Generic Tableaux for Answer Set Programming

Martin Gebser and Torsten Schaub[⋆]

Institut für Informatik, Universität Potsdam, August-Bebel-Str. 89, D-14482 Potsdam, Germany

Abstract. We provide a general and modular framework for describing inferences in Answer Set Programming (ASP) that aims at an easy incorporation of additional language constructs. To this end, we generalize previous work characterizing computations in ASP by means of tableau methods. We start with a very basic core fragment in which rule heads and bodies consist of atomic literals. We then gradually extend this setting by focusing on the concept of an aggregate, understood as an operation on a collection of entities. We exemplify our framework by applying it to conjunctions in rule bodies, cardinality constraints as used in *smodels*, and finally to disjunctions in rule heads.

1 Introduction

Answer Set Programming (ASP; [1]) has become an appealing tool for declarative problem solving. Unlike the related area of satisfiability checking (SAT; [2]), it however lacked a formal framework for describing inferences conducted by ASP solvers, such as the resolution proof theory in SAT [3]. This deficiency has led to a great heterogeneity in the description of ASP algorithms and has impeded their formal comparability. We addressed this problem in [4] by introducing a family of tableau calculi [5] for ASP. The idea is to view answer set computations as derivations in an inference system: A branch in a tableau corresponds to a successful or unsuccessful computation of an answer set; an entire tableau represents a traversal of the search space. This approach provided a uniform proof-theoretic framework for analyzing and comparing different operations, strategies, or even algorithms of ASP solvers. Among others, we related the approaches of existing ASP solvers to appropriate tableau calculi, in the sense that computations of solvers are described by tableaux in corresponding calculi.

In this work, our primary goal is to generalize this approach towards a flexible and modular framework that is easily amenable to new language constructs. We begin with characterizing inferences in a basic core fragment in which rule heads and bodies consist of atomic literals. We then gradually extend this setting by focusing on *aggregates*. An aggregate is understood as an operation on a collection of entities. To obtain a basic understanding of how to describe inferences on aggregates, we view conjunctions in rule bodies as simple Boolean aggregates. We then extend the framework to cardinality constraints, as used in *smodels* [6], and to disjunctions in rule heads.

After establishing the formal background, we introduce in Section 3 our generic tableau framework. In Section 4, 5, and 6, we gradually extend this framework with

[⋆] Affiliated with the School of Computing Science at Simon Fraser University, Burnaby, Canada, and IIIS at Griffith University, Brisbane, Australia.

conjunctions, cardinality constraints, and disjunctions, respectively. Section 7 is dedicated to the proof complexity associated with these constructs. Finally, we discuss our approach in Section 8.

2 Background

We are interested in an extensible proof-theoretic framework dealing with logic programs incorporating composite language constructs, like aggregates. To this end, we need a semantic account of answer sets being, on the one hand, flexible and general enough to accommodate a variety of composite language constructs, and, on the other hand, conservative enough to correspond to standard answer set semantics on well-established language fragments. Among several proposals [6–10], we have chosen Paolo Ferraris' approach [10] that deals with propositional theories and gives meaning to complex constructs by mapping them into propositional formulas.

We start by recalling Ferraris' answer set semantics [10]. A propositional theory is a finite set of propositional formulas, constructed of atoms from an alphabet $\mathcal{P}$ and the connectives $\bot$, $\wedge$, $\vee$, and $\rightarrow$. Any other connective is considered as an abbreviation, in particular, $\neg\phi$ stands for $(\phi \rightarrow \bot)$. An interpretation X, represented by the set of atoms true in X, is a model of a propositional theory Φ if $X \models \phi$ for all $\phi \in \Phi$. The *reduct*, Φ^X, of Φ wrt X is a propositional theory, (recursively) defined as follows:

$$\Phi^X = \{\phi^X \mid \phi \in \Phi\}$$
$$\phi^X = \begin{cases} \bot & \text{if } X \not\models \phi \\ \phi & \text{if } \phi \in X \\ \phi_1^X \circ \phi_2^X & \text{if } X \models \phi \text{ and } \phi = \phi_1 \circ \phi_2 \text{ for } \circ \in \{\wedge, \vee, \rightarrow\} \end{cases}$$

Intuitively, all (maximal) subformulas of Φ that are false in X are replaced by $\bot$ in Φ^X, all other subformulas of Φ stay unchanged. It is clear that any model of Φ^X is also a model of Φ, since Φ^X contains $\bot$ if $X \not\models \phi$ for some $\phi \in \Phi$. Also note that all occurrences of negation, that is, subformulas of the form $(\phi \rightarrow \bot)$, are replaced by constants in Φ^X, since either ϕ or $(\phi \rightarrow \bot)$ is false in X. An interpretation X is an *answer set* of a propositional theory Φ if X is a minimal model of Φ^X.

We next consider logic programs. Given an alphabet $\mathcal{P}$, a *logic program* is a finite set of *rules* of the form $\alpha \leftarrow \beta$ where α and β are *literals*, understood here as *expressions* over $\mathcal{P}$ possibly preceded by the negation as failure operator not. In the following sections, we gradually refine heads α and bodies β for obtaining particular classes of logic programs. The semantics of logic programs is given by the answer sets of corresponding propositional theories, obtained via particular translations to be provided in the following sections. However, the proof-theoretic characterizations below apply directly to logic programs, without translating them into propositional theories.

We describe calculi for the construction of answer sets from logic programs. Such constructions are associated with an (initial) assignment and a binary tree called a tableau [5]. An *assignment* A is a partial mapping from the expressions in a program into $\{\boldsymbol{T}, \boldsymbol{F}\}$, indicating whether a member of the *domain* of A, denoted by $dom(A)$, is true or false, respectively. The domain of A, $dom(A)$, varies throughout this paper depending on the considered language fragment. We define $A^{\boldsymbol{T}} = \{v \in \mathit{dom}(A) \mid A(v) = \boldsymbol{T}\}$ and $A^{\boldsymbol{F}} = \{v \in \mathit{dom}(A) \mid A(v) = \boldsymbol{F}\}$. We also denote A by a set of signed expressions: $\{\boldsymbol{T}v \mid v \in A^{\boldsymbol{T}}\} \cup \{\boldsymbol{F}v \mid v \in A^{\boldsymbol{F}}\}$. Furthering this notation, we call an assignment that leaves all expressions undefined *empty* and denote it by $\emptyset$.

The nodes of tableaux are (mainly) *signed expressions*, that is, expressions preceded by either $\boldsymbol{T}$ or $\boldsymbol{F}$, indicating an assumed truth value. A *tableau* for a logic program Π and an initial assignment A is a binary tree such that the root node of the tree consists of the rules in Π and all members of A. The other nodes in the tree are *entries* of the form $\boldsymbol{T}v$ or $\boldsymbol{F}v$, where $v \in \mathit{dom}(A)$, generated by extending a tableau using tableau rules (given in the following sections) in the standard way [5]: Given a tableau rule and a branch in the tableau containing the prerequisites of the rule, the tableau can be extended by adding new entries to the end of the branch as specified by the rule. If the rule is the cut rule (cf. *(g)* in Figure 1), then entries $\boldsymbol{T}v$ and $\boldsymbol{F}v$ are added as the left and the right children to the end of the branch. For the other rules, the consequents are added to the end of the branch. For convenience, the representation of tableau rules makes use of two conjugation functions, $\boldsymbol{t}$ and $\boldsymbol{f}$. For a literal l, define:

$$\boldsymbol{t}l = \begin{cases} \boldsymbol{T}l & \text{if } l \in \mathit{dom}(A) \\ \boldsymbol{F}v & \text{if } l = \mathit{not}\ v \text{ for } v \in \mathit{dom}(A) \end{cases} \qquad \boldsymbol{f}l = \begin{cases} \boldsymbol{F}l & \text{if } l \in \mathit{dom}(A) \\ \boldsymbol{T}v & \text{if } l = \mathit{not}\ v \text{ for } v \in \mathit{dom}(A) \end{cases}$$

Some rule applications are subject to provisos. For instance, $(v \in \Gamma)$ guides the application of the cut rule by restricting cut objects to members of Γ (cf. Figure 1).

A *tableau calculus* $\mathcal{T}$ is a set of tableau rules. An entry $\boldsymbol{T}v$ (or $\boldsymbol{F}v$) can be deduced in a branch if $\boldsymbol{T}v$ (or $\boldsymbol{F}v$) can be generated from nodes in the branch by applying rules of $\mathcal{T}$. Note that any *branch* corresponds to a pair (Π, A) consisting of a program Π and an assignment A; we draw on this relationship for identifying branches in the sequel. A branch in a tableau is *contradictory* if it contains both entries $\boldsymbol{T}v$ and $\boldsymbol{F}v$ for some $v \in \mathit{dom}(A)$. A branch is *complete* if it is contradictory or if it is closed under all rules in $\mathcal{T}$, except for the cut rule, and either $\boldsymbol{T}v \in A$ or $\boldsymbol{F}v \in A$ for each $v \in \mathit{dom}(A)$. A tableau is complete if all its branches are complete. A complete tableau for a program Π and the empty assignment $\emptyset$ such that all branches are contradictory is a *refutation* for Π; provided that $\mathcal{T}$ is a complete calculus, it means that Π has no answer set.

3 Generic Tableau Rules

We begin with a simple class of *unary programs* where rules $\alpha \leftarrow \beta$ are restricted to *atomic literals*, that is, α and β are either equal to p or $\mathit{not}\ p$ for some atom $p \in \mathcal{P}$. The semantics of a unary program Π is given by the answer sets of a propositional theory, $\tau[\Pi]$, (recursively) defined as follows:

$$\tau[\Pi] = \{\tau[\beta] \rightarrow \tau[\alpha] \mid (\alpha \leftarrow \beta) \in \Pi\} \tag{1}$$

$$\tau[\pi] = \begin{cases} \neg\tau[v] & \text{if } \pi = \mathit{not}\ v \\ \pi & \text{if } \pi \in \mathcal{P} \end{cases} \tag{2}$$

For illustration, consider $\Pi_1 = \{a \leftarrow \mathit{not}\ b; \mathit{not}\ a \leftarrow c; b \leftarrow c; c \leftarrow b\}$, whose corresponding propositional theory is $\tau[\Pi_1] = \{\neg b \rightarrow a; c \rightarrow \neg a; c \rightarrow b; b \rightarrow c\}$. The sets $X_1 = \{a\}$ and $X_2 = \{b, c\}$ are models of $\tau[\Pi_1]$, their reducts are $(\tau[\Pi_1])^{X_1} = \{\neg\bot \rightarrow a; \bot \rightarrow \bot\}$ and $(\tau[\Pi_1])^{X_2} = \{\bot \rightarrow \bot; c \rightarrow \neg\bot; c \rightarrow b; b \rightarrow c\}$. Clearly, X_1 is the unique minimal model of $(\tau[\Pi_1])^{X_1}$, thus, X_1 is an answer set of Π_1. The unique minimal model of $(\tau[\Pi_1])^{X_2}$ is $\emptyset$, that is, X_2 is not an answer set of Π_1.

Unlike the semantics, our tableau framework directly deals with logic programs. The global design, however, follows the two semantic requirements for answer sets: modelhood wrt a program and (non-circular) support wrt the reduct. To begin with, we define for a program Π, two sets $S, S' \subseteq atom(\Pi)$, viz. the set of atoms occurring in Π, and an assignment A:

$$sup_A(\Pi, S, S') = \{(\alpha \leftarrow \beta) \in \Pi \mid \boldsymbol{f}\beta \notin A,\ \overleftarrow{sup}_A(\alpha, S),\ \overrightarrow{sup}_A(\beta, S')\} \tag{3}$$

The purpose of $sup_A(\Pi, S, S')$ is to determine all rules in Π that can, wrt A, provide a support for the atoms in S external wrt S'. Of particular interest are the cases where $sup_A(\Pi, S, S')$ is empty or a singleton $\{\alpha \leftarrow \beta\}$. In the first case, the atoms in S cannot be supported and are prone to be false, while the second case tells us that $\alpha \leftarrow \beta$ is the unique support for S external wrt S' so that β must be true to (non-circularly) support S.

Looking at the definition of $sup_A(\Pi, S, S')$ in (3), we note that a rule $\alpha \leftarrow \beta$ such that $\boldsymbol{f}\beta \in A$ cannot provide any support wrt A. Otherwise, we check via $\overleftarrow{sup}_A(\alpha, S)$ that α can support S and via $\overrightarrow{sup}_A(\beta, S')$ that β does not (positively) rely on S'. For the simple case of unary programs, these concepts are defined as follows:

$$\overleftarrow{sup}_A(p, S) \quad \text{if } p \in S \tag{4}$$
$$\overrightarrow{sup}_A(p, S') \quad \text{if } p \in (\mathcal{P} \setminus S') \tag{5}$$
$$\overrightarrow{sup}_A(not\ v, S') \quad \text{for every expression } v \tag{6}$$

The universal validity of (6) is because only *positive* dependencies are taken into account. Also note that a rule $\alpha \leftarrow \beta$ such that $\alpha = not\ v$ cannot support any set S of atoms. (We further illustrate the above concepts after Theorem 1.)

The tableau rules constituting our basic calculus are given in Figure 1. Rules $I\!\uparrow$ and $I\!\downarrow$ provide basic *rule-based* inferences such as modus ponens and modus tollens. Tableau rules $N\!\uparrow$ and $N\!\downarrow$ amount to *negation* and *support* for atoms stemming from Clark's completion [11]. Note that the derivability of an atom p and thus the applicability of tableau rules $N\!\uparrow$ and $N\!\downarrow$, respectively, is determined by $sup_A(\Pi, \{p\}, \emptyset)$. In the general case, rule $N\!\downarrow$ makes use of two further constructs, $min_A(\alpha, S)$ and $max_A(\beta, S')$, that are used to determine entries that must necessarily be added to A in order to support some atom in S via rule $\alpha \leftarrow \beta$ without positively relying on S'. However, these concepts play no role in the setting of unary programs:

$$min_A(p, S) = \emptyset \quad \text{for } p \in \mathcal{P} \tag{7}$$
$$max_A(p, S') = \emptyset \quad \text{for } p \in \mathcal{P} \tag{8}$$
$$max_A(not\ v, S') = \emptyset \quad \text{for every expression } v \tag{9}$$

Tableau rules $U\!\uparrow$ and $U\!\downarrow$ take care of *unfounded sets* [12], either by identifying atoms that cannot be non-circularly supported ($U\!\uparrow$) or by preventing true atoms to become

$$\frac{\begin{array}{c}\alpha \leftarrow \beta \\ \boldsymbol{t}\beta\end{array}}{\boldsymbol{t}\alpha}$$
(a) Implication ($I\!\uparrow$)

$$\frac{\begin{array}{c}\alpha \leftarrow \beta \\ \boldsymbol{f}\alpha\end{array}}{\boldsymbol{f}\beta}$$
(b) Contraposition ($I\!\downarrow$)

$$\frac{\mathit{\Pi}, A}{\boldsymbol{F}p}\,(p \in atom(\mathit{\Pi}), sup_A(\mathit{\Pi}, \{p\}, \emptyset) = \emptyset)$$
(c) Negation ($N\!\uparrow$)

$$\frac{\mathit{\Pi}, A}{\boldsymbol{t}\beta, min_A(\alpha, \{p\}), max_A(\beta, \emptyset)}\,(p \in (A^{\boldsymbol{T}} \cap atom(\mathit{\Pi})), sup_A(\mathit{\Pi}, \{p\}, \emptyset) = \{\alpha \leftarrow \beta\})$$
(d) Support ($N\!\downarrow$)

$$\frac{\mathit{\Pi}, A}{\boldsymbol{F}p}\,(S \subseteq atom(\mathit{\Pi}), p \in S, sup_A(\mathit{\Pi}, S, S) = \emptyset)$$
(e) Unfounded Set ($U\!\uparrow$)

$$\frac{\mathit{\Pi}, A}{\boldsymbol{t}\beta, min_A(\alpha, S), max_A(\beta, S)}\,(S \subseteq atom(\mathit{\Pi}), (A^{\boldsymbol{T}} \cap S) \neq \emptyset, sup_A(\mathit{\Pi}, S, S) = \{\alpha \leftarrow \beta\})$$
(f) Well-founded Set ($U\!\downarrow$)

$$\frac{}{\boldsymbol{T}v \mid \boldsymbol{F}v}\,(v \in \mathit{\Gamma})$$
(g) Cut ($C[\mathit{\Gamma}]$)

Fig. 1. Generic tableau rules for rules *(a),(b)*; atoms *(c),(d)*; sets of atoms *(e),(f)*; and cutting *(g)*.

unfounded ($U\!\downarrow$). The applicability of $U\!\uparrow$ and $U\!\downarrow$ is determined by $sup_A(\mathit{\Pi}, S, S)$ for a set S of atoms; hence, these rules subsume $N\!\uparrow$ and $N\!\downarrow$ relying on $sup_A(\mathit{\Pi}, \{p\}, \emptyset)$. We nonetheless include $N\!\uparrow$ and $N\!\downarrow$ since their applicability is easy to determine, and so they have counterparts in virtually all ASP solvers. Finally, the cut rule $C[\mathit{\Gamma}]$ in *(g)* constitutes the only rule introducing multiple branches. It allows for case analysis on the expressions in $\mathit{\Gamma}$, if the deterministic tableau rules do not yield a complete branch.

Note that the definition of $sup_A(\mathit{\Pi}, S, S')$ in (3) is common to both support-driven (viz. $N\!\uparrow$ and $N\!\downarrow$) and unfoundedness-driven (viz. $U\!\uparrow$ and $U\!\downarrow$) inferences. As we demonstrate in the following sections, an additional language construct is then incorporated into the basic tableau calculus by supplying tableau rules for handling its truth value and by extending the definition of $\overleftarrow{sup}_A(\alpha, S)$ and $\overrightarrow{sup}_A(\beta, S')$ (along with $min_A(\alpha, S)$ and $max_A(\beta, S')$) to impose an appropriate notion of support.

For a unary program $\mathit{\Pi}$, we fix the domain of assignments A as well as the cut objects used by $C[\mathit{\Gamma}]$ to $dom(A) = \mathit{\Gamma} = atom(\mathit{\Pi})$. This allows us to characterize the answer sets of unary programs by tableaux.

Theorem 1. *Let $\mathit{\Pi}$ be a unary program and $\emptyset$ the empty assignment.*
Then, the following hold for the tableau calculus consisting of tableau rules (a-g):

1. *Program $\mathit{\Pi}$ has no answer set iff every complete tableau for $\mathit{\Pi}$ and $\emptyset$ is a refutation.*
2. *If $\mathit{\Pi}$ has an answer set X, then every complete tableau for $\mathit{\Pi}$ and $\emptyset$ has a unique non-contradictory branch $(\mathit{\Pi}, A)$ such that $X = A^{\boldsymbol{T}} \cap atom(\mathit{\Pi})$.*

3. *If a tableau for Π and $\emptyset$ has a non-contradictory complete branch (Π, A), then $A^{\boldsymbol{T}} \cap atom(\Pi)$ is an answer set of Π.*

For illustration, consider the program $\Pi_2 = \{a \leftarrow not\ b; b \leftarrow not\ a; c \leftarrow not\ a\}$. Cutting on c results in branches with $\boldsymbol{T}c$ and $\boldsymbol{F}c$, respectively. The first one can be extended by $\boldsymbol{t}not\ a = \boldsymbol{F}a$ via $N{\downarrow}$. Indeed, $sup_{\{\boldsymbol{T}c\}}(\Pi_2, \{c\}, \emptyset) = \{c \leftarrow not\ a\}$ tells us that $c \leftarrow not\ a$ is the only rule that allows for supporting c, which necessitates a to be false. To be more precise, we have $\boldsymbol{f}not\ a = \boldsymbol{T}a \notin \{\boldsymbol{T}c\}$, and both $\overleftarrow{sup}_{\{\boldsymbol{T}c\}}(c, \{c\})$ and $\overrightarrow{sup}_{\{\boldsymbol{T}c\}}(not\ a, \emptyset)$ are satisfied; the proviso of $N{\downarrow}$ is thus established with respect to rule $c \leftarrow not\ a$, so we deduce $\boldsymbol{t}not\ a = \boldsymbol{F}a$. The two remaining sets, $min_{\{\boldsymbol{T}c\}}(c, \{c\}) = max_{\{\boldsymbol{T}c\}}(not\ a, \emptyset) = \emptyset$, are superfluous in the simple language fragment of unary programs. Having deduced $\boldsymbol{F}a$, we can either apply $I{\uparrow}$ or $I{\downarrow}$ to conclude $\boldsymbol{T}b$ and so to obtain a complete branch. The second branch with $\boldsymbol{F}c$ can be extended by $\boldsymbol{f}not\ a = \boldsymbol{T}a$ via $I{\downarrow}$. From this, we deduce $\boldsymbol{F}b$, either by $N{\uparrow}$ or $N{\downarrow}$, obtaining a second complete branch. The two complete branches tell us that $\{b, c\}$ and $\{a\}$ are the answer sets of Π_2.

Further, consider the program $\Pi_3 = \{a \leftarrow b; b \leftarrow a; b \leftarrow c; c \leftarrow not\ d; d \leftarrow not\ c\}$ along with the following two complete branches:

Π_3		Π_3	
$\boldsymbol{T}d$	$(C[atom(\Pi)])$	$\boldsymbol{T}a$	$(C[atom(\Pi)])$
$\boldsymbol{F}c$	$(N{\uparrow}, N{\downarrow}, U{\uparrow}, U{\downarrow})$	$\boldsymbol{T}b$	$(I{\uparrow}, N{\downarrow}, U{\downarrow})$
$\boldsymbol{F}a$	$(U{\uparrow})$	$\boldsymbol{T}c$	$(U{\downarrow})$
$\boldsymbol{F}b$	$(I{\downarrow}, N{\uparrow}, U{\uparrow})$	$\boldsymbol{F}d$	$(N{\uparrow}, N{\downarrow}, U{\uparrow}, U{\downarrow})$

We have chosen these branches for illustrating the application of the unfounded set rule ($U{\uparrow}$) and the well-founded set rule ($U{\downarrow}$), respectively. (Along the branches, we indicate all possible inferences leading to the same result.) We first inspect the deduction of $\boldsymbol{F}a$ by $U{\uparrow}$ in the left branch. Taking set $\{a, b\}$ (and its element a) makes us check whether $sup_{\{\boldsymbol{T}d,\boldsymbol{F}c\}}(\Pi_3, \{a, b\}, \{a, b\})$ is empty. To this end, we have to inspect all rules that allow for deriving an atom in $\{a, b\}$ (as stipulated via $\overleftarrow{sup}_{\{\boldsymbol{T}d,\boldsymbol{F}c\}}(\alpha, \{a, b\})$). Given $\boldsymbol{f}c = \boldsymbol{F}c$, we only need to consider $a \leftarrow b$ and $b \leftarrow a$. Neither rule satisfies $\overrightarrow{sup}_{\{\boldsymbol{T}d,\boldsymbol{F}c\}}(\beta, \{a, b\})$, which leaves us with $sup_{\{\boldsymbol{T}d,\boldsymbol{F}c\}}(\Pi_3, \{a, b\}, \{a, b\}) = \emptyset$. The well-founded set inference of $\boldsymbol{T}c$ in the right branch requires a set of atoms, some of whose elements is true, such that only one rule can non-circularly support the set. Taking $\{a, b\}$, we can verify that $sup_{\{\boldsymbol{T}a,\boldsymbol{T}b\}}(\Pi_3, \{a, b\}, \{a, b\}) = \{b \leftarrow c\}$. The membership of $b \leftarrow c$ is justified by the fact that $\boldsymbol{f}c = \boldsymbol{F}c \notin \{\boldsymbol{T}a, \boldsymbol{T}b\}$, and that both $\overleftarrow{sup}_{\{\boldsymbol{T}a,\boldsymbol{T}b\}}(b, \{a, b\})$ and $\overrightarrow{sup}_{\{\boldsymbol{T}a,\boldsymbol{T}b\}}(c, \{a, b\})$ hold. Furthermore, neither $a \leftarrow b$ nor $b \leftarrow a$ is included in $sup_{\{\boldsymbol{T}a,\boldsymbol{T}b\}}(\Pi_3, \{a, b\}, \{a, b\})$ because $\overrightarrow{sup}_{\{\boldsymbol{T}a,\boldsymbol{T}b\}}(b, \{a, b\})$ and $\overrightarrow{sup}_{\{\boldsymbol{T}a,\boldsymbol{T}b\}}(a, \{a, b\})$ do not hold. Hence, only $\boldsymbol{t}c = \boldsymbol{T}c$ can justify $\boldsymbol{T}a$ and $\boldsymbol{T}b$.

4 Conjunctive Bodies

Having settled our basic framework, we now allow rule bodies to contain conjunctions. While rule bodies themselves are often regarded as conjunctions, we here take a slightly different perspective in viewing conjunctions as (simple) Boolean aggregates, which like atoms can be preceded by not. This gives us some first insights into the treatment

$$\frac{\boldsymbol{t}l_1, \dots, \boldsymbol{t}l_n}{\boldsymbol{T}\{l_1, \dots, l_n\}}$$
(h) True Conjunction ($TC\uparrow$)

$$\frac{\begin{array}{c}\boldsymbol{F}\{l_1, \dots, l_{i-1}, l_i, l_{i+1}, \dots, l_n\}\\ \boldsymbol{t}l_1, \dots, \boldsymbol{t}l_{i-1}, \boldsymbol{t}l_{i+1}, \dots, \boldsymbol{t}l_n\end{array}}{\boldsymbol{f}l_i}$$
(i) Falsify Conjunction ($TC\downarrow$)

$$\frac{\boldsymbol{f}l_i}{\boldsymbol{F}\{l_1, \dots, l_i, \dots, l_n\}}$$
(j) False Conjunction ($FC\uparrow$)

$$\frac{\boldsymbol{T}\{l_1, \dots, l_n\}}{\boldsymbol{t}l_1, \dots, \boldsymbol{t}l_n}$$
(k) Justify Conjunction ($FC\downarrow$)

Fig. 2. Tableau rules for conjunctions.

of more sophisticated aggregates like cardinality constraints to be dealt with in the next section.

A *conjunction* over an alphabet $\mathcal{P}$ is an expression of the form $\{l_1, \dots, l_n\}$ where l_i is an atomic literal for $1 \leq i \leq n$. We denote by $conj(\mathcal{P})$ the set of all conjunctions that can be constructed from atoms in $\mathcal{P}$. A rule $\alpha \leftarrow \beta$ such that α is an atomic literal and β is an atomic literal or a (possibly negated) conjunction of atomic literals is a *conjunctive rule*. A logic program is a *conjunctive program* if every rule in it is conjunctive. For defining the semantics of conjunctive programs, we add the following case to translation $\tau[\pi]$ in (2):

$$\tau[\pi] = \textstyle\bigwedge_{l \in \pi} \tau[l] \qquad \text{if } \pi \in conj(\mathcal{P})$$

For accommodating conjunctions within the generic tableau rules in Figure 1, we extend the previous concepts in (4-9) in a straightforward way:

$$\begin{array}{l}\overrightarrow{sup}_A(\{l_1, \dots, l_n\}, S') \quad \text{if } \overrightarrow{sup}_A(l, S') \text{ for every } l \in \{l_1, \dots, l_n\}\\ max_A(\{l_1, \dots, l_n\}, S') = \bigcup_{l \in \{l_1, \dots, l_n\}} max_A(l, S')\end{array}$$

Note that $max_A(\{l_1, \dots, l_n\}, S')$ is still empty since $max_A(l, S') = \emptyset$ for every atomic literal $l \in \{l_1, \dots, l_n\}$. It thus has no effect yet, but this changes in the next section.

For a conjunctive program Π, we fix the domain $dom(A)$ of assignments A and the cut objects Γ of $C[\Gamma]$ to $dom(A) = \Gamma = atom(\Pi) \cup conj(\Pi)$, where $conj(\Pi)$ is the set of conjunctions occurring in Π. The additional tableau rules for handling conjunctions are shown in Figure 2. Their purpose is to ensure that $\boldsymbol{T}\{l_1, \dots, l_n\} \in A$ iff $(A^{\boldsymbol{T}} \cap \mathcal{P}) \models (\tau[l_1] \wedge \dots \wedge \tau[l_n])$. By augmenting the basic calculus with the tableau rules in Figure 2, Theorem 1 extends to conjunctive programs.

Theorem 2. *Let Π be a conjunctive program and $\emptyset$ the empty assignment.*

Then, statements 1. to 3. given in Theorem 1 hold for the tableau calculus consisting of tableau rules (a-k).

5 Cardinality Constraints

We define a *cardinality constraint* over an alphabet $\mathcal{P}$ as an expression of the form $j\{l_1, \dots, l_n\}k$ where l_i is an atomic literal for $1 \leq i \leq n$ and j, k are integers such that

$0 \leq j \leq k \leq n$. We denote by $card(\mathcal{P})$ the set of all cardinality constraints that can be constructed from atoms in $\mathcal{P}$. For $v \in (\mathcal{P} \cup card(\mathcal{P}))$, we say that v and $not\ v$ are *cardinality literals*. A rule $\alpha \leftarrow \beta$ such that α is a cardinality literal and β is a cardinality literal or a (possibly negated) conjunction of cardinality literals is a *cardinality rule*. A logic program is a *cardinality program* if it consists of cardinality rules.

Several syntactic classes of programs with cardinality constraints (and other aggregates) can be found in the literature [6–10]. Furthermore, the semantics differ on aggregates in the heads of rules; which semantics is the right one depends on the intention and cannot be answered a priori. On the one hand, we here adopt the approach of [6–8] and interpret cardinality constraints in heads as "choice constructs," that is, derived atoms within a cardinality constraint are not minimized. On the other hand, the syntactic class of programs (or formulas, respectively) considered in [10] is presumably the most general one. It allows for arbitrary aggregates over arbitrary formulas, so that cardinality constraints and rules as defined here form a syntactic fragment. The other approaches [6–9], however, do not cover our notion of a cardinality rule. As before, we thus embed cardinality programs into the framework of [10] to fix their semantics.

The fact that true atoms are not minimized within cardinality constraints in heads of rules necessitates an extended translation of cardinality programs into propositional theories, since the semantics of the latter per default relies on the minimization of true atoms.[1] We now replace the definition of $\tau[\Pi]$ in (1) with the following one:

$$\begin{aligned}\tau[\Pi] = {} & \{\tau[\beta] \rightarrow \tau[\alpha] \mid (\alpha \leftarrow \beta) \in \Pi, \alpha \notin card(\mathcal{P})\} \cup \\ & \{\tau[\beta] \rightarrow \big(\tau[\alpha] \wedge \textstyle\bigwedge_{p \in atom(\alpha)}(p \vee \neg p)\big) \mid (\alpha \leftarrow \beta) \in \Pi, \alpha \in card(\mathcal{P})\}\end{aligned} \tag{10}$$

where $atom(j\{l_1, \ldots, l_n\}k) = \{l_1, \ldots, l_n\} \cap \mathcal{P}$ for $(j\{l_1, \ldots, l_n\}k) \in card(\mathcal{P})$. Note that conjuncts $\bigwedge_{p \in atom(\alpha)}(p \vee \neg p)$ are tautological and thus neutral as regards the (classical) models of $\tau[\Pi]$. Given an interpretation X, they rather justify the truth of all $p \in atom(\alpha) \cap X$ in $(\tau[\Pi])^X$ where $\neg p$ is replaced by $\bot$. We further add another case to translation $\tau[\pi]$ in (2), which amounts to the aggregate translation given in [10]:

$$\begin{aligned}\tau[\pi] = {} & \big(\textstyle\bigvee_{C \subseteq \{l_1, \ldots, l_n\}, |C| = j} \tau[C]\big) \wedge \neg\big(\textstyle\bigvee_{C \subseteq \{l_1, \ldots, l_n\}, |C| = k+1} \tau[C]\big) \\ & \text{if } (\pi = j\{l_1, \ldots, l_n\}k) \in card(\mathcal{P})\end{aligned}$$

Notably, the upper bound k is translated into a negative subformula, and only the subformula for the lower bound j still appears potentially in the reduct wrt a set of atoms. Also note that $\tau[j\{l_1, \ldots, l_n\}k]$ is of exponential size. Even if auxiliary atoms are used to obtain a polynomial translation, like the one described in [6], compilation approaches incur a significant blow-up in space. Our proof-theoretic characterizations, provided in the following, apply directly to cardinality constraints and thus avoid any such blow-up.

Figure 3 shows the tableau rules for cardinality constraints. For a cardinality program Π, we fix the domain $dom(A)$ of assignments A and the cut objects Γ of $\mathcal{C}[\Gamma]$ to $dom(A) = \Gamma = atom(\Pi) \cup conj(\Pi) \cup card(\Pi)$, where $card(\Pi)$ is the set of cardinality constraints occurring in Π. For a cardinality constraint $j\{l_1, \ldots, l_n\}k$, the tableau rules in Figure 3 ensure that $\boldsymbol{T}(j\{l_1, \ldots, l_n\}k) \in A$ iff $(A^{\boldsymbol{T}} \cap \mathcal{P}) \models \tau[j\{l_1, \ldots, l_n\}k]$.

[1] Interpreting aggregates in rule heads as "choice constructs" avoids an increase of complexity by one level in the polynomial hierarchy. If derived atoms were to be minimized, it would be straightforward to embed disjunctive programs (see next section) into cardinality programs.

$$\frac{\boldsymbol{t}l_1,\dots,\boldsymbol{t}l_j,\boldsymbol{f}l_{k+1},\dots,\boldsymbol{f}l_n}{\boldsymbol{T}j\{l_1,\dots,l_j,\dots,l_{k+1},\dots,l_n\}k}$$

(l) True Bounds ($TLU\!\uparrow$)

$$\frac{\begin{array}{c}\boldsymbol{F}j\{l_1,\dots,l_{j-1},l_j,\dots,l_k,l_{k+1},\dots,l_n\}k\\ \boldsymbol{t}l_1,\dots,\boldsymbol{t}l_{j-1},\boldsymbol{f}l_{k+1},\dots,\boldsymbol{f}l_n\end{array}}{\boldsymbol{f}l_j,\dots,\boldsymbol{f}l_k}$$

(m) Falsify Lower Bound ($TL\!\downarrow$)

$$\frac{\begin{array}{c}\boldsymbol{F}j\{l_1,\dots,l_j,l_{j+1},\dots,l_{k+1},l_{k+2},\dots,l_n\}k\\ \boldsymbol{t}l_1,\dots,\boldsymbol{t}l_j,\boldsymbol{f}l_{k+2},\dots,\boldsymbol{f}l_n\end{array}}{\boldsymbol{t}l_{j+1},\dots,\boldsymbol{t}l_{k+1}}$$

(n) Falsify Upper Bound ($TU\!\downarrow$)

$$\frac{\boldsymbol{f}l_j,\dots,\boldsymbol{f}l_n}{\boldsymbol{F}j\{l_1,\dots,l_j,\dots,l_n\}k}$$

(o) False Lower Bound ($FL\!\uparrow$)

$$\frac{\boldsymbol{t}l_1,\dots,\boldsymbol{t}l_{k+1}}{\boldsymbol{F}j\{l_1,\dots,l_{k+1},\dots,l_n\}k}$$

(p) False Upper Bound ($FU\!\uparrow$)

$$\frac{\begin{array}{c}\boldsymbol{T}j\{l_1,\dots,l_j,l_{j+1}\dots,l_n\}k\\ \boldsymbol{f}l_{j+1},\dots,\boldsymbol{f}l_n\end{array}}{\boldsymbol{t}l_1,\dots,\boldsymbol{t}l_j}$$

(q) Justify Lower Bound ($FL\!\downarrow$)

$$\frac{\begin{array}{c}\boldsymbol{T}j\{l_1,\dots,l_k,l_{k+1}\dots,l_n\}k\\ \boldsymbol{t}l_1,\dots,\boldsymbol{t}l_k\end{array}}{\boldsymbol{f}l_{k+1},\dots,\boldsymbol{f}l_n}$$

(r) Justify Upper Bound ($FU\!\downarrow$)

Fig. 3. Tableau rules for cardinality constraints.

For illustration, consider the cardinality constraint

$$\gamma = 2\{a,b,c,\mathit{not}\ d,\mathit{not}\ e\}3\ ,$$

having $n=5$ literals, lower bound $j=2$, and upper bound $k=3$. For an assignment A, tableau rule $TLU\!\uparrow$ allows for deducing $\boldsymbol{T}\gamma$ if at least $j=2$ literals l of γ are true (i.e., $\boldsymbol{t}l\in A$) and at least $n-k=2$ literals l of γ are false (i.e., $\boldsymbol{f}l\in A$). This holds, for instance, for assignment $A_1=\{\boldsymbol{T}a,\boldsymbol{F}b,\boldsymbol{T}d,\boldsymbol{F}e\}$; hence, $\boldsymbol{T}\gamma$ can be deduced via $TLU\!\uparrow$. Indeed, the lower and upper bound of γ are satisfied in every non-contradictory branch that extends A_1, no matter whether $\boldsymbol{T}c$ or $\boldsymbol{F}c$ is additionally included. Rules $TL\!\downarrow$ and $TU\!\downarrow$ are the contrapositives of $TLU\!\uparrow$, ensuring that either the lower or the upper bound of γ is violated if $\boldsymbol{F}\gamma$ belongs to an assignment. For instance, $\boldsymbol{F}c$ and $\boldsymbol{T}e$ can be deduced via $TL\!\downarrow$ wrt assignment $A_2=\{\boldsymbol{F}\gamma,\boldsymbol{T}a,\boldsymbol{F}b,\boldsymbol{T}d\}$. Observe that the upper bound $k=3$ cannot be violated in non-contradictory extensions of A_2 since $n-k=2$ literals of γ are already false wrt A_2, as it contains $\boldsymbol{F}b$ and $\boldsymbol{T}d$. Conversely, $TU\!\downarrow$ allows for deducing $\boldsymbol{T}c$ and $\boldsymbol{F}e$ wrt $\{\boldsymbol{F}\gamma,\boldsymbol{T}a,\boldsymbol{F}b,\boldsymbol{F}d\}$ in order to violate the upper bound of γ; the lower bound of γ cannot be violated because of $\boldsymbol{T}a$ and $\boldsymbol{F}d$. The remaining four tableau rules in Figure 3 either allow for deducing $\boldsymbol{F}\gamma$, if its lower bound ($FL\!\uparrow$) or its upper bound ($FU\!\uparrow$) is violated, or make sure that the lower bound ($FL\!\downarrow$) and the upper bound ($FU\!\downarrow$) are satisfied, if $\boldsymbol{T}\gamma$ belongs to an assignment. For γ as above, $\boldsymbol{F}\gamma$ can be deduced via $FL\!\uparrow$ wrt assignment $\{\boldsymbol{F}b,\boldsymbol{F}c,\boldsymbol{T}d,\boldsymbol{T}e\}$ or via $FU\!\uparrow$ wrt assignment $\{\boldsymbol{T}b,\boldsymbol{T}c,\boldsymbol{F}d,\boldsymbol{F}e\}$. Conversely, $FL\!\downarrow$ allows for deducing $\boldsymbol{T}a$ and $\boldsymbol{F}e$ wrt $\{\boldsymbol{T}\gamma,\boldsymbol{F}b,\boldsymbol{F}c,\boldsymbol{T}d\}$, and $FU\!\downarrow$ allows for deducing $\boldsymbol{F}a$ and $\boldsymbol{T}e$ wrt $\{\boldsymbol{T}\gamma,\boldsymbol{T}b,\boldsymbol{T}c,\boldsymbol{F}d\}$.

To integrate cardinality constraints into the generic setting of the tableau rules in Figure 1, we also have to extend the concepts in (4-9):

$$\begin{array}{rcl}
\overleftarrow{sup}_A(j\{l_1,\dots,l_n\}k,S) & \text{if} & \{l_1,\dots,l_n\}\cap S\neq\emptyset \text{ and} \\
 & & |\{l\in(\{l_1,\dots,l_n\}\setminus S)\mid \boldsymbol{t}l\in A\}|<k \\
\overrightarrow{sup}_A(j\{l_1,\dots,l_n\}k,S') & \text{if} & |\{l\in(\{l_1,\dots,l_n\}\setminus S')\mid \boldsymbol{f}l\notin A\}|\geq j \\
min_A(j\{l_1,\dots,l_n\}k,S) & = & \begin{cases}\{\boldsymbol{f}l\mid l\in(\{l_1,\dots,l_n\}\setminus S),\ \boldsymbol{t}l\notin A\} \\ \quad \text{if } |\{l\in(\{l_1,\dots,l_n\}\setminus S)\mid \boldsymbol{t}l\in A\}|=k-1 \\ \emptyset \ \text{ if } |\{l\in(\{l_1,\dots,l_n\}\setminus S)\mid \boldsymbol{t}l\in A\}|\neq k-1\end{cases} \\
max_A(j\{l_1,\dots,l_n\}k,S') & = & \begin{cases}\{\boldsymbol{t}l\mid l\in(\{l_1,\dots,l_n\}\setminus S'),\ \boldsymbol{f}l\notin A\} \\ \quad \text{if } |\{l\in(\{l_1,\dots,l_n\}\setminus S')\mid \boldsymbol{f}l\notin A\}|=j \\ \emptyset \ \text{ if } |\{l\in(\{l_1,\dots,l_n\}\setminus S')\mid \boldsymbol{f}l\notin A\}|\neq j\end{cases}
\end{array}$$

Recall that $\overleftarrow{sup}_A(\alpha,S)$ is used to determine whether a rule with head α can provide support for the atoms in S. If $\alpha=j\{l_1,\dots,l_n\}k$, then some atom of S must be contained in $\{l_1,\dots,l_n\}$. Furthermore, if $(\{l_1,\dots,l_n\}\setminus S)$ already contains k (or more) literals that are true wrt A, then the addition of $\boldsymbol{T}p$ to A for $p\in(\{l_1,\dots,l_n\}\cap S)$ would violate the upper bound k, so that the corresponding rule $\alpha\leftarrow\beta$ cannot support S. This also explains the false literals in $min_A(j\{l_1,\dots,l_n\}k,S)$ that can be deduced if $k-1$ literals of $(\{l_1,\dots,l_n\}\setminus S)$ are already true wrt A. If one more literal in $(\{l_1,\dots,l_n\}\setminus S)$ were made true, S would lose its last (external) support $\alpha\leftarrow\beta$, though containing some atom that is true wrt A (cf. $N\!\downarrow$ and $U\!\downarrow$ in Figure 1). In addition, $\overrightarrow{sup}_A(\beta,S')$ is used to verify whether a support via β is external wrt S'. If, for a rule $\alpha\leftarrow\beta$, either $\beta=j\{l_1,\dots,l_n\}k$ or β is a conjunction such that $j\{l_1,\dots,l_n\}k\in\beta$, then there must be enough non-false literals in $(\{l_1,\dots,l_n\}\setminus S')$ to achieve the lower bound j. Furthermore, if the number of such literals is exactly j, then all of them must be true for providing a support that is external wrt S'. This is expressed by $max_A(j\{l_1,\dots,l_n\}k,S')$.

For illustration, consider the following cardinality program Π_4:[2]

$$\Pi_4=\left\{\begin{array}{llll} r_1: & 0\{c,d,e\}3\leftarrow & & \\ r_2: & 1\{a,b\}2\leftarrow c,d & r_4: & 1\{b,d\}2\leftarrow 1\{a,c\}2 \\ r_3: & 0\{a,d\}1\leftarrow 1\{b,\mathit{not}\ e\}2 & r_5: & 1\{a,d\}2\leftarrow b\end{array}\right\}$$

Let assignment A contain $\boldsymbol{T}a$, $\boldsymbol{F}c$, and $\boldsymbol{F}\{c,d\}$, and note that tableau rule $U\!\downarrow$ (or $N\!\downarrow$) does not apply to set $\{a\}$ since $sup_A(\Pi_4,\{a\},\{a\})=\{r_3,r_5\}$. Let us however consider set $\{a,b\}$. Given that $\{c,d,e\}\cap\{a,b\}=\emptyset$ and $\boldsymbol{F}\{c,d\}\in A$, we have $r_1\notin sup_A(\Pi_4,\{a,b\},\{a,b\})$ and $r_2\notin sup_A(\Pi_4,\{a,b\},\{a,b\})$. Considering r_5, we have $b\in\{a,b\}$ and thus $\overrightarrow{sup}_A(b,\{a,b\})$ does not hold. For r_4, we have $\{a,c\}\setminus\{a,b\}=\{c\}$ and $\boldsymbol{f}c=\boldsymbol{F}c\in A$, so that there are no non-false literals in $\{a,c\}\setminus\{a,b\}$. That is, the lower bound 1 of $1\{a,c\}2$ cannot be achieved independently from $\{a,b\}$, and thus $\overrightarrow{sup}_A(1\{a,c\}2,\{a,b\})$ does not hold. We have now established that only r_3 is potentially contained in $sup_A(\Pi_4,\{a,b\},\{a,b\})$. Since $(\mathit{not}\ e)\in\{b,\mathit{not}\ e\}$ is a non-false literal not belonging to $\{a,b\}$, $\overrightarrow{sup}_A(1\{b,\mathit{not}\ e\}2,\{a,b\})$ holds. Furthermore, $\overleftarrow{sup}_A(0\{a,d\}1,\{a,b\})$ holds because $\boldsymbol{t}d=\boldsymbol{T}d$ does not belong

[2] In the sequel, we skip set notation for conjunctive bodies within rules, like $\{c,d\}$ in rule r_2.

$$\frac{\boldsymbol{t}l_i}{\boldsymbol{T}\{l_1;\ldots;l_i;\ldots;l_n\}}$$
(s) True Disjunction ($TD\!\uparrow$)

$$\frac{\boldsymbol{F}\{l_1;\ldots;l_n\}}{\boldsymbol{f}l_1,\ldots,\boldsymbol{f}l_n}$$
(t) Falsify Disjunction ($TD\!\downarrow$)

$$\frac{\boldsymbol{f}l_1,\ldots,\boldsymbol{f}l_n}{\boldsymbol{F}\{l_1;\ldots;l_n\}}$$
(u) False Disjunction ($FD\!\uparrow$)

$$\frac{\begin{array}{c}\boldsymbol{T}\{l_1;\ldots;l_{i-1};l_i;l_{i+1};\ldots;l_n\}\\ \boldsymbol{f}l_1,\ldots,\boldsymbol{f}l_{i-1},\boldsymbol{f}l_{i+1},\ldots,\boldsymbol{f}l_n\end{array}}{\boldsymbol{t}l_i}$$
(v) Justify Disjunction ($FD\!\downarrow$)

Fig. 4. Tableau rules for disjunctions.

to A. We thus have $sup_A(\Pi_4,\{a,b\},\{a,b\}) = \{r_3\}$. As literals deducible via $U\!\downarrow$, we obtain $\boldsymbol{T}(1\{b, not\ e\}2)$, $max_A(1\{b, not\ e\}2,\{a,b\}) = \{\boldsymbol{t}not\ e = \boldsymbol{F}e\}$, and $min_A(0\{a,d\}1,\{a,b\}) = \{\boldsymbol{f}d = \boldsymbol{F}d\}$. Finally, note that, if $\boldsymbol{T}d$ had been contained in A, we would have obtained $sup_A(\Pi_4,\{a,b\},\{a,b\}) = \emptyset$, so that deducing $\boldsymbol{F}a$ via $U\!\uparrow$ would have led to a contradiction. Indeed, $\{a,b\}$ is the only answer set of Π_4 compatible with $\boldsymbol{T}a$ and $\boldsymbol{F}c$.

The tableau calculus for cardinality programs is obtained by adding the rules in Figure 3 to those in Figure 1 and 2. Then, Theorem 1 extends to cardinality programs.

Theorem 3. *Let Π be a cardinality program and $\emptyset$ the empty assignment.*

Then, statements 1. to 3. given in Theorem 1 hold for the tableau calculus consisting of tableau rules (a-r).

6 Disjunctive Heads

A *disjunction* over an alphabet $\mathcal{P}$ is an expression of the form $\{l_1;\ldots;l_n\}$ where l_i is an atomic literal for $1 \le i \le n$. We denote by $disj(\mathcal{P})$ the set of all disjunctions that can be constructed from atoms in $\mathcal{P}$. For $v \in (\mathcal{P} \cup card(\mathcal{P}) \cup disj(\mathcal{P}))$, v and $not\ v$ are *disjunctive literals*. A rule $\alpha \leftarrow \beta$ such that α is a disjunctive literal and β is a cardinality literal or a (possibly negated) conjunction of cardinality literals is a *disjunctive rule*. A logic program is a *disjunctive program* if it consists of disjunctive rules.

In contrast to cardinality constraints serving as "choice constructs," the common semantics for disjunctions relies on the minimization of derived atoms. Hence, we adhere to the definition of $\tau[\Pi]$ in (10) and just add another case to $\tau[\pi]$ in (2):

$$\tau[\pi] = \textstyle\bigvee_{l\in\pi}\tau[l] \qquad \text{if } \pi \in disj(\mathcal{P})$$

We further extend the concepts in (4-9) to disjunctive heads:

$$\begin{aligned}\overleftarrow{sup}_A(\{l_1;\ldots;l_n\},S) &\text{ if } \{l_1,\ldots,l_n\}\cap S \neq \emptyset \text{ and}\\ &\qquad \{l \in (\{l_1,\ldots,l_n\}\setminus S) \mid \boldsymbol{t}l \in A\} = \emptyset\\ min_A(\{l_1;\ldots;l_n\},S) &= \{\boldsymbol{f}l \mid l \in (\{l_1,\ldots,l_n\}\setminus S)\}\end{aligned}$$

For a disjunctive program Π, we fix the domain $dom(A)$ of assignments A and the cut objects Γ of $C[\Gamma]$ to $dom(A) = \Gamma = atom(\Pi) \cup conj(\Pi) \cup card(\Pi) \cup disj(\Pi)$, where $\mathit{disj}(\Pi)$ is the set of disjunctions occurring in Π. The additional tableau rules for handling disjunctions are shown in Figure 4. Their purpose is to ensure that $\boldsymbol{T}\{l_1;\ldots;l_n\} \in A$ iff $(A^{\boldsymbol{T}} \cap \mathcal{P}) \models (\tau[l_1] \vee \cdots \vee \tau[l_n])$. The tableau calculus for disjunctive programs is obtained by adding the rules in Figure 4 to those in Figure 1, 2, and 3. In this way, Theorem 1 extends to disjunctive programs.

Theorem 4. *Let Π be a disjunctive program and $\emptyset$ the empty assignment.*
Then, statements 1. to 3. given in Theorem 1 hold for the tableau calculus consisting of tableau rules (a-v).

7 Proof Complexity

For comparing different tableau calculi, we use the concept of *proof complexity* [3, 13, 14]. Intuitively, proof complexity is concerned with lower bounds on the run-times of proof-finding algorithms independent from heuristic influences. We thus compare the sizes of *minimal* refutations for unsatisfiable logic programs, that is, programs without answer sets. The size of a tableau is determined in the standard way as the number of nodes it contains. A tableau calculus $\mathcal{T}$ is *not polynomially simulated* by another tableau calculus $\mathcal{T}'$ if there is an infinite (witnessing) family $\{\Pi^n\}$ of unsatisfiable programs such that minimal refutations of $\mathcal{T}'$ for Π are asymptotically exponential in the size of minimal refutations of $\mathcal{T}$ for Π. A tableau calculus $\mathcal{T}$ is *exponentially stronger* than a tableau calculus $\mathcal{T}'$ if $\mathcal{T}$ polynomially simulates $\mathcal{T}'$, but not vice versa.

We have shown in [4] that the cut rule has a major influence on proof complexity. For normal logic programs Π, both $C[\mathit{atom}(\Pi)]$ and $C[\mathit{conj}(\Pi)]$ yield sound and complete tableau calculi. However, the calculus obtained with $C[\mathit{atom}(\Pi) \cup \mathit{conj}(\Pi)]$ is exponentially stronger than both of them [4]. It is thus an interesting question whether cutting on other composite constructs, namely, cardinality constraints and disjunctions, leads to stronger calculi. In this context, let us stress that cutting on atoms is sufficient for obtaining complete calculi even in the presence of language extensions, provided that the truth values of all composite constructs can be deduced from atomic literals via (deterministic) tableau rules. For cardinality constraints and disjunctions, this is possible using the tableau rules in Figure 3 and 4. In general, the assumption that knowing the atoms' truth values is enough to evaluate all constructs in a program seems reasonable.

We first consider cardinality programs Π. Let $\mathcal{T}_c = \{\textit{(a-f)}, \textit{(h-r)}, C[\mathit{atom}(\Pi) \cup \mathit{conj}(\Pi) \cup \mathit{card}(\Pi)]\}$ and $\mathcal{T}_{\overline{c}} = \{\textit{(a-f)}, \textit{(h-r)}, C[\mathit{atom}(\Pi) \cup \mathit{conj}(\Pi)]\}$. Both calculi contain all deterministic tableau rules dealing with cardinality programs; the difference is that cutting on cardinality constraints is allowed with $\mathcal{T}_c$, but not with $\mathcal{T}_{\overline{c}}$. Since every refutation of $\mathcal{T}_{\overline{c}}$ is as well a refutation of $\mathcal{T}_c$, it is clear that $\mathcal{T}_c$ polynomially simulates $\mathcal{T}_{\overline{c}}$.

As the following result shows, the converse does not hold.

Theorem 5. *Tableau calculus $\mathcal{T}_c$ is exponentially stronger than $\mathcal{T}_{\overline{c}}$.*

This result is witnessed by the following unsatisfiable cardinality programs:

$$\Pi_c^n = \left\{ \begin{array}{ll} x \leftarrow 1\{a_1, b_1\}2, \ldots, 1\{a_n, b_n\}2, \mathit{not}\ x & \\ a_i \leftarrow \mathit{not}\ b_i \qquad\qquad\qquad\qquad b_i \leftarrow \mathit{not}\ a_i \end{array} \;\middle|\; i = 1..n \right\}$$

Roughly, a branch containing $\boldsymbol{F}(1\{a_i, b_i\}2)$ for $1 \leq i \leq n$ yields an immediate contradiction because $\boldsymbol{F}a_i$ and $\boldsymbol{F}b_i$ can be deduced via $TL\!\downarrow$, violating the last two rules in Π_c^n. The unrestricted cut rule of $\mathcal{T}_c$ permits cutting on $1\{a_i, b_i\}2$ for $1 \leq i \leq n$, and the resulting minimal refutation has linear size in n. In contrast to this, $\mathcal{T}_{\overline{c}}$ must cut on atoms a_i or b_i, spanning a complete binary tree whose size is exponential in n.

The practical consequence of Theorem 5 is that ASP solvers dealing with cardinality constraints can gain significant speed-ups by branching on them. Notably, the compilation of cardinality constraints into so-called "basic constraint rules" [6], as done by *lparse* [15], introduces auxiliary atoms abbreviating cardinality constraints. In this way, *smodels* can (implicitly) branch on cardinality constraints although its choices are restricted to atoms. In contrast to this compilation approach introducing abbreviations, we here however consider cardinality constraints as self-contained structural entities.

Regarding disjunctive programs, we mention that verifying the non-applicability of tableau rules $U\!\uparrow$ and $U\!\downarrow$, dealing with unfounded sets, is coNP-hard [16].[3] Thus, tableaux for disjunctive programs are generally not polynomially verifiable, unless P=NP. Different from cardinality constraints occurring in bodies of rules, the syntax of disjunctive programs usually restricts disjunctions to occur in heads of rules; this is done here as well. If this restriction were dropped, program Π_c^n could be rewritten using disjunctions $\{a_i; b_i\}$ rather than $1\{a_i, b_i\}2$ for $1 \leq i \leq n$. This would yield the same exponential separation between cut rules with and without disjunctions as encountered on cardinality constraints. With disjunctions $\{l_1; \dots; l_n\}$ restricted to heads of rules, the crux is that the information gained in the branch of $\boldsymbol{T}\{l_1; \dots; l_n\}$ is rather weak. We thus conjecture that, with disjunctions restricted to heads, the possibility to branch on them does not yield exponential improvements, though it is certainly convenient to apply the cut rule to disjunctions as well. The effect of allowing or disallowing cutting on disjunctions in heads of rules however remains a subject to future investigation.

Finally, when looking at the inferences of existing ASP solvers, an asymmetry can be observed in unfounded set handling [4]. While (non-SAT-based) ASP solvers make inferences that can be described by tableau rule $U\!\uparrow$, no solver implements $U\!\downarrow$. This brings our attention to the question whether omitted inferences deteriorate proof complexity. (Of course, the available inferences must still be strong enough to guarantee soundness.) In what follows, we denote by $R\!\uparrow$ and $R\!\downarrow$ the forward and backward variant, respectively, of any of the (deterministic) tableau rules in Figure 1 to 4. Given a tableau calculus $\mathcal{T}$, we say that $\mathcal{T}' \subseteq \mathcal{T}$ is an *approximation* of $\mathcal{T}$ if $(\mathcal{T} \setminus \mathcal{T}') \subseteq \{R\!\downarrow \mid R\!\uparrow \in \mathcal{T}'\}$. (We assume that $TLU\!\uparrow \in \mathcal{T}'$ if $\{TL\!\downarrow, TU\!\downarrow\} \cap (\mathcal{T} \setminus \mathcal{T}') \neq \emptyset$ given that $TLU\!\uparrow$ has *two* backward counterparts, viz. $TL\!\downarrow$ and $TU\!\downarrow$.) That is, if $\mathcal{T}$ contains both $R\!\uparrow$ and $R\!\downarrow$, then an approximation $\mathcal{T}'$ of $\mathcal{T}$ might drop $R\!\downarrow$. Of course, $R\!\downarrow$ can also be kept, so that $\mathcal{T}$ is an (the greatest) approximation of itself. It is clear that every approximation $\mathcal{T}'$ of $\mathcal{T}$ is polynomially simulated by $\mathcal{T}$.

Assuming an unrestricted cut rule, the next result shows that the converse also holds.

Theorem 6. *Let $\mathcal{T}$ be a tableau calculus and $\mathcal{T}'$ an approximation of $\mathcal{T}$.*

If $C[atom(\Pi) \cup conj(\Pi) \cup card(\Pi) \cup disj(\Pi)] \in \mathcal{T}'$, then $\mathcal{T}$ is polynomially simulated by $\mathcal{T}'$.

[3] For cardinality programs, verifying the absence of unfounded sets is tractable. Suitably adjusted, the operations described in [8] could be used to accomplish this task.

In fact, an inference conducted by $R\downarrow$ can alternatively be obtained by cutting on the consequent of $R\downarrow$. Then, one of the two branches becomes contradictory by applying $R\uparrow$. However, recall that proof complexity assumes an optimal heuristics determining the "right" objects to cut on, which is inaccessible in practice. Hence, it is reasonable to also deduce the consequents of $R\downarrow$ within an ASP solver whenever it is feasible.

8 Discussion

In contrast to propositional logic, where the proof-theoretic foundations of SAT solvers are well-understood [2, 3, 13], little work has so far been done on proof theory for ASP and its solving approaches. The imbalance becomes even more apparent in view of the comprehensive semantic characterizations that nowadays exist for ASP. For instance, the approach in [10] specifies answer sets for propositional theories, going beyond the syntax of logic programs; it also provides a general semantics for aggregates. Though the latter are a key issue in ASP, where knowledge representation is a major objective, the support of such composite constructs in ASP solvers is rather ad hoc. This can, for instance, be seen with *lparse* compiling away negative weights within weight constraints, sometimes leading to counterintuitive results [10], and with *dlv* [17] not supporting aggregates in the heads of rules. While we have in [4] restricted our attention to normal logic programs, whose role in ASP is comparable to CNF in SAT, in this work, we have primarily aimed at language extensions and their integration into a common proof-theoretic framework. On the examples of cardinality constraints and disjunctive heads, we have seen that tableaux are well-suited for augmenting the core language and basic inference mechanisms with additional constructs.

Several lessons can be learned from the illustrative integration of cardinality constraints and disjunctive heads. Independently of the construct under consideration, inference rules follow two major objectives: first, making sure that solutions correspond to models of programs, and second, verifying that all true atoms are non-circularly supported. Different notions of support are possible. For instance, atoms derived via composite constructs in heads of rules might be subject to minimization, as with disjunctions, or not, as with cardinality constraints allowing for "choices." Such issues need to be settled in order to specify the required inference patterns. This also concerns computational complexity; for instance, it increases by one level in the polynomial hierarchy with disjunctive heads or negative weights within weight constraints. Furthermore, the proof complexity of an approach critically depends on the cut rule determining the objects available for case analysis. Composite constructs often constitute structures that are valuable in this respect, as it has been shown for conjunctions and cardinality constraints. We conclude that complex language constructs deserve particular attention in the context of ASP solving. For solvers using learning, like *clasp* [18], it is important not only that inferences are performed but also that their reasons are properly identified. The declarative nature of tableau rules provides a basis to describe such reasons. Importantly, the extensibility of the framework also allows for combining the processes of designing novel language constructs and of fixing their proof-theoretic meaning.

Acknowledgments. We are grateful to Robert Mercer, Richard Tichy, and the anonymous referees for many helpful suggestions.

References

1. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press (2003)
2. Mitchell, D.: A SAT solver primer. Bulletin of the European Association for Theoretical Computer Science **85** (2005) 112–133
3. Beame, P., Kautz, H., Sabharwal, A.: Towards understanding and harnessing the potential of clause learning. Journal of Artificial Intelligence Research **22** (2004) 319–351
4. Gebser, M., Schaub, T.: Tableau calculi for answer set programming. In Etalle, S., Truszczyński, M., eds.: Proceedings of the International Conference on Logic Programming (ICLP'06). Springer-Verlag (2006) 11–25
5. D'Agostino, M., Gabbay, D., Hähnle, R., Posegga, J., eds.: Handbook of Tableau Methods. Kluwer Academic Publishers (1999)
6. Simons, P., Niemelä, I., Soininen, T.: Extending and implementing the stable model semantics. Artificial Intelligence **138**(1-2) (2002) 181–234
7. Ferraris, P., Lifschitz, V.: Weight constraints as nested expressions. Theory and Practice of Logic Programming **5**(1-2) (2005) 45–74
8. Liu, L., Truszczyński, M.: Properties of programs with monotone and convex constraints. In Veloso, M., Kambhampati, S., eds.: Proceedings of the National Conference on Artificial Intelligence (AAAI'05). AAAI/MIT Press (2005) 701–706
9. Faber, W.: Unfounded sets for disjunctive logic programs with arbitrary aggregates. [19] 40–52
10. Ferraris, P.: Answer sets for propositional theories. [19] 119–131
11. Clark, K.: Negation as failure. In Gallaire, H., Minker, J., eds.: Logic and Data Bases. Plenum Press (1978) 293–322
12. van Gelder, A., Ross, K., Schlipf, J.: The well-founded semantics for general logic programs. Journal of the ACM **38**(3) (1991) 620–650
13. Beame, P., Pitassi, T.: Propositional proof complexity: Past, present, and future. Bulletin of the European Association for Theoretical Computer Science **65** (1998) 66–89
14. Järvisalo, M., Junttila, T., Niemelä, I.: Unrestricted vs restricted cut in a tableau method for Boolean circuits. Annals of Mathematics and Artificial Intelligence **44**(4) (2005) 373–399
15. Syrjänen, T.: Lparse 1.0 user's manual. http://www.tcs.hut.fi/Software/smodels/lparse.ps.gz
16. Leone, N., Rullo, P., Scarcello, F.: Disjunctive stable models: Unfounded sets, fixpoint semantics, and computation. Information and Computation **135**(2) (1997) 69–112
17. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV system for knowledge representation and reasoning. ACM Transactions on Computational Logic **7**(3) (2006) 499–562
18. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven answer set solving. In Veloso, M., ed.: Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI'07). AAAI/MIT Press (2007) 386–392
19. Baral, C., Greco, G., Leone, N., Terracina, G., eds.: Proceedings of the International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'05). Springer-Verlag (2005)