Binary Frontier-guarded
ASP with Function Symbols

Mantas Šimkus
Institute of Information Systems
TU Wien, Austria

Abstract. It has been acknowledged that emerging Web applications require features that are not available in standard rule languages like Datalog or Answer Set Programming (ASP), e.g., they are not powerful enough to deal with anonymous values (objects that are not explicitly mentioned in the data but whose existence is implied by the background knowledge). In this paper, we introduce a new rule language based on ASP extended with function symbols, which can be used to reason about anonymous values. In particular, we define binary frontier-guarded programs (BFG programs) that allow for disjunction, function symbols, and negation under the stable model semantics. In order to ensure decidability, BFG programs are syntactically restricted by allowing at most binary predicates and by requiring rules to be frontier-guarded. BFG programs are expressive enough to simulate ontologies expressed in popular Description Logics (DLs), capture their recent non-monotonic extensions, and can simulate conjunctive query answering over many standard DLs. We provide an elegant automata-based algorithm to reason in BFG programs, which yields a $3\text{ExpTime}$ upper bound for reasoning tasks like deciding consistency or cautious entailment. Due to existing results, these problems are known to be $2\text{ExpTime}$-hard.

1 Introduction

Rule-based languages are becoming a major tool to cope with the increasing complexity of available data and knowledge. This is particularly true in applications that query and manage data on the Web. A prime example of a rule-based language is Datalog, which was developed as a recursive query language for relational databases. However, it has been acknowledged that emerging applications on the Web require features that are not available in plain Datalog. In particular, Datalog was designed for closed-world reasoning, i.e., each input database is assumed to be a complete description of the application’s data. Unfortunately, such assumption is often not appropriate for Web applications, where data is likely to be incomplete, e.g., due to missing values or facts.

A significant extension of plain Datalog is Answer Set Programming (ASP), which allows to partially deal with incompleteness. In particular, ASP features disjunction and default negation under the stable model semantics, which enable powerful case-based reasoning and inference based on the lack of information. The presence of these features allows for intelligent management of domain objects under incomplete information. However, plain Datalog and ASP are not powerful enough to deal with missing values,
i.e., objects that are not explicitly mentioned in the data but whose existence is implied by the background knowledge.

One of the possible approaches to deal with missing values is to allow some form of existential quantification in rule heads. In the setting of databases and Datalog, this can be formalized using tuple-generating dependencies, Datalog with value invention (see, e.g., [1, 21]), or Datalog± [9]. In ASP, missing values are usually simulated using function symbols (see Section 7 for a discussion of such examples and further related work). Allowing rules to create new values causes a lot of difficulties; due to the presence of recursion, naive approaches to allow value creation immediately lead to undecidability (this is true already for Horn rules [2]). A prominent approach to regain decidability is to use special atoms to “guard” variables in rule bodies. Examples of this approach are Datalog± in [9] and the frontier-guarded rules in [4]. Importantly, these restrictions are not geared towards limiting recursion, but rather towards ensuring the semantics of a given program can be finitely represented, e.g., by resorting to tree decompositions of infinite structures.

In this paper we show how frontier-guardedness can be used to ensure decidability of ASP with function symbols. Our contributions are as follows:

- We introduce a new fragment of ASP with function symbols, called binary frontier-guarded programs (BFG programs). Such programs allow for disjunction, function symbols, and negation under the stable model semantics. The programs are syntactically restricted by allowing at most binary predicates and by requiring rules to be frontier-guarded.
- BFG programs generalize FNDC and core BD programs [14, 13], and can be used for common-sense reasoning in the presence of a possibly infinite number of domain objects.
- BFG programs allow to simulate ontologies expressed in popular Description Logics (DLs), capture some of their recent non-monotonic extensions, and can simulate conjunctive query answering over many standard DLs.
- We show that BFG programs have the so-called forest-model property, also enjoyed by many standard DLs.
- We provide an elegant automata based procedure for reasoning in BFG programs. In addition to the forest-model property, the algorithm employs a two-world characterization of ASP, reminiscent to the here-and-there approach in [26].
- The construction yields a 3ExpTime upper bound for reasoning tasks like consistency or cautious entailment. These problems are known to be 2ExpTime-hard, e.g., already for positive normal programs [5].

The paper is organized as follows. In Section 2 we recall ASP with function symbols together with the basic notions of automata over infinite trees, which will be our main technical tool. In Section 3 we formally define BFG programs and discuss their features. In Section 4 we show how the stable models of a BFG program can be seen as forests, and then present an encoding of such forests into trees, on which tree automata can run. In Section 5 and 6 we present our automata-based procedure for reasoning in BFG programs. We discuss related work and conclude in Sections 7 and 8, respectively.

BFG programs were first studied in [27], where they were called GT programs.
2 Preliminaries

Answer Set Programming We assume mutually disjoint sets of constants, function symbols, relation (predicate) symbols and variables. Each function and relation symbol \( \sigma \) is associated with a positive integer \( \text{arity}(\sigma) \), called the \emph{arity} of \( \sigma \). A \emph{term} is either a constant, a variable, or an expression of the form \( f(t) \) such that \( f \) is an \( n \)-ary function symbol and \( t \) is an \( n \)-tuple of terms. An \emph{atom} is an expression of the form \( R(t) \) where \( R \) is an \( n \)-ary relation symbol and \( t \) is an \( n \)-tuple of terms. A \emph{(disjunctive) program} \( P \) is any set of \emph{rules} \( r \) of the form

\[
A_1 \lor \ldots \lor A_n \leftarrow A_{n+1}, \ldots, A_m, \text{not } A_{m+1}, \ldots, \text{not } A_k,
\]

where each \( A_j \) is an atom. If \( r \) is of the form \( A \leftarrow \), then \( r \) is a \emph{fact} (often written simply \( A \)). If \( n = 0 \), then \( r \) is a \emph{constraint}. We let \( \text{head}(r) = \{ A_1, \ldots, A_n \} \), \( \text{body}^+(r) = \{ A_{n+1}, \ldots, A_m \} \), and \( \text{body}^-(r) = \{ A_{m+1}, \ldots, A_k \} \). If \( \text{body}^-(r) = \emptyset \), then \( r \) is \emph{positive}. A program \( P \) is \emph{positive}, if all rules of \( P \) are positive. A term, atom, rule or program is \emph{ground}, if it contains no variables. Let \( \mathcal{HU}^P \) be the \emph{Herbrand universe} of \( P \), i.e. the set of terms that can be built from constants and function symbols occurring in a program \( P \). Similarly, \( \mathcal{HB}^P \) is the \emph{Herbrand base} of \( P \), i.e. the set of atoms that can be built from relation symbols of \( P \) and terms in \( \mathcal{HU}^P \). An \emph{interpretation} \( I \) for \( P \) is any set \( I \subseteq \mathcal{HB}^P \). We use \( \text{ground}(P) \) to denote the \emph{grounding} of \( P \), i.e., the set of all ground rules that can be obtained from rules in \( P \) by applying some substitution from variables to terms in \( \mathcal{HU}^P \). An interpretation \( I \) \emph{satisfies} a ground positive rule \( r \), denoted \( I \models r \), if \( \text{body}^+(r) \subseteq I \) implies \( I \cap \text{head}(r) \neq \emptyset \). An interpretation \( I \) is a \emph{model} of a ground positive program \( P \), denoted \( I \models P \), if \( I \) satisfies each rule \( r \in \text{ground}(P) \). A model \( I \) of \( P \) is called \emph{minimal}, if there is no \( J \subseteq I \) such that \( J \) is a model of \( P \). Assume an interpretation \( I \) for a program \( P \). The \emph{GL-reduct} \( P^I \) (see [17]) is the program obtained from \( \text{ground}(P) \) by

(i) removing all rules \( r \) such that \( \text{body}^-(r) \cap I \neq \emptyset \), and

(ii) deleting every expression of the form \text{not } A \) in the remaining rules.

If \( I \) is a minimal model of \( P^I \), then \( I \) is called a \emph{stable model} (or \emph{answer set}) of \( P \). A ground (atomic) query is any ground atom \( A \). A program \( P \) is \emph{bravely} (resp., \emph{cautiously}) \emph{entails} a ground query \( A \), denoted \( P \models_b A \) (resp., \( P \models_c A \)), if \( A \in I \) holds for some (resp., each) stable model \( I \) of \( P \).

Automata over Infinite Trees We recall here finite state automata over infinite trees, which we will use as a tool to reason in BFG programs. In particular, following [29] closely we define here \emph{2-way alternating tree automata}.

A \emph{(full infinite) tree} \( T \) is any set \( T \subseteq \mathbb{N}^* \) of words over the set \( \mathbb{N} \) of positive integers such that \( x \cdot c \in T \), where \( x \in \mathbb{N}^* \) and \( c \in \mathbb{N} \), implies (i) \( x \in T \) and (ii) \( x \cdot c' \in T \) for all \( 0 < c' < c \). Each element \( x \in T \) is a \emph{node} of \( T \), where \( e \) (the empty word) is the root of \( T \). The nodes \( x \cdot c \in T \), where \( c \in \mathbb{N} \), are the \emph{successors} of \( x \). By convention, \( x \cdot 0 = x \) and \((x \cdot i) \cdot (-1) = x \) (note that \( e \cdot (-1) \) is undefined). \( T \) is \emph{k-ary} if each node in \( T \) has \( k \) successors.

An \emph{infinite path} in \( T \) is any set \( p \subseteq T \) of nodes such that (i) \( x \cdot c \in p \) implies \( x \in p \), and (ii) for every \( i \geq 0 \) there is a unique \( x \in p \) such that \(|x| = i \). A \emph{labeled tree} over an
alphabet $\Sigma$ is a tuple $(T, \mathcal{L})$, where $\mathcal{L}: T \rightarrow \Sigma$, i.e., a tree where the nodes are labeled with symbols from $\Sigma$.

For a finite set $V$, let $B(V)$ be the set of formulae that can be built from $V \cup \{\top, \bot\}$ using $\lor$ and $\land$ as connectives. We say that $I \subseteq V$ satisfies $\varphi \in B(V)$, if $I$ is a model of $\varphi$, when elements in $V$ as seen as propositional variables and $\varphi$ as a propositional formula. Let $[k] = \{-1, 0, 1, \ldots, k\}$. A two-way alternating tree automaton (2ATA) over infinite $k$-ary trees is a tuple $A = \langle \Sigma, Q, \delta, q_0, F \rangle$, where $\Sigma$ is an input alphabet, $Q$ is a finite set of states, $\delta: Q \times \Sigma \rightarrow B([k] \times Q)$ is a transition function, $q_0 \in Q$ is an initial state, and $F$ is an acceptance condition.

Assume a 2ATA $A = \langle \Sigma, Q, \delta, q_0, F \rangle$ over $k$-ary trees. A run of $A$ over a $k$-ary labeled tree $(T, \mathcal{L})$ is a labeled tree $(T_r, r)$ over $T \times Q$ that satisfies the following:

(i) $r(\epsilon) = (q_0)$. 
(ii) For each $y \in T_r$, with $r(y) = (x, q)$ and $\delta(q, \mathcal{L}(x)) = \varphi$, there is a set $S = \{(c_1, q_1), \ldots, (c_n, q_n)\} \subseteq [k] \times Q$ such that (i) $S$ satisfies $\varphi$, and (ii) for all $1 \leq i \leq n$, we have that $y \cdot i \in T_r$, $x \cdot c_i$ is defined, and $r(y \cdot i) = (x, c_i, q_i)$. The run $(T_r, r)$ above is accepting, if every infinite path $p \subseteq T_r$ satisfies the acceptance condition $F$ as follows. Let $\text{inf} (p)$ be the set of states $q \in Q$ that occur infinitely often in $p$. A parity acceptance condition $F$ is given by a tuple $F = (G_1, G_2, \ldots, G_m)$ where $G_1 \subseteq G_2 \subseteq \ldots \subseteq G_m$ and $G_m = Q$. Then $p$ satisfies $F$, if an even $i$ exists for which $\text{inf}(p) \cap G_i \neq \emptyset$ and $\text{inf}(p) \cap G_i = \emptyset$. A pairs or a co-pairs acceptance condition is given by a set $F = \{(G_1, R_1), \ldots, (G_n, R_n)\}$ of pairs with $(G_i, R_i) \in 2^Q \times 2^Q$ and $G_i \cap R_i = \emptyset$. Then $p$ satisfies a pairs condition $F$ as above if there is $(G, R) \in F$ such that $\text{inf}(p) \cap G = \emptyset$ and $\text{inf}(p) \cap R \neq \emptyset$. Dually, $p$ satisfies a co-pairs condition $F$ as above if for all $(G, R) \in F$ we have $\text{inf}(p) \cap G \neq \emptyset$ or $\text{inf}(p) \cap R = \emptyset$. An automaton accepts a labeled tree, if there is a run that accepts it. By $L(A)$ we denote the set of trees that $A$ accepts. Unless stated otherwise, by default automata a parity automata, i.e., they have a parity acceptance condition.

We say $A$ is a nondeterministic one-way tree automaton (INTA) if $\delta(q, \sigma)$ is of the form $\delta(q, \sigma) = \left( (1, q^0_1) \land \ldots \land (k, q^0_k) \right) \lor \ldots \lor \left( (1, q^n_1) \land \ldots \land (k, q^n_k) \right)$, for every $q \in Q$ and $\sigma \in \Sigma$. Intuitively, INTAs only move down the tree and with each guess the automaton proceeds with exactly one state for each child node.

2ATAs can be translated into INTAs while preserving the language.

**Theorem 1** ([29]). Let $A$ be a 2ATA with a parity acceptance condition. Then there is a parity INTA $A^n$ such that $L(A) = L(A^n)$. The number of states in $A^n$ is exponential in the number of states in $A$, but the size of the acceptance condition of $A^n$ is linear in the size of the acceptance condition of $A$.

### 3 Binary Frontier-guarded Programs

In this section we define binary frontier-guarded ASP programs with function symbols (BFG programs). Intuitively, they only allow for at most binary relation symbols and at most unary function symbols. In addition, to ensure decidability we require that the
rules are frontier-guarded. As we shall see, these restrictions are not too severe; e.g., BFG programs allows to capture many standard DLs and some of their recent non-monotonic extensions.

**Definition 1.** A BFG program $P$ is a program satisfying the next restrictions.
(1) All ground rules are facts of the form $A(c) \leftarrow$ and $R(c, d) \leftarrow$, where $c, d$ are constants. Constants occur in facts only.
(2) The rules with variables have the following properties:
   (i) atoms have the form $A(x)$, $A(f(x))$, $R(x, y)$, $R(x, f(x))$ or $R(f(x), x)$, where $x \neq y$;
   (ii) (frontier-guardedness) if $r \in P$ and $H \in \text{head}(r)$, then there is $B \in \text{body}^+(r)$ that contains all the variables of $H$.

We first note that BFG programs subsume FDNC programs and core BD programs, which allow for at most two variables in rules [14, 13]. Note that the body of a rule in a BFG program may have the shape of an arbitrary graph. This allows, e.g., to pose a binary Boolean conjunctive query over the stable models of a BFG program. Indeed, a constraint $\leftarrow A_1, \ldots, A_n$, where each $A_i$ is as in (2.i) above is frontier-guarded and thus in the syntax of BFG programs.

**Example 1.** In Figure 1 we present an example of a BFG program. In particular, we consider a publication database, which stores information about publications, authors, editors and venues. The first two rules state that a document $x$ published in a venue $y$ that is known to be a journal or conference is a publication in a known venue. The third rule deals with a possibly missing information about publication venues that are known to exist; for a document $x$ that has an ISBN number but does not have a known publication venue, the rule creates a fresh value for it. The 4th rule states that every document that has a publication venue is a publication. Using the 5th rule we state an author’s profile is incomplete if he/she has a publication in an anonymous venue. Finaly, the 6th rule collects pairs $x, y$ of authors and publications such that $x$ is an editor of the venue in which $y$ is published. Consider the program $P$ that consists of the rules in Figure 1 and includes the set of facts $F = \{ \text{PublishedIn}(p_1, v), \text{Journal}(v), \text{hasISBN}(p_2, n), \text{EditorOf}(a_1, v), \text{AuthorOf}(a_1, p_1), \text{AuthorOf}(a_2, p_2) \}$. It is not difficult to see that $P$ has a single stable model $I = F \cup \{ \text{KnownVenuePub}(p_1), \text{PublishedIn}(p_2, f(p_2)), \text{Published}(p_1), \text{Published}(p_2), \text{IncompleteProfile}(a_2), \text{EditorAuthorship}(a_1, p_1) \}$. 

---

$$
\text{KnownVenuePub}(x) \leftarrow \text{PublishedIn}(x, y), \text{Conference}(y) \\
\text{KnownVenuePub}(x) \leftarrow \text{PublishedIn}(x, y), \text{Journal}(y) \\
\text{PublishedIn}(x, f(x)) \leftarrow \text{hasISBN}(x, y), \text{not} \text{KnownVenuePub}(x) \\
\text{Published}(x) \leftarrow \text{PublishedIn}(x, y) \\
\text{IncompleteProfile}(x) \leftarrow \text{AuthorOf}(x, y), \text{PublishedIn}(y, f(y)) \\
\text{EditorAuthorship}(x, y) \leftarrow \text{EditorOf}(x, z), \text{AuthorOf}(x, y), \text{PublishedIn}(y, z) \\
$$

**Fig. 1.** Example BFG program
Many standard DLs can be seen as fragments of first-order logic. Moreover, DL knowledge bases can be transformed into theories that are syntactically very close to BFG programs. For example, the DL \textit{ALCHI} can be seen as a first-order theory consisting of only the following formulae:

\begin{enumerate}
\item[(DL1)] \(\forall x. (A_1(x) \land \cdots \land A_n(x) \rightarrow A'_1(x) \lor \cdots \lor A'_k(x))\), where \(n \geq 1\) and \(k \geq 0\);
\item[(DL2)] \(\forall x. (A(x) \land R(x,y) \rightarrow A'(y))\);
\item[(DL3)] \(\forall x,y. (R(x,y) \rightarrow R'(x,y))\);
\item[(DL4)] \(\forall x,y. (R(x,y) \rightarrow R'(y,x))\);
\item[(DL5)] \(\forall x. (A(x) \rightarrow \exists y. (R(x,y) \land A'(y)))\);
\item[(DL6)] atomic formula of the form \(A(c)\) or \(R(c,d)\).
\end{enumerate}

More precisely, a general \textit{ALCHI} KB can be transformed into a theory of the above shape while preserving satisfiability and answers to conjunctive queries. The above rules (DL1-DL6) can almost immediately be stated as a BFG program. e.g., (DL1) translates into a rule \(A'_1(x) \lor \cdots \lor A'_k(x) \leftarrow A_1(x), \ldots, A_n(x)\). The formula in (DL5) requires \textit{skolemization}, i.e., we capture it by the rules (i) \(R'(x,f(x)) \leftarrow A(x)\), (ii) \(A'(y) \leftarrow R'(x,y)\), and (iii) \(R(x,y) \leftarrow R'(x,y)\), where \(f\) is a fresh function symbol and \(R'\) is a fresh binary relation symbol. The above translation, which in fact does not employ stable negation, leads to a program that has a model iff the input DL KB has a model. A (constant-free) Boolean conjunctive query over \(K\) can now be expressed by adding a corresponding constraint to the program. In [18] the authors show how to extend DLs of the DL-Lite and \(EL\) families with stable negation, where the semantics is given by a translation into a normal guarded Datalog± program whose existential variables are treated via skolemization. It is easy to see that the target programs used in the translation are a fragment of BFG programs. The presence of disjunction in BFG programs can be used to generalize the proposal of [18] to support DLs that support disjunction, e.g., \textit{ALCHI}.

In the remainder of this paper we show how consistency of BFG programs can be decided by employing tree automata. We concentrate on the existence of stable models because cautious and brave entailment of atomic queries can be reduced in linear time to checking (non)existence of a stable model. We also note that, similarly as for \textit{FDNC} and \textit{BD} programs, decidability of BFG programs can be inferred from the decidability of monadic second-order logic over trees (see, e.g., [12] for an overview). However, we provide a direct automata-based algorithm, that allows us to obtain a 3\textit{ExpTime} upper bound. We build on the method used in [11] for answering (extensions of) conjunctive queries over expressive DLs, but require a non-trivial adaptation to handle frontier-guarded rules and to perform minimality tests as required by the stable model semantics.

\section{Forest-model Property}

We show here that stable models of a BFG program can be seen as forests and describe their encoding into labeled trees, on which automata can run. We assume for the rest of the paper an arbitrary BFG program \(P\), and proceed with the following observation:
**Proposition 1.** If $I$ is a stable model of $P$, then every atom in $I$ is of the form $A(t), R(c,d), R(t,f(t))$ or $R(f(t),t)$, where $c,d$ are constants and $t$ is a term.

**Proof.** Suppose there exists a stable model $I$ of $P$ that violates the above property. Then we can simply remove from $I$ all atoms $W$ that are not of the mentioned forms. Since the rules of $P^I$ are frontier-guarded, removing such a $W$ can not cause a rule in $P^I$ to be violated, hence the resulting interpretation $J$ is a model of $P^I$. This contradicts the assumption that $I$ is a stable model of $P$.

If $P$ has only one constant $c$, then each stable model of $P$ can be seen as a tree, where $c$ is the root and each term $f(t)$ is a child of the term $t$. If $P$ has more than one constant, then a stable model can be viewed as a forest, i.e., a set of trees, where roots correspond to the constants and may be arbitrarily interconnected.

To obtain an automata-based algorithm, we must encode the above forests-shaped interpretations into labeled trees. To this end, let $a_1, \ldots, a_n, f_{n+1}, \ldots, f_m$ be an enumeration of constants and function symbols that appear in $P$, where each $a_i$ is a constant and each $f_j$ is a function symbol. We let $C = \{1, \ldots, n\}$ and $F = \{n + 1, \ldots, m\}$. A word $w \in C \times F^*$ is called a term node. For a term node $w = i \cdot j_1 \cdots j_k$, we let $\text{term}(w) = f_{j_k}(\ldots f_{j_1}(a_i) \ldots)$. Let $L_P$ be the set of unary relation symbols consisting of:

(T1) each unary $A$ that appears in $P$;
(T2) fresh unary $R_f$ and $R^{-}_f$ for each binary $R$ and function $f$ occurring in $P$;
(T3) a fresh unary $R_{c,d}$ for each binary $R$ and constants $c,d$ occurring in $P$.

Intuitively, $R_f$ and $R^{-}_f$ will encode atoms of the form $R(t,f(t))$ and $R(f(t),t)$, respectively, while unary symbols $R_{c,d}$ will encode ground atoms $R(c,d)$.

We let $\Sigma_P = 2^{L_P}$, and call a tree $T = (T, L)$ over $\Sigma_P$ proper, if the following are true for every $n \in T$:

(P1) if $L(n)$ contains some relation of type (T3), then $n = \epsilon$;
(P2) if $L(n)$ contains some relation of type (T1) or (T2), then $n$ is a term node;
(P3) if $L(n) \neq \emptyset$, then $n = \epsilon$ or $n$ is a term node.

Note that the size of $\Sigma_P$ is exponential in the size of $P$. A proper tree $T = (T, L)$ over $\Sigma_P$ is a representation of an interpretation for $P$. Indeed, the root $\epsilon$ of $T$ stores the binary atoms of the form $R(c,d)$. The nodes $1, \ldots, n$ correspond to constants of $P$, and the $F^+$ descendants of such nodes correspond to functional terms. The labeling of nodes provides the relations that are satisfied in the interpretation. More formally, given a proper tree $T = (T, L)$ over $\Sigma_P$, we use $\text{int}(T)$ to denote the interpretation consisting of:

(i) $R(c,d)$, for each $R_{c,d} \in L(\epsilon)$;
(ii) $A(\text{term}(w))$, for each term node $w \in T$ and unary $A \in L(w)$ of type (T1);
(iii) $R(\text{term}(w), f(\text{term}(w)))$ for each term node $w \in T$ with $R_f \in L(w)$;
(iv) $R(f(\text{term}(w)), \text{term}(w))$ for each term node $w \in T$ with $R^{-}_f \in L(w)$.

Observe that for any interpretation $I$ with atoms of the forms given in Proposition 1, we can find a proper $T$ with $\text{int}(T) = I$. Due to Proposition 1, we then know that for any stable model $I$ of $P$ there exists a proper $T$ with $\text{int}(T) = I$. 

5 Outline of the Algorithm

We present here our algorithm for checking the existence of a stable model for $P$. To this end, we will build tree automata running on trees that encode interpretations as well as pairs of interpretations.

We say an automaton $A$ with alphabet $\Sigma_P$ is proper if every tree accepted by $A$ is proper. A proper $A$ with alphabet $\Sigma_P$ accepts an interpretation $I$ for $P$ if there is a proper $T$ such that $\text{int}(T) = I$ and $A$ accepts $T$. We also use trees that represent a pair of interpretations for $P$. Let $T = (T, L)$ be a tree over $\Sigma_P \times \Sigma_P$. We denote by $T|_1 = (T, L_1)$ (resp., $T|_2 = (T, L_2)$) the tree over $\Sigma_P$ such that, for each $n \in T$, $L_1(n)$ (resp., $L_2(n)$) is the first (resp., second) component of $L(n)$. We say that $T$ is proper if $T|_1$ and $T|_2$ are proper. We say that an automaton $A$ with alphabet $\Sigma_P \times \Sigma_P$ is proper if it accepts proper trees only. Such an $A$ accepts an interpretation pair $(I_1, I_2)$ if there is a proper $T$ over $\Sigma_P \times \Sigma_P$ such that $A$ accepts $T$, $I_1 = \text{int}(T|_1)$ and $I_2 = \text{int}(T|_2)$.

To check if $P$ has a stable model, we build an automaton $A^\text{sm}_P$ that accepts exactly the proper trees $T$ such that $\text{int}(T)$ is a stable model of $P$. In other words, the program $P$ has a stable model iff the automaton $A^\text{sm}_P$ is nonempty, i.e., accepts some tree. We build $A^\text{sm}_P$ by manipulating the following simpler automata.

**Proposition 2.** The following proper parity $\text{INTA}$ can be constructed:

(a) $A^\text{p}^\cup_P$ that accepts exactly the pairs $(I, I')$ such that $I \not\models P I'$. The number of states in $A^\text{p}^\cup_P$ is exponential in the size of $P$, while the acceptance condition is of polynomial size in the size of $P$.

(b) $A^\text{p}^\cup_P$ that accepts exactly the pairs $(I, I')$ such that $I \not\models I'$. The automaton $A^\text{p}^\cup_P$ has a fixed number of states and an acceptance condition of fixed size.

(c) $A^\text{p}_P$ that accepts exactly the pairs $(I, I')$ such that $I = I'$. The automaton $A^\text{p}_P$ has a fixed number of states and an acceptance condition of fixed size.

The precise construction of $A^\text{p}^\cup_P$, $A^\text{p}^\cup_P$, and $A^\text{p}_P$ is presented in Section 6. By manipulating these automata we can obtain the desired automaton $A^\text{sm}_P$.

(1) We construct an automaton $A_1$ by complementing $A^\text{p}^\cup_P$ and intersecting the resulting automaton with $A^\text{sm}_P$, i.e., $L(A_1) = L(A^\text{p}^\cup_P) \cap L(A^\text{sm}_P)$. Then $A_1$ accepts pairs of interpretations $(I, I')$ such that $I = I'$ and $I \models P I'$. We can use the results in [25] for the complementation step. Measured in the size of $P$, the automaton $A_1$ has at most double exponential number of states and a co-pairs acceptance condition with exponentially many pairs.

(2) We let $A^\text{mod}_P$ be an automaton accepting trees obtained by projecting away the first interpretation in the language of $A_1$. That is, $A^\text{mod}_P$ accepts a tree $T'$ iff there exists a tree $T$ over $\Sigma_P \times \Sigma_P$ such that $T|_2 = T'$ and $A_1$ accepts $T$. Due to the construction of $A_1$, we then get that $A^\text{mod}_P$ accepts an interpretation $I$ iff $I \models P I'$. The construction of $A^\text{mod}_P$ is fairly standard. Assume $A_1 = (\Sigma_P \times \Sigma_P, Q, \delta, q_0, F)$. We simply define $A^\text{mod}_P = (\Sigma_P, Q, \delta', q_0, F)$, where $\delta'$ is as follows. For each $N' \in \Sigma_P$ and each state $q \in Q$, we have $\delta'(N', q) = \bigvee_{N \in \Sigma_P} \delta((N, N'), q)$. Note that this construction does not modify the state set or the co-pairs acceptance condition of $A_1$. 
(3) We construct an automaton $A_2$ that accepts the language $L(A_2) = L(A'_p) \cup L(A'^*_p)$. In other words, $A_2$ accepts a pair $(I, I')$ iff $I \subset I'$ implies $I \not\models P'I'$. The automaton $A_2$ requires at most exponentially many states and a parity condition that is of polynomial size in the size of $P$.

(4) We construct an automaton $A_3$ that accepts a pair $(I, I')$ iff $I \subset I'$ and $I \models P'I'$. This construction simply complements the automaton $A_2$. Using the results of [5] and measured in the size of $P$, the automaton $A_3$ has at most double exponential number of states and a co-pairs acceptance condition with exponentially many pairs.

(5) We let $A_4$ be an automaton accepting trees obtained by projecting away the first interpretation in the language of $A_3$. That is, $A_4$ accepts a tree $T$ iff there exists a tree $T'$ over $\Sigma_P \times \Sigma_P$ such that $T|_2 = T'$ and $A_3$ accepts $T$. Due to the construction of $A_3$, we then get that $A_4$ accepts an interpretation $I'$ iff there exists $I \subset I'$ such that $I \models P'I'$. The construction of $A_4$ is identical to the construction of $A'^{\text{mods}}_p$ from $A_1$ and does not modify the state set or the co-pairs acceptance condition of $A_3$.

(6) We construct an automaton $A'^{\text{min}}_p$ that accepts a tree $T'$ over $\Sigma_P \times \Sigma_P$ with $T|_2 = T'$ we have that $A_2$ accepts $T$. In other words, $A'^{\text{min}}_p$ accepts an interpretation $I'$ iff $I \models P'I'$ holds for all $I \subset I'$. The automaton $A'^{\text{min}}_p$ is a 1NTA obtained by employing the complementation of $A_3$. Again, using the results of [25], $A'^{\text{min}}_p$ is a 1NTA with at most triple exponential number of states and a pairs condition with doubly exponentially many pairs, measured in the size of $P$.

(7) Finally, we construct an automaton $A'^{\text{sm}}_p$ by intersecting $A'^{\text{min}}_p$ with the automaton $A'^{\text{mods}}_p$. That is, the automaton $A'^{\text{sm}}_p$ accepts the language $L(A'^{\text{sm}}_p) = L(A'^{\text{min}}_p) \cap L(A'^{\text{mods}}_p)$. We have that $A'^{\text{sm}}_p$ accepts an interpretation $I'$ iff $I' \models P'I'$ and there is no $I \subset I'$ with $I \models P'I'$. The 1NTA $A'^{\text{sm}}_p$ requires at most triple exponential number of states and a pairs condition with doubly exponentially many pairs, measured in the size of $P$.

Due to the above construction, consistency of $P$ can be decided by checking non-emptiness of $A'^{\text{sm}}_p$.

**Theorem 2.** $P$ has a stable model iff the language of $A'^{\text{sm}}_p$ is non-empty.

Overall, the automaton $A'^{\text{sm}}_p$ has a triple exponential number of states and a pairs acceptance condition with doubly exponentially many pairs in the size of $P$. Due to [15], testing emptiness of $A'^{\text{sm}}_p$ is feasible in triple exponential time in the size of $P$.

**Theorem 3.** Checking consistency of BFG programs is in $3\text{ExpTime}$.

We do not know whether the above upper bound is worst-case optimal, but we know that the problem is $2\text{ExpTime}$-hard. This is already true for core BFG programs, which is a fragment of BFG programs [27]. We note that $2\text{ExpTime}$-hardness already holds for positive normal BFG programs due to [5]. An yet another way to see the lower bound is a straightforward reduction from the conjunctive query entailment problem in the DL $\mathcal{ALC}$, which was shown to be $2\text{ExpTime}$-hard in [22] (the reduction only requires positive disjunctive BFG programs).
6 Automata Constructions

In this section we prove Proposition 2, i.e., show how to build the automata \(A_P^\text{\(prop\)}, A_P^=, A_P^\prec\) and \(A_P^\preceq\). Before we begin, note that we can easily construct a 1NTA \(A_P^\text{\(prop\)}\) that accepts a tree \(T\) over \(\Sigma_P \times \Sigma_P\) iff \(T\) is proper. Such an automaton only requires a constant number of states and an acceptance condition of fixed size.

The automata \(A_P^=\) and \(A_P^\prec\). We now proceed with the construction of the automata \(A_P^=\) and \(A_P^\prec\) for checking the equality or a violation of strict containment between interpretations, respectively. We start by constructing two alternating automata \(A_0^=\) and \(A_0^\prec\), and then we transform them into the desired 1NTAs. We let

\[
A_0^= = (\Sigma_P \times \Sigma_P, \{q^=\}, \delta, q^=, F),
\]

where \(F = (\emptyset, \{q^=\})\) is a parity acceptance condition, and \(\delta\) is as follows. For each \((N, N') \in \Sigma_P \times \Sigma_P\),

\[
\delta((N, N'), q^=) = [N = N'] \land \bigwedge_{i \in C(i,P)}(i, q^=).
\]

Here \([\text{cond}]\) stands for \(\top\) if \(\text{cond}\) is true and for \(\bot\) if \(\text{cond}\) is false. We let

\[
A_0^\prec = (\Sigma_P \times \Sigma_P, \{q^\prec\}, \delta, q^\prec, F),
\]

where \(F = (\{q^\prec\})\) is a parity acceptance condition, and \(\delta\) is as follows. For each \((N, N') \in \Sigma_P \times \Sigma_P\),

\[
\delta((N, N'), q^\prec) = [N \not\subset N'] \lor \bigvee_{i \in C(i,P)}(i, q^\prec).
\]

We construct a union automaton \(A_0^\prec = A_0^\prec \cup A_0^=\). The desired automata \(A_P^=\) and \(A_P^\prec\) are obtained by transforming \(A_0^=\) and \(A_0^\prec\), respectively, into proper 2ATAs (i.e., intersecting them with \(A_P^\text{\(prop\)}\)) and then into 1NTAs (in fact, it is not hard to see that 2-wayness and alternation are not really needed in these automata). Both automata \(A_P^=\) and \(A_P^\prec\) have boundedly many states and a bounded acceptance condition.

The automaton \(A_P^\preceq\). The remainder of this section is devoted to constructing the automaton \(A_P^\preceq\) that accepts a pair \((I, I')\) iff \(I \not\preceq I'\). This construction is the most involved one. It requires some auxiliary automata and requires the definition of another kind of trees. Let \(X\) be the set of variables occurring \(P\). We let \(\Sigma = 2^X \times \Sigma_P \times \Sigma_P\). Intuitively, a tree \(T\) over \(\Sigma\) represents a pair \((I, I')\) of interpretations where, additionally, the variables of \(P\) are assigned to some terms. Our first step is to define an automaton \(A_P^\preceq\) that ensures that in a tree \(T = (T, \mathcal{L})\) over \(\Sigma\) every variable is assigned to exactly one node, i.e., the tree encodes a function \(\pi\) from \(X\) to \(T\). In the second step we define another automaton \(A\) that verifies whether the given variable assignment witnesses \(I \not\preceq I'\). In the third and final step, we use \(A_P^\preceq\) and \(A\) to obtain \(A_P^\preceq\).

Step 1. We define the automaton \(A_P^\preceq = (\Sigma, Q, \delta, q_0, F)\) to ensure that in a tree \(T = (T, \mathcal{L})\) over \(\Sigma\) every variable is assigned to exactly one node.

The state set \(Q\) of \(A_P^\preceq\) consists of an initial state \(q_0\) and the states \(q_x, q'_x, q^\preceq_x\) and \(q^\prec_x\) for each variable \(x\) of \(P\). Intuitively, the automaton uses \(q_x\) to verify that some node is labeled with \(x\), and uses the state \(q'_x\) to verify that \(x\) is neither in the labeling of the current symbol, nor in the labeling of any descendant. The states \(q^\preceq_x\) and \(q^\prec_x\) verify the presence or absence of the variable \(x\) in the labeling of the current node, respectively.
The transition function $\delta$ is as follows. From the initial state the automaton switches to states $q_x$ for each variable $x \in X$, i.e., for each $\sigma \in \Sigma$, we have $\delta(\sigma, q_0) = \wedge_{x \in X}(0, q_x)$.

When in state $q_x$, the automaton either decides to place the variable in the current node, or chooses a branch where it will be placed. After placing the variable, it enters the state $q'_x$ to ensure that a variable does not occur more than once. This is implemented by the following transition for each $\sigma \in \Sigma$ and variable $x \in X$:

$$\delta(\sigma, q_x) = \left( (0, q_x^\Sigma) \land \bigwedge_{i \in C \cup F} (i, q_x^i) \right) \lor \left( \bigvee_{i \in C \cup F} ((i, q_x) \land \bigwedge_{j \in C \cup F, j \neq i} (j, q_x^j)) \right),$$

$$\delta(\sigma, q'_x) = \left( (0, q_x^\Sigma) \land \bigwedge_{i \in C \cup F} (i, q_x^i) \right).$$

The transitions for $q_x^\Sigma$ and $q_x^{\bar{\Sigma}}$ are simple. We let $\delta(\sigma, q_x^\Sigma) = [x \in V]$ and $\delta(\sigma, q_x^{\bar{\Sigma}}) = [x \notin V]$ for each $\sigma = (V, N, N')$ in $\Sigma$ and variable $x \in X$.

Finally, we need to ensure that each variable is eventually placed in the tree by prohibiting the states $q_x$ from occurring infinitely often. For this, we simply take the acceptance condition $F = (\{q_x \mid x \in X\}, Q)$.

**Step 2.** Now we build the automaton $A$ that verifies whether a given variable assignment $\pi$ witnesses $I \not\models P^I$. More precisely, we assume a given tree $T = (T, \mathcal{L})$ over $\Sigma$ such that $T$ represents an assignment $\pi$ of variables to nodes of the tree (i.e., each query variable $x$ occurs in the label of exactly one node $\pi(x) \in T$) together with a pair of interpretations $(I, I')$. We construct an automaton $A$ such that $A$ accepts $T$ if and only if $\sigma$ witnesses $I \not\models P^I$, that is, if under the assignment $\pi$ the atoms of its positive body are true in $I$, the atoms of its negative body are false in $I'$, and the atoms in its head are false in $I$.

The automaton $A = (\Sigma, Q, \delta, q_0, F)$ is defined as follows. The state set $Q$ is as follows:

$$Q = \{q_{W}^A, q_{W}^F, q_{W}^{f^A}, q_{W}^{f^F}, q_{W}^{+}, q_{W}^{-}, q_{W}^{f^A^{+}}, q_{W}^{f^F^{+}} \mid W \text{ is an atom occurring in } P\} \cup \{q_{A}^W, q_{A}^{f^A}, q_{A}^{f^F} \mid A \text{ is a unary predicate name occurring in } P\} \cup \{q_{W}^{R(x)}, q_{W}^{R(x,y)}, q_{W}^{f^R(x)}, q_{W}^{f^R(x,y)}, q_{W}^{f^R(x,y)} \mid R(x, y) \text{ is from } P\} \cup \{q_{x} \mid x \text{ is a variable from } P\}.$$

We next explain how the transition function is defined.

(I) The state set $Q$ contains $q_{W}^{A}$, $q_{W}^{f^A}$ and $q_{W}^{f^F}$ for each atom $W$ occurring in $P$. Intuitively, $A$ moves to $q_{W}^{A}$, $q_{W}^{f^A}$ or $q_{W}^{f^F}$, to verify that under the assignment $\pi$ the atom $W$ is true in $I$, false in $I'$, or false in $I$, respectively.

From the initial state $q_0$, the automaton nondeterministically chooses a rule $r \in P$ and verifies that it is violated, by moving to $q_{W}^{A}$ for each positive body atom $W$, to $q_{W}^{f^A}$ for each negative body atom $W$, and to $q_{W}^{f^F}$ for each head atom $W$. Hence, for each $\sigma \in \Sigma$, we have:

$$\delta(\sigma, q_0) = \bigvee_{r \in P} \left( \bigwedge_{W \in \text{body}_+^+(r)} (0, q_{W}^A) \land \bigwedge_{W \in \text{body}_-^-(r)} (0, q_{W}^{f^A}) \land \bigwedge_{W \in \text{head}_+(r)} (0, q_{W}^{f^F}) \right).$$
It only remains to implement the transitions for \( q_{W}^{f} \), \( q_{W}^{G} \) and \( q_{W}^{w} \).

(II) The transitions for \( q_{W}^{f} \) use the states \( q_{i}^{f} \) to check that, at the current position in the tree, the atom \( W \) is satisfied. The transitions from the state \( q_{W}^{f} \) depend on the form of the atom \( W \). For ground atoms they are simple. Recall that we store binary ground atoms \( R_{c,d} \) in the label of the root, and that unary atoms \( A(c) \) are represented by the symbol \( A \) in the label of the term node \( i \) with \( c = a_{i} \). Hence, to verify the satisfaction of \( R(c, d) \) we simply look for the corresponding symbol at the root. If the atom is unary, we use the auxiliary state \( q_{A}^{f} \) to check that the labeling of the corresponding term node contains \( A \). For non-ground atoms the automaton non-deterministically navigates to some node of the tree. Then it uses the state \( q_{W}^{w} \) to test there the satisfaction of \( W \).

First, depending on the type of \( W \), we let for each \( \sigma = (V, N, N') \) in \( 2^{X} \times \Sigma_{P} \times \Sigma_{P'} \):

\[
\delta(\sigma, q_{W}^{f}) = \begin{cases} 
(0, q_{W}^{f}) \lor \bigvee_{i \in \mathcal{C} \cup \mathcal{F}(i)} (i, q_{W}^{f}) & \text{if } W \text{ is not ground}, \\
[R_{c,d} \in N] & \text{if } W = R(c, d), \\
(i, q_{A}^{f}) & \text{if } W = A(c) \text{ and } c = a_{i},
\end{cases}
\]

and for all \( (V, N, N') \in \hat{\Sigma} \) and unary \( A \) of \( P \), we let \( \delta(\sigma, q_{A}^{f}) = [A \in N] \).

For the case where \( W \) is not ground, we also define transitions from the state \( q_{W}^{w} \), which again depend on the form of the atom \( W \). In case \( W \) is unary, for each \( \sigma = (V, N, N') \) in \( \hat{\Sigma} \), we let:

\[
\delta(\sigma, q_{W}^{w}) = \begin{cases} 
[A \in N \text{ and } x \in V] & \text{if } W = A(x), \\
[x \in V] \land (i, q_{A}^{f}) & \text{if } W = A(f(x)) \text{ and } f = f_{i}.
\end{cases}
\]

If \( W \) is binary with a function symbol (i.e., if \( W = R(x, f(x)) \) or \( W = R(f(x), x) \)), we define, for each \( \sigma = (V, N, N') \) in \( \hat{\Sigma} \):

\[
\delta(\sigma, q_{W}^{w}) = \begin{cases} 
[R_{f} \in N \text{ and } x \in V] & \text{if } W = R(x, f(x)) \\
[R_{f} \in N \text{ and } x \in V] & \text{if } W = R(f(x), x).
\end{cases}
\]

For atoms \( R(x, y) \) it is a bit more complicated. For all \( (V, N, N') \in \hat{\Sigma} \) and \( W = R(x, y) \), we have:

\[
\delta(\sigma, q_{W}^{w}) = (0, q_{W}^{f}) \lor \left( [x \in V] \land \left( \bigvee_{i \in \mathcal{C} \cup \mathcal{F}(i)} ([R_{f_{i}} \in N] \land (i, q_{y})) \right) \right) \lor \\
\left( [x \in V] \land (-1, q_{y}) \land (-1, q_{W}^{w}) \right)
\]

Intuitively, the three disjuncts verify the three possible ways in which an atom \( R(x, y) \) can be satisfied: (i) \( x \) and \( y \) are assigned to constants, (ii) \( y \) is mapped to a functional successor of \( \pi(x) \), and (iii) \( x \) is mapped to a functional successor of \( \pi(y) \). In the first disjunct, the automaton moves to the auxiliary state \( q_{W}^{f} \) to verify whether there is a pair of constants witnessing the satisfaction of the atom \( R(x, y) \), i.e., whether there is a pair \( c, d \) such that \( x \) is assigned to \( c \), \( y \) is assigned to \( d \), and \( R(c, d) \) holds; recall that the latter is stored at the label of the root. Hence we have, for each \( \sigma = (V, N, N') \) in \( \hat{\Sigma} \):

\[
\delta(\sigma, q_{W}^{f}) = \bigvee_{i,j \subseteq \mathcal{C}} ([R_{a_{i},a_{j}} \in N] \land (i, q_{x}) \land (j, q_{y}))
\]
Finally, for \( q_x \) and \( q_{(R,x)}^f \) we have \( \delta(\sigma, q_x) = [x \in V] \) and \( \delta(\sigma, q_{(R,x)}^f) = \bigvee_{i \in F} R_i \notin N \) for all \( \sigma = (V, N, N') \) in \( \hat{\Sigma} \).

(III) The transitions for \( q_R^f \) are analogous, but tests \([s \notin N]\) for a symbol \( s \in L_P\), is replaced by the test \([s \notin N]\), and we use the states super-indexed with \( f \) instead of their \( t \) counterparts \( (q_W^f, q_W^f, q_W^f, \text{ etc.}) \).

(IV) Similarly, in the transitions for \( q_R^f \) we test for \([s \notin N']\) and use the states super-indexed with \( f' \).

In the acceptance condition, we only need to prohibit the states \( q_W^f, q_W^f, q_W^f, \text{ etc.} \), which can postpone the tests for the truth or falsity of atoms, from occurring infinitely often. Hence we set \( F = \{ q_W^f, q_W^f, q_W^f, W \text{ is an atom in } P \} \), \( Q \).

**Step 3.** We can finalize the construction of \( A_P^{\#} \). First we let \( B = (\hat{\Sigma}, Q, \delta, q_0, F) \) be the result of translating the intersection automaton \( A \cap A_P^{\#} \) into a 1NTA. The state set of \( B \) is exponential in \( P \), and its parity condition is of polynomial size. To obtain \( A_P^{\#} \), we first obtain \( B' \) by projecting away the variable assignment in the first component of the labels. That is, \( B' = (\Sigma_P \times \Sigma_P, Q, \delta', q_0, F) \) where for each \( (N, N') \in \Sigma_P \times \Sigma_P \) and each state \( q \in Q \),

\[
\delta'((N, N'), q) = \bigvee_{V \in 2^N} \delta((V, N, N'), q).
\]

The automaton \( B' \) accepts a tree \( T \) over \( \Sigma_P \times \Sigma_P \) iff \( T \) can be decorated with variables in a way that the resulting tree \( T' \) over \( \hat{\Sigma} \) is accepted by \( B \). Finally, the automaton \( A_P^{\#} \) is obtained by transforming \( B' \) into a proper automaton, by intersecting it with \( A_P^{\#} \).

This involves a linear increase in the number of states, and hence the state set of \( A_P^{\#} \) remains exponential and the parity condition of polynomial size. The automaton \( A_P^{\#} \) accepts exactly the pairs \( (I, I') \) such that \( I \not\models P' \), as required.

### 7 Related Work

Since ASP with function symbols is highly undecidable, e.g., checking existence of a stable model lies at the second level of the analytical hierarchy [24], many authors have suggested ways to reduce the complexity of reasoning. To this end, “mild” restrictions were considered in [7, 6, 10] to obtain fragments that are very expressive and computationally better behaved (e.g., obtaining semi-decidability). Unfortunately, reasoning in these fragments is either not decidable, or checking whether a program belongs to a given fragment is undecidable. Another approach is to consider various acyclicity notions, with \( \omega \)-restricted programs of [28] being one the first approaches. See, e.g., [20, 8] and the references therein for the recent works in this direction. They ensure decidability by guaranteeing finiteness (and a relatively small size) of stable models of a program. In contrast, BFG programs may have infinite stable models and thus are in line with [13, 14, 16], where efficiently verifiable restrictions are used to ensure that the possibly infinite stable models are forest-shaped.

The presence of negation is not the only cause of undecidability: basic reasoning is undecidable already for Horn programs with existentially quantified variables in rule
Ensuring decidability by requiring rules to be *guarded* was first proposed by Calì et al. [9]. Here “guarded” means that each rule is required to have a positive body atom that contains all universal variables of a given rule. The authors also relax this condition to “weak guardedness”, which excludes from guarding the variables that can be safely assumed to range over constants. The notion of frontier-guarded rules, which generalizes guarded rules, was proposed in [4]. Further generalization of guarded and frontier-guarded rules were considered in [5]. The recent work in [18] adds to guarded rules negation under the stable model semantics. Our BFG programs are incomparable to the fragments of [18] as we consider predicates of arity at most 2, but allow for disjunction and non-guarded rules. Adding stable negation to existential rules in combination with various acyclicity notions was recently considered in [23, 3].

8 Discussion

In this paper we have introduced BFG programs, which is a new decidable fragment of ASP with function symbols. Understanding whether the provided 3ExpTime upper bound is worst-case optimal is left for future work. We believe that, using word automata instead of tree automata, the 3ExpTime upper bound for general BFG programs can be recast to show a 2ExpSpace upper bound BFG programs that allow for a single function symbol.

An important issue for future research is to characterize the data complexity of BFG programs, i.e. the complexity measured in the size of program facts. Unfortunately, automata based techniques, including the one used in this paper, don’t seem to be adequate for characterizing data complexity as often too much structure is lost when encoding desired structures into labeled trees. In the future we also plan to investigate the possibility of rewriting BFG programs into ASP programs without function symbols, similarly to the approach of [19] to rewrite existential frontier-guarded rules into plain Datalog.

Acknowledgments

This work has been supported by the Austrian Science Fund (FWF) grants P20840 and P25207, and the Vienna Science and Technology Fund (WWTF) project ICT12-15. The author is grateful to Thomas Eiter for all the inspiring discussions on the topic.

References