i) a generic tool for computing the model of a Datalog program [25] and (ii) a compact representation (*product form*) of the determinised FTA used to construct the pre-interpretation. Determinisation can cause exponential blow-up in the number of states and rules, but our experience is that the number of states remains manageable. The number of rules can blow up in any case but using the product form often yields orders of magnitude reduction in the representation size. The product form can also be represented as a Datalog program.

(Definite) Datalog programs consist of Horn clauses containing no function symbols of arity more than zero. Our Horn clauses contain such functions but they are easily transformed away; an atomic formula of the form $f(X_1, \ldots, X_n) \to Y$ is transformed to $\mathit{rule\_f}(X_1, \ldots, X_n, Y)$ which conforms to the Datalog syntax. Details of how to represent the pre-interpretation and the determinised FTA as Datalog clauses can be found in [14].

Preliminary experiments using examples from [9,1] indicate that the BDD-based tool gives substantial speedup compared to other approaches. The `evenodd` and `evensq` examples are those in Figures 1 and 3. `smart`, `combi` and `nspk` were kindly supplied by the authors of [1], where they are described in more detail. `smart` and `nspk` are cryptographic protocol models, while `combi` is simply a combinatorial example which produces a large FTA. In Figure 5, P gives the number of clauses in the Horn clause program, the two $\mathcal{D}$ columns give the size

Name	P	$\mathcal{D}$ size (states)	$\mathcal{D}$ time (msecs)	model (msecs)	proof
`evenodd`	22	17	20	50	✓
`evensq`	30	53	66	179	✓
`smart`	123	97 (pos)	8	3270	✓
`combi`	44	25	54	38	n/a
`nspk`	56	183	1782	1170	×

Fig. 5. Experimental results

of the pre-interpretation and the time required to generate it, and the next column shows the time taken to compute the model. The last column indicates whether the required property was proved. In the case of `nspk` the property was not proved in the abstract model. The generation of the pre-interpretation for `smart` as described in Section 4 exhausted memory; however the simpler pre-interpretation ($\mathcal{D}_{pos}$, see Section 4) was sufficient to prove the required property. Timings were taken on a machine with a dual-core Athlon 64-bit processor, with 4GByte RAM, and the tools are implemented in Ciao Prolog and the `bddbddb` tool developed by Whaley [29,28]. These results indicate that the model computation appears to scale well, but further research on generating useful pre-interpretation is needed, and comparing with the strategies for generating abstractions in other approaches, especially the Timbuk system [16].

6 Discussion and Conclusions

The most closely related work to ours, as already mentioned, is that of Feuillade *et al.* [9]. That research has yielded impressive experimental results and extensive analysis of the various classes of TRS that have exact solutions. A tool (Timbuk [16]) has been developed to support the work. Our approach contrasts with that work by being completely automatic, relying more on established analysis techniques based on abstract interpretation, and by permitting a BDD-based implementation that should give greater scalability.

The work of Jones and Andersen [20,21] pre-dates the above work though the authors of [9] seem to be unaware of it. It has some similarities with the above work, in that it is based on a completion procedure. It differs in allowing a more flexible tree grammar to represent the reachable terms. However this flexibility leads to a more complex completion procedure. The set of FTA states (tree grammar non-terminals) is fixed in their procedure. Their work is concerned with flow analysis and follows in the footsteps of tree grammar approximation going back to Reynolds [24].

The next stage in our research is to experiment with the various available regular tree approximation algorithms for Horn clauses, and the generation of pre-interpretations from the resulting FTAs. Following this we hope to perform substantial experiments on the scalability and precision of our techniques, building on the promising application on smaller examples so far.

Acknowledgements. We thank the anonymous referees for LPAR 2008 for detailed and helpful comments.

References

1. Balland, E., Boichut, Y., Moreau, P.-E., Genet, T.: Towards an efficient implementation of tree automata completion. In: Meseguer, J., Roşu, G. (eds.) AMAST 2008. LNCS, vol. 5140, pp. 67–82. Springer, Heidelberg (2008)
2. Boichut, Y., Genet, T., Jensen, T.P., Roux, L.L.: Rewriting approximations for fast prototyping of static analyzers. In: Baader, F. (ed.) RTA 2007. LNCS, vol. 4533, pp. 48–62. Springer, Heidelberg (2007)
3. Boulanger, D., Bruynooghe, M.: A systematic construction of abstract domains. In: LeCharlier, B. (ed.) SAS 1994. LNCS, vol. 864, pp. 61–77. Springer, Heidelberg (1994)
4. Boulanger, D., Bruynooghe, M., Denecker, M.: Abstracting *s*-semantics using a model-theoretic approach. In: Hermenegildo, M., Penjam, J. (eds.) PLILP 1994. LNCS, vol. 844, pp. 432–446. Springer, Heidelberg (1994)
5. Bruynooghe, M., Janssens, G.: An instance of abstract interpretation integrating type and mode inferencing. In: Kowalski, R., Bowen, K. (eds.) Proceedings of ICLP/SLP, pp. 669–683. MIT Press, Cambridge (1988)
6. Comon, H., Dauchet, M., Gilleron, R., Jacquemard, F., Lugiez, D., Tison, S., Tommasi, M.: Tree Automata Techniques and Applications (1999), `http://www.grappa.univ-lille3.fr/tata`
7. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of the 4th ACM Symposium on Principles of Programming Languages, Los Angeles, pp. 238–252 (1977)
8. Dershowitz, N., Jouannaud, J.-P.: Rewrite systems. In: Handbook of Theoretical Computer Science, Formal Models and Semantics, vol B, pp. 243–320. Elsevier and MIT Press (1990)
9. Feuillade, G., Genet, T., Tong, V.V.T.: Reachability analysis over term rewriting systems. J. Autom. Reasoning 33(3-4), 341–383 (2004)
10. Frühwirth, T., Shapiro, E., Vardi, M., Yardeni, E.: Logic programs as types for logic programs. In: Proceedings of the IEEE Symposium on Logic in Computer Science, Amsterdam (July 1991)
11. Gallagher, J.P., Boulanger, D., Sağlam, H.: Practical model-based static analysis for definite logic programs. In: Lloyd, J.W. (ed.) Proc. of International Logic Programming Symposium, pp. 351–365. MIT Press, Cambridge (1995)
12. Gallagher, J.P., de Waal, D.: Fast and precise regular approximation of logic programs. In: Van Hentenryck, P. (ed.) Proceedings of the International Conference on Logic Programming (ICLP 1994), Santa Margherita Ligure. MIT Press (1994)
13. Gallagher, J.P., Henriksen, K.S.: Abstract domains based on regular types. In: Demoen, B., Lifschitz, V. (eds.) ICLP 2004. LNCS, vol. 3132, pp. 27–42. Springer, Heidelberg (2004)
14. Gallagher, J.P., Henriksen, K.S., Banda, G.: Techniques for scaling up analyses based on pre-interpretations. In: Gabbrielli, M., Gupta, G. (eds.) ICLP 2005. LNCS, vol. 3668, pp. 280–296. Springer, Heidelberg (2005)
15. Gallagher, J.P., Puebla, G.: Abstract interpretation over non-deterministic finite tree automata for set-based analysis of logic programs. In: Krishnamurthi, S., Ramakrishnan, C.R. (eds.) PADL 2002. LNCS, vol. 2257. Springer, Heidelberg (2002)

16. Genet, T., Tong, V.V.T.: Reachability analysis of term rewriting systems with *timbuk*. In: Nieuwenhuis, R., Voronkov, A. (eds.) LPAR 2001. LNCS, vol. 2250, pp. 695–706. Springer, Heidelberg (2001)
17. Heintze, N., Jaffar, J.: A Finite Presentation Theorem for Approximating Logic Programs. In: Proceedings of the 17th Annual ACM Symposium on Principles of Programming Languages, pp. 197–209. ACM Press, San Francisco (1990)
18. Horiuchi, K., Kanamori, T.: Polymorphic type inference in Prolog by abstract interpretation. In: Proc. 6th Conference on Logic Programming. LNCS, vol. 315, pp. 195–214. Springer, Heidelberg (1987)
19. Jacquemard, F.: Decidable approximations of term rewriting systems. In: Ganzinger, H. (ed.) RTA 1996. LNCS, vol. 1103, pp. 362–376. Springer, Heidelberg (1996)
20. Jones, N.: Flow analysis of lazy higher order functional programs. In: Abramsky, S., Hankin, C. (eds.) Abstract Interpretation of Declarative Languages, Ellis-Horwood (1987)
21. Jones, N.D., Andersen, N.: Flow analysis of lazy higher-order functional programs. Theor. Comput. Sci. 375(1-3), 120–136 (2007)
22. Lloyd, J.: Foundations of Logic Programming, 2nd edn. Springer, Heidelberg (1987)
23. Mishra, P.: Towards a theory of types in Prolog. In: Proceedings of the IEEE International Symposium on Logic Programming (1984)
24. Reynolds, J.C.: Automatic construction of data set definitions. In: Morrell, J. (ed.) Information Processing, vol. 68, pp. 456–461. North-Holland, Amsterdam (1969)
25. Ullman, J.: Principles of Knowledge and Database Systems, vol. 1. Computer Science Press (1988)
26. Van Hentenryck, P., Cortesi, A., Le Charlier, B.: Type analysis of Prolog using type graphs. Journal of Logic Programming 22(3), 179–210 (1994)
27. Vaucheret, C., Bueno, F.: More precise yet efficient type inference for logic programs. In: Hermenegildo, M.V., Puebla, G. (eds.) SAS 2002. LNCS, vol. 2477, pp. 102–116. Springer, Heidelberg (2002)
28. Whaley, J., Lam, M.S.: Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In: Pugh, W., Chambers, C. (eds.) PLDI, pp. 131–144. ACM, New York (2004)
29. Whaley, J., Unkel, C., Lam, M.S.: A bdd-based deductive database for program analysis (2004), `http://bddbddb.sourceforge.net/`