INSTITUT FÜR INFORMATIONSSYSTEME
ARBEITSBEREICH WISSENSBASIERTE SYSTEME

𝔽𝔻ℕℂ: DECIDABLE NONMONOTONIC DISJUNCTIVE LOGIC PROGRAMS WITH FUNCTION SYMBOLS

Mantas Šimkus Thomas Eiter

INFSYS RESEARCH REPORT 1843-08-01
JANUARY 2008

INFSYS Research Report

INFSYS Research Report 1843-08-01, January 2008

𝔽𝔻ℕℂ: Decidable Nonmonotonic Disjunctive Logic Programs with Function Symbols

Mantas Šimkus[1] and Thomas Eiter[1]

Abstract. Current Answer Set Programming frameworks and systems are built on non-monotonic logic programming without function symbols. As well-known, permitting function symbols leads to high undecidability in general. However, function symbols are highly desirable for various applications that involve common-sense reasoning over infinite domains, e.g., transition-based planning systems. This raises the challenge to find meaningful but still decidable fragments of this setting. To this end, we present the class 𝔽𝔻ℕℂ of logic programs which allows for function symbols, disjunction, non-monotonic negation under the answer set semantics, and constraints, while still retaining the decidability of the standard reasoning tasks. Thanks to these features, 𝔽𝔻ℕℂ programs are a powerful formalism for rule-based modeling of applications with potentially infinite processes and objects, which allows also for common-sense reasoning. This is evidenced, for instance, by tasks in reasoning about actions and planning: brave and open queries capture the well-known problems of plan existence and secure (conformant) plan existence problem, respectively, in transition-based actions domains. As for reasoning from 𝔽𝔻ℕℂ programs, we show that consistency checking and brave as well as cautious reasoning tasks are ExpTime-complete in general, but have lower complexity under syntactic restrictions that give rise to a family of program classes. Furthermore, we also determine the complexity of open queries (i.e., with answer variables), for which deciding non-empty answers is shown to be ExpSpace-complete under cautious entailment. Furthermore, we present for all reasoning tasks algorithms that are worst-case optimal. The majority of these algorithms resort to a finite representation of the stable models of a 𝔽𝔻ℕℂ program that employs maximal founded sets of knots, which are labeled trees of depth 1 from which each stable model can be reconstructed. Due to this property, reasoning over 𝔽𝔻ℕℂ programs can in many cases be reduced to reasoning from knots. Once the knot-representation for a program is derived (which can be done off-line), several reasoning tasks are not more expensive than in the function-free case, and some are even feasible in polynomial time. This knowledge compilation technique paves the way to potentially more efficient online reasoning methods for 𝔽𝔻ℕℂ and other formalisms.

[1]Institute of Information Systems, Knowledge-Based Systems Group, TU Vienna, Favoritenstraße 9-11, A-1040 Vienna, Austria. Email: (eiter|simkus)@kr.tuwien.ac.at

Acknowledgements: This work was partially supported by the Austrian Science Funds (FWF) project P17212 and the EC project REWERSE (IST-2003-506779).

Some of the results in this paper have been presented, in preliminary form, at the 14th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR 2007), Yerevan, Armenia, October 15-19, 2007.

Contents

1 Introduction **1**

2 Preliminaries **4**

3 $\mathbb{FDNC}$ Programs **6**
3.1 Characterization of Stable Models 8
3.2 Finite Representation of Stable Models 12

4 Complexity Results **16**

5 Complexity of $\mathbb{FDNC}$ **18**
5.1 Deriving Maximal Founded Set of Knots 18
5.2 Deciding Consistency 19
5.3 Brave Entailment of Queries 22
5.4 Cautious Entailment of Open Queries 24

6 Complexity of Fragments **26**
6.1 Reasoning in $\mathbb{FN}$ and $\mathbb{FNC}$ 26
6.2 Reasoning in $\mathbb{FC}$ 27
6.3 Reasoning in $\mathbb{F}$ and $\mathbb{FD}$ 31

7 Applications and Extensions **34**
7.1 Reasoning about Actions and Planning 34
7.2 Higher-arity $\mathbb{FDNC}$ 38

8 Conclusion **41**
8.1 Related Work 41
8.2 Summary and Further Issues 44

A Proofs and Constructions **45**
A.1 Auxiliary Lemma 45
A.2 Normalization of $\mathcal{ALC}$ KBs 47
A.3 Brave Entailment in $\mathbb{FDNC}$ 48
A.4 Open Queries 49
A.5 Reasoning in $\mathbb{FN}$ 53
A.6 Higher-arity $\mathbb{FDNC}$ 55

1 Introduction

Answer Set Programming (ASP) is a declarative problem solving paradigm that emerged from Logic Programming and Non-Monotonic Reasoning [6, 37, 39, 44], and is particularly well-suited for modeling and solving problems that involve common-sense reasoning. It is based on non-monotonic logic programs under the Answer Set Semantics (also known as stable model semantics) [23], which assigns a given program one, no, or multiple answer sets; this facilitates to encode many problems into logic programs such that their solutions correspond to the answer sets of the programs and can be easily extracted from them. This paradigm has been successfully applied to a range of applications including data integration, configuration, reasoning about actions and change, etc. We refer the reader [53] for a more detailed discussion and an overview of applications, whose number has rapidly increased in the last years.

While Answer Set Semantics, which underlies ASP, was defined in the setting of a general first-order language, current ASP frameworks and implementations, like DLV [21], Smodels [45], clasp [22] and other efficient solvers (see [3]) are based in essence on function-free languages and resort to Datalog with negation and its extensions.

However, it is widely acknowledged that this leads to drawbacks related to expressiveness, and also to inconvenience in knowledge representation, cf. [8]. Since one is forced to work with finite domains, potentially infinite processes cannot be represented naturally in ASP. Additional tools to simulate unbounded domains must be used. A notable example is the DLV$^{\mathcal{K}}$ [16] front-end of the DLV system which implements the action language $\mathcal{K}$ [17]. Constants are used to instantiate a sufficiently large domain (estimated by the user) for solving the problem; this may incur high space requirements, and does not scale to large instances.

Function symbols, in turn, are a very convenient means for generating infinite domains and objects, and allow for a more natural representation of problems in such domains. However, they have been banned in ASP for a good reason: allowing them leads to undecidability even for (rather simple) Horn programs [2],[1] and negation under the answer set semantics, leads to high undecidability, cf. [41, 38, 40]. This raises the challenge to single out meaningful fragments of ASP with function symbols which allow to model infinite domains while still retaining the decidability of the standard reasoning tasks. Several works have addressed this issue, including [11, 10, 8, 7, 51]. Most recently, Bonatti and his co-workers recently introduced *finitary* and *finitely recursive logic programs* [8, 7]. They imposed syntactic conditions on the groundings of logic programs, which are infinite in the presence of function symbols. Therefore, the verification of the conditions is difficult (in fact, unsoluble), which limits the applicability of the results. Syrjänen [51] used a generalization of stratification which can be effectively checked, while Chomicki and Imieliński [11, 10] programs without negation in which restrictions applied to the rules individually. We refer to Section 8 for more details and discussion of related work.

In this paper, we pursue an approach to obtain decidable logic programs with function symbols by merely constraining, similar as in [11, 10], the rule syntax in a way that can be effectively checked. To this end, we take inspiration from results in automated deduction and other areas of knowledge representation, where many procedures, like tableaux algorithms with blocking, or hyper-resolution, have been developed for deciding satisfiability in various fragments of first-order logic. When function symbols (or existential quantification) may occur, these procedures are often sophisticated because they have to deal with possibly infinite models. However, because of the

[1]See [34] for an interesting historic account of this result (quoted in [12]).

peculiarities of Answer Set Semantics, transferring these results to logic programs with function symbols is not straightforward. Reasoning with logic programs needs to be more refined since only Herbrand models of a program are of interest and, moreover, only particular such models (fulfilling the condition of stability), which happen to be special minimal Herbrand models.

The main contributions of this paper are briefly summarized as follows.

• We introduce the class $\mathbb{FDNC}$ of logic programs, which allow for function symbols, disjunction, constraints, and non-monotonic negation under the answer set semantics [23]. In order to provide decidable reasoning, $\mathbb{FDNC}$ programs are syntactically restricted in order to ensure that they have the *forest-shaped model* property. To this end, in the first stage programs are considered in which the predicates are unary and binary, and function symbols are unary; this gives us the class of ordinary $\mathbb{FDNC}$ programs, described in Section 3. To accommodate predicates of higher arity, an extension of $\mathbb{FDNC}$ to higher-arity predicates is conceived in Section 7. The syntactic restrictions are similar to those in [11], and limit the use of functions symbols, but are more restrictive. This, however, facilitates the development of special techniques for handling $\mathbb{FDNC}$-programs, which are needed in order to cope with negation, disjunction, and constraints, which had not been considered in [11].

Furthermore, we consider the natural restrictions of $\mathbb{FDNC}$ program that arise if the constructs of negation ($\mathbb{N}$), disjunction ($\mathbb{D}$) and constraints ($\mathbb{C}$) are disallowed, giving rise to a whole family of logic programs ranging from $\mathbb{F}$ to $\mathbb{FDNC}$; the plainest language in this family, $\mathbb{F}$, is a subclass of Horn programs that is (apart from minor deviations) a fragment of $Datalog_{nS}$ in [11].

• We study standard reasoning tasks for $\mathbb{FDNC}$, including deciding the consistency of a given program, i.e., existence of an answer set, brave entailment of a given ground atom A from a given program P, as well as cautious entailment of a given ground atom A from a given program P. Furthermore, we also consider answering existential queries of the form $\exists\vec{x}.p(\vec{x})$, where $\vec{x}$ is a tuple of variables, and open queries $\lambda\vec{x}.p(\vec{x})$, (where the variables $\vec{x}$ must be bound to a ground term prior to entailment checking) under both brave and cautious entailment.

For these problems, we develop algorithms and characterize their computational complexity over the whole family programs from of $\mathbb{F}$ to $\mathbb{FDNC}$, in terms of completeness results for suitable complexity classes. As we show, for $\mathbb{FDNC}$ all reasoning tasks are ExpTime-complete, with the exception of deciding answer existence for open queries under cautious entailment, which is ExpSpace-complete. Disallowing either disjunction and constraints (which gives $\mathbb{FN}$) or non-monotonic negation (which gives $\mathbb{FDC}$) does not lead to lower complexity, while all problems drop to PSpace-completeness if both negation and disjunction are disallowed (which gives $\mathbb{FC}$, that are Horn logic programs with constraints). Depending on the reasoning task and the constructs available, the complexity ranges in the other cases from polynomial time over co-NP, Σ_2^P, PSpace and ExpTime up to ExpSpace. In particular, for $\mathbb{F}$ programs (which are Horn programs), entailment of ground atoms is polynomial; note that even in the absence of function symbols, this problem is NP-hard for Horn programs with binary predicates. A compact summary of the complexity results is given in Table 1 of Section 4, which also provides a detailed discussion.

• The ExpTime-hardness proofs for consistency checking of programs in the fragments $\mathbb{FN}$, $\mathbb{FDC}$, and $\mathbb{FDNC}$, are by a reduction from satisfiability testing in the ExpTime-complete Description Logic $\mathcal{ALC}$. As a side result, we obtain a polynomial time mapping of a well-known Description

Logic (cf. [5]) to logic programs under answer set semantics. The mapping takes advantage of a normal form of $\mathcal{ALC}$ knowledge bases which makes the mapping task quite easy, and is balanced in the sense that it maps to class of logic program whose complexity is not higher then the one of $\mathcal{ALC}$ (see Section 8 for a discussion of other mappings). These results are interesting in their own right and may be exploited in other contexts.

- $\mathbb{FDNC}$ programs can have infinitely many and infinitely large stable models, which therefore can not be explicitly represented for reasoning purposes. We provide a method to finitely represent all the stable models of a given $\mathbb{FDNC}$ program. This is achieved by a composition technique that allows to reconstruct stable models as forests, i.e., sets of trees, from *knots*, which are instances of generic labeled trees of depth 1. The finite representation technique allows us to define an elegant decision procedure for brave reasoning in $\mathbb{FDNC}$. It may also be exploited for offline *knowledge compilation* [9, 13] to speed up online reasoning, by precomputing and storing a knot representation of a logic program P. Given such a representation, multiple query answering from P can be done comparatively efficiently (some problems are solvable in polynomial time), and furthermore also model building (which is of concern in ASP): starting from the knots as building blocks, any stable model of P can be gradually constructed (leading to an infinite process in general), at no higher cost than in the function-free case. In general, a knot representation of a logic program is exponential in the size of the program; this is the usual tradeoff between time and space for such compilation, and is encountered in other compilation forms as well (e.g., compilation of a propositional formula into all its prime implicates [13]).

Thanks to their features, $\mathbb{FDNC}$ programs are a powerful formalism for rule-based modeling of applications with potentially infinite processes or objects, which also accommodates common-sense reasoning through non-monotonic negation. From a complexity perspective, $\mathbb{FDNC}$ and its subclasses provide *effective syntax* for expressing problems in PSPACE, EXPTIME and EXPSPACE using logic programs with function symbols.

This can, for instance, be fruitfully exploited for reasoning about actions and planning. The usability of answer set programming in this area is well-known and has been explored in many works, including [14, 36, 6, 17, 52, 48, 49, 42]; the excellent book of Baral [6] devotes a whole chapter to this subject and is recommended for a background. $\mathbb{FDNC}$ programs allow to encode action domain descriptions in transition-based action formalisms which support incomplete states and nondeterministic action effects, like $\mathcal{C}$ [25], $\mathcal{K}$ [17], or fragments of the situation calculus (see e.g. [35] for background) in a way such that arbitrarily long action sequences can be naturally handled.

As an appetizer for the use of $\mathbb{FDNC}$ programs in this area, we sketch here informally elements of a simple encoding of a plain propositional variant of the situation calculus into $\mathbb{FDNC}$ programs. To this end, we use unary predicates $F(x)$ for fluents F that describe the state of the domain in a certain situation, a unary predicate $s(x)$ to denote situations x, and the constant *init* for the initial situation. For the latter, a fact $s(\mathit{init}) \leftarrow$ is set up, and the initial state of the domain is then described by facts of the form $F(\mathit{init}) \leftarrow$.

Transitions happen through the execution of actions $a_1, \ldots, a_n$, which are represented by function symbols $f_{a_1}, \ldots, f_{a_n}$; intuitively, $f_{a_i}(x)$ is the situation resulting if action a_i is taken in situation x. Using a binary predicate tr, we can express that a transition happened by $tr(x, f_{a_i}(x))$; a rule $a_1(x) \vee \cdots \vee a_n(x) \leftarrow s(x)$ singles out some action in situation x for moving on. If the action a_i can be taken, which is assessed by some predicate $poss_{a_i}(x)$, then the transition is made, described by the rule $tr(x, f_{a_i}(x)) \leftarrow a_i(x), poss_{a_i}(x)$; the new situation after taking an action is described with $s(y) \leftarrow tr(x, y)$.

These rules and facts provide a generic backbone for describing an evolving action domain. Particular action effects during transitions can be stated by rules of $\mathbb{FDNC}$; e.g., the rule $F(f_a(x)) \leftarrow tr(x, f_a(x))$ states that after executing the action α, F holds in the follow up situation. Importantly, the availability of non-monotonic negation allows to conveniently state fluent inertia, i.e., the fluent value when taking an action remains the same *by default.* For fluent F, this can be expressed using the following two rules

$$\begin{aligned} F(y) &\leftarrow F(x), tr(x, y), not\ \bar{F}(y), \\ \bar{F}(y) &\leftarrow \bar{F}(x), tr(x, y), not\ F(y), \end{aligned}$$

where $\bar{F}(x)$ is a predicate for the complement of $F(x)$ that can be simulated by adding the constraint $\leftarrow F(x), \bar{F}(x)$. Possible states of the domain in a situation (in case of incomplete information) can be captured by rules $F(x) \vee \bar{F}(x) \leftarrow s(x)$. Overall, the stable models of the program will then correspond to trajectories of the action domain, i.e., sequences of actions together with the fluent values at each stage of action execution. If we replace the disjunctive rule $a_1(x) \vee \cdots \vee a_n(x) \leftarrow s(x)$ with the rules $a_1(x) \leftarrow s(x); \ldots ; a_n(x) \leftarrow s(x)$, then the stable models correspond to the unwindings of the initial state according to the possible transitions.

Using these elements, $\mathbb{FDNC}$ may be used to represent a number of actions domains from the literature, e.g., the Yale Shooting [26], Bomb in the Toilet, and others cf. [17], and to solve reasoning and planning problems on them. In Section 7 we more concretely elaborate on an encoding of action domains in a fragment of the language $\mathcal{K}$ into $\mathbb{FDNC}$, and show on an example how query answering can be used to elegantly solve, among others, conformant planning problems in $\mathcal{K}$. The latter problems are ExpSpace-complete in general, and show that $\mathbb{FDNC}$-programs offer the complexity which is tailored to these problems.

The remainder of this paper is organized as follows. Section 2 briefly introduces the basic concepts and notation of disjunctive logic programs used in this paper. Section 3 then introduces $\mathbb{FDNC}$ programs, and establishes basic semantic properties of them. It also introduces the finite representation of stable models in terms of knots. Section 4 gives an overview and a discussion of the complexity results in this paper, which are established in the subsequent Sections 5 and 6. In the course of this, also reasoning techniques and algorithms are developed. Section 7 discusses a possible application of $\mathbb{FDNC}$-programs for planning, and considers and extension of ordinary to higher-arity $\mathbb{FDNC}$ programs. The final Section 8 discusses related work and concludes with issues for future work.

2 Preliminaries

We assume fixed countably infinite sets of *constant symbols*, *function symbols*, *predicate symbols*, and *variables.* Moreover, each function and relation symbol has an associated positive integer, its *arity.* A *term* is either a constant symbol, a variable or an expression of the form $f(\vec{t})$ where f is a function symbol, $\vec{t}$ is an n-tuple of terms and n is the arity of f. An *atom* is an expression of the form $R(\vec{t})$ where R is a predicate symbol, $\vec{t}$ is an n-tuple of terms and n is the arity of R. An atom is ground if it contains no variables. An atom is also called a *positive literal.* An expression of the form $not\ A$, where A is an atom, is a *negative literal.* A *literal* is a either a positive or a negative literal.

A *disjunctive logic program* (briefly, a *program*) is a set of *(disjunctive) rules* of the form

$$A_1 \vee \ldots \vee A_n \leftarrow L_1, \ldots, L_m, \tag{1}$$

where $n + m > 0$, $A_1, \ldots, A_n$ are atoms and $L_1, \ldots, L_m$ are literals. The atoms $A_1, \ldots, A_n$ are the *head atoms of the rule*, while $L_1, \ldots, L_m$ are the *body literals of the rule.* For a rule r, let $\mathsf{head}(r)$, $\mathsf{body}^+(r)$ and $\mathsf{body}^-(r)$ respectively denote the sets of head atoms, positive body literals, and negative body literals of r, respectively. A *fact* is a rule (1) with empty body ($m = 0$), also written A_0., while a *constraint* is a rule with no atoms in the head ($n = 0$).

If $\mathsf{body}^-(r) = \emptyset$, then the rule r is *positive.* For a positive rule r, let $\mathsf{body}(r) := \mathsf{body}^+(r)$. A program is *positive*, if it contains only positive rules.

The semantics of a program P is given in terms of Herbrand interpretations. Let $\mathcal{HU}^P$ be the *Herbrand universe* of P, i.e., the set of all terms that can be built from constants and function symbols occurring in P. Similarly, $\mathcal{HB}^P$ is the *Herbrand base* of P, i.e., the set of all atoms that can be built from predicate symbols of P and terms in $\mathcal{HU}^P$. A *Herbrand interpretation* for P is an arbitrary subset of $\mathcal{HB}^P$.

A term, atom, rule etc. is *ground*, if it contains no variables. A rule r' is a *ground instance of a rule* $r \in P$, if r' is a ground rule obtained from r by replacing each variable in r by a term in $\mathcal{HU}^P$; by $\mathsf{Ground}(P)$ we denote the set of all ground instances of the rules in P.

A (Herbrand) interpretation I *satisfies* a positive ground rule r, if $\mathsf{body}(r) \subseteq I$ implies $I \cap \mathsf{head}(r) \neq \emptyset$. An interpretation I is a *model* of a positive ground program P, if I satisfies each rule $r \in P$; moreover, I is a *minimal model* of P, if no $J \subset I$ is a model of P. The set of minimal models of P is denoted by $MM(P)$.

Given an interpretation I for a program P, the *Gelfond-Lifschitz reduct* [23] of P, denoted P^I, is obtained from $\mathsf{Ground}(P)$ by

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

(ii) removing all negative literals from the remaining rules.

Then I is a *stable model* (or *answer set*) of P, if $I \in MM(P^I)$. The set of all stable models of a program P is denoted by $SM(P)$. A program P is *consistent*, if $SM(P) \neq \emptyset$.

A *ground (atomic) query* is a ground atom A, and an *existential (atomic) query* is an expression $\exists\vec{x}.Q(\vec{x})$, where $\vec{x}$ is a n-tuple of variables and Q is an n-ary predicate symbol. An *open query* is a similar expression $\lambda\vec{x}.Q(\vec{x})$.

As usual, a program P *bravely entails* a ground query A (resp., an existential query $\exists\vec{x}.Q(\vec{x})$, denoted $P \models_b A$ (resp., $P \models_b \exists\vec{x}.Q(\vec{x})$), if A (resp. $\exists\vec{x}.Q(\vec{x})$) is true in some stable model I of P, i.e., I contains A (resp., some atom $Q(\vec{t})$). Furthermore, P bravely entails an open query $\lambda\vec{x}.Q(\vec{x})$, denoted $P \models_b \lambda\vec{x}.Q(\vec{x})$, if P bravely entails some ground query $Q(\vec{t})$; any such $\vec{t}$ is called an *answer* for the query.

The notion of cautious entailment, $\models_c$, is dually defined, where "some stable model" replaces "every stable." Note that $P \models_b \lambda\vec{x}.Q(\vec{x})$ iff $P \models_b \exists\vec{x}.Q(\vec{x})$, while $P \models_c \lambda\vec{x}.Q(\vec{x})$ implies $P \models_c \exists\vec{x}.Q(\vec{x})$ but not vice versa; this is because $\lambda\vec{x}$ requires that $\vec{t}$ is the *same* in all stable models, while $\exists\vec{x}$ permits varying terms in different stable models. Cautious entailment of open queries is a useful tool e.g. in planning, to determine *conformant (alias secure) plans*, i.e., sequences of actions whose execution lead to the goal, regardless of possibly incomplete knowledge about the initial state and/or nondeterministic action effects (see Section 7).

Example 1. Consider the program P consisting of the following rules:

$$\begin{array}{rl} D(a) \leftarrow & \\ B(f(x)) \leftarrow D(x), \mathit{not}\ A(x) & \qquad C(x) \leftarrow A(x) \\ A(x) \leftarrow D(x), \mathit{not}\ B(f(x)) & \qquad C(x) \leftarrow B(f(x)) \end{array}$$

P has two stable models $I_1 = \{D(a), B(f(a)), C(a)\}$ and $I_2 = \{D(a), A(a), C(a)\}$. This is because I_1 is a minimal model of
$P^{I_1} = \{D(a) \leftarrow; B(f^{i+1}(a)) \leftarrow D(f^i(a)); A(f^{i+1}(a)) \leftarrow D(f^{i+1}(a)); C(f^i(a)) \leftarrow A(f^i(a)) \mid i \geq 0\}$,
and I_2 is a minimal model of
$P^{I_2} = \{D(a) \leftarrow; B(f^{i+2}(a));\ A(f^i(a)) \leftarrow D(f^i(a)); C(f^i(a)) \leftarrow A(f^i(a)); C(f^i(x)) \leftarrow B(f^i(x))\}$.

No other interpretation is a stable model of P. Note that $P \models_b \exists x.C(x)$ and $P \models_c \exists \vec{x}.\ C(x)$, while $P \not\models_c \lambda\vec{x}.\ C(x)$, i.e., has no answer. On the other hand, $P \models_c \lambda.D(x)$ has the answer $x = a$.

3 $\mathbb{FDNC}$ Programs

We now introduce the class $\mathbb{FDNC}$ of logic programs with function symbols. The syntactic restrictions that are imposed ensure the decidability of the formalism, but allow infinitely many and possibly infinite stable models. We then analyze the model-theoretic properties of $\mathbb{FDNC}$ programs and introduce a method to finitely represent the (possibly infinite) collection of stable models of a program.

For convenience, we use $P^{\pm}(\vec{t})$ to generically denote one of the literals $P(\vec{t})$ and *not* $P(\vec{t})$.

Definition 3.1 ($\mathbb{FDNC}$ program)**.** An $\mathbb{FDNC}$ program is a finite disjunctive logic program whose rules are of the following forms:

$$\begin{array}{lrcl}
\text{(R1)} & A_1(x) \vee \ldots \vee A_n(x) & \leftarrow & B_0^{\pm}(x), \ldots, B_l^{\pm}(x) \\
\text{(R2)} & R_1(x,y) \vee \ldots \vee R_n(x,y) & \leftarrow & P_0^{\pm}(x,y), \ldots, P_l^{\pm}(x,y) \\
\text{(R3)} & R_1(x,f_1(x)) \vee \ldots \vee R_n(x,f_n(x)) & \leftarrow & P_0^{\pm}(x,g_0(x)), \ldots, P_l^{\pm}(x,g_l(x)) \\
\text{(R4)} & A_1(y) \vee \ldots \vee A_n(y) & \leftarrow & B_0^{\pm}(Z_0), \ldots, B_l^{\pm}(Z_l), R_0^{\pm}(x,y), \ldots, R_k^{\pm}(x,y) \\
\text{(R5)} & A_1(f(x)) \vee \ldots \vee A_n(f(x)) & \leftarrow & B_0^{\pm}(W_0), \ldots, B_l^{\pm}(W_l), R_0^{\pm}(x,f(x)), \ldots, R_k^{\pm}(x,f(x)) \\
\text{(R6)} & R_1(x,f_1(x)) \vee \ldots \vee R_n(x,f_k(x)) & \leftarrow & B_0^{\pm}(x), \ldots, B_l^{\pm}(x) \\
\text{(R7)} & C_1(\vec{c_1}) \vee \ldots \vee C_n(\vec{c_n}) & \leftarrow & D_1^{\pm}(\vec{b_1}), \ldots, D_l^{\pm}(\vec{b_l}),
\end{array}$$

where $n, l, k \geq 0$, each $Z_i \in \{x, y\}$, $W_i \in \{x, f(x)\}$, and each $\vec{c_i}$, $\vec{b_i}$ is a tuple of constants of arity ≤ 2. Each rule r is *safe*, i.e., each of its variables occurs in $\mathsf{body}^+(r)$. For rules of type (R5) we require at least on binary literal to be positive. Moreover, at least one rule in the program is of type (R7) and is a fact.

The fragments obtained from $\mathbb{FDNC}$ by disallowing disjunction, constraints or negative literals are denoted by omitting respectively $\mathbb{D}$, $\mathbb{C}$, and $\mathbb{N}$ in the name. The collection of all these fragments is called the $\mathbb{F}$ *family*.

The structure of the rules in $\mathbb{FDNC}$ syntax, the availability of non-monotonic negation and function symbols allows us to represent possibly infinite processes in a rather natural way. We provide here an example from the biology domain.

(01) $Change(x, grow(x)) \leftarrow Young(x),\ Warm(x)$
(02) $Change(x, cell_1(x)) \leftarrow Mature(x),\ Warm(x)$
(03) $Change(x, cell_2(x)) \leftarrow Mature(x),\ Warm(x)$
(04) $Change(x, die(x)) \leftarrow Cold(x)$
(05) $Young(cell_1(x)) \leftarrow Change(x, cell_1(x))$
(06) $Young(cell_2(x)) \leftarrow Change(x, cell_2(x))$
(07) $Mature(grow(x)) \leftarrow Young(x),\ Change(x, grow(x))$
(08) $Warm(grow(x)) \vee Cold(grow(x)) \leftarrow Change(x, grow(x))$
(09) $Warm(y) \leftarrow Warm(x), Change(x, y), not\ \ Cold(y)$
(10) $Cold(y) \leftarrow Cold(x), Change(x, y), not\ \ Warm(y)$
(11) $\leftarrow Cold(x),\ Warm(x)$
(12) $Young(b) \leftarrow$
(13) $Warm(b) \leftarrow$

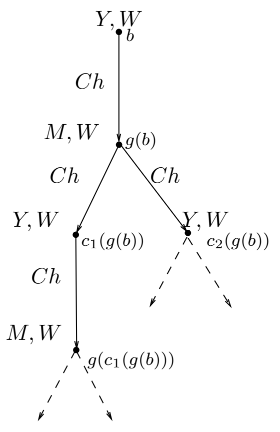


Figure 1: Example: Evolution of a Cell

Example 2. As a running example, we use the $\mathbb{FDNC}$ program P^{ex} in Figure 1, which represents the evolution of a cell; its growth and splitting into two cells. (1)-(4) describe changes of a cell. If it is warm, a young cell will grow and a mature cell will split into two cells; any cell dies if it is cold. The rules (5)-(7) determine whether a cell is young or mature. The rules (8)-(11) state the knowledge about the temperature. During the growth (which takes longer time), it might alter, while in the other changes (which take short time), it stays the same; the latter is expressed by inertia rules (9) and (10). Finally, (12) and (13) are the initialization facts. (For brevity, we also shorten predicate symbols to $W(arm)$, $C(old)$, $Y(oung)$, $M(ature)$, and $Ch(ange)$ and function symbols to $c(ell)_1$, $c(ell)_2$, $g(row)$, $d(ie)$.)

It is easy to see that P is consistent. In fact, it has infinitely many stable models, corresponding to the possible evolutions of the initial situation. It might have finite and infinite stable models, as cell splitting might go on forever. The part of the stable model that is depicted in Figure 1 represents a development where the temperature does not change during the growth of b and its child. Another stable model is $\{Young(b),\ Warm(b),\ Change(b, grow(b)),\ Cold(grow(b)),\ Mature(grow(b)),$ $Change(grow(b),\ die(grow(b)))\}$ which corresponds to the situation that the temperature changes and the bacterium dies.

The brave query $\exists x. Cold(x)$ evaluates to true; this is not the case for the brave query $Change(b, die(b))$. The query whether there is some evolution in which bacteria never die is expressed by adding the constraint $\leftarrow Change(x, die(x))$ and asking whether the resulting program is consistent (which is indeed the case).

Example 2 shows that in presence of function symbols, an $\mathbb{FDNC}$ program may have infinite stable models. We note that $\mathbb{FDNC}$ programs do not have the finite-model property, i.e., a program might have only infinite stable models. This is witnessed by the simple $\mathbb{F}$ program $P = \{A(c) \leftarrow;\ R(x, f(x)) \leftarrow A(x);\ A(y) \leftarrow R(x, y)\}$, whose single stable model contains infinitely many atoms.

Due to the lack of finite-model property, the search for stable models of an $\mathbb{FDNC}$ program cannot be confined to a finite search-space, i.e., consistency cannot be decided by considering a finite subset of the grounding of the program. We present in the sequel a method to finitely represent the possibly infinite stable models. To this end, we first provide a semantic characterization of the stable models of an $\mathbb{FDNC}$ program.

3.1 Characterization of Stable Models

Like many decidable logics, including Description Logics, $\mathbb{FDNC}$ programs enjoy a *forest-shaped model property.* A stable model of an $\mathbb{FDNC}$ program can be viewed as a graph and a set of trees rooted at each of nodes in the graph.

Proposition 1. *An interpretation I is* forest-shaped*, if the following hold:*

(a) All the atoms in I are either unary or binary. Additionally, each binary atom in I of of the form $R(c, d)$, $R(t, f(t))$ or $R(f(t), t)$, where c, d are constants, t is a term.

(b) If $A \in I$ is an atom with a term of the form $f(t)$ occurring as an argument, then for some binary predicate symbol R, $R(t, f(t)) \in I$.

If H is an arbitrary interpretation for an $\mathbb{FDNC}$ program P and $J \in MM(P^H)$, then J is forest-shaped. Therefore, every $J \in SM(P)$ is forest-shaped.

Proof. The property follows directly from the structure of the rules and the minimality requirements. Suppose H is an arbitrary interpretation for P. Assume some $J \in MM(P^H)$ such that it contains an atom violating *(a)* or *(b)*. We can simply collect all the atoms violating *(a)* or *(b)* and remove them from J. It is easy to see that such removal does not violate any rule in P^H, and, hence, we have that J is not minimal. Contradiction. The second claim follows from the definition of stable models. $\square$

The methods that we present in this paper are aimed at providing the decidability results together with the worst-case optimal algorithms for $\mathbb{FDNC}$. We note, however, that the decidability of the reasoning tasks discussed in this paper can be inferred from the results in [19]. The technique in [19] shows how the stable model semantics for the disjunctive logic programs with functions symbols can be expressed by formulae in second-order logic, where the minimality of models is enforced by second-order quantifiers. Due to the forest-shaped model property, one can express the semantics of $\mathbb{FDNC}$ programs in monadic second-order logic over trees *SkS*, which is known to be decidable (see [43] for a related encoding). Unfortunately, optimal algorithms or exact complexity characterizations are not apparent from such encodings, which are usually processed using automata-based algorithms.

The semantic characterization and the reasoning methods later on follow an intuition that stable models for an $\mathbb{FDNC}$ program P can be constructed by the iterative computation of stable models of *local programs.* During the construction, local programs are obtained "on the fly" by taking certain finite subsets of $\mathsf{Ground}(P)$ and adding facts (*states*) obtained in the previous iteration.

In the rest of Section 3, we assume that P is an arbitrary $\mathbb{FDNC}$ program. For convenience, given a term t and a set of atoms I, we write $t \hat{\in} I$, if there exists an atom in I having t as an argument.

Definition 3.2 (State)**.** Let t be a term. A *state of* t is an arbitrary set U^t containing only unary atoms ground with t (i.e., with t as the argument); the superscript t will be dropped if t is not of particular interest. For a set of atoms I and a term $t \hat{\in} I$, we denote by $\mathsf{st}(I, t)$ the state of t in I, i.e., the set $\{A(t) \mid A(t) \in I\}$.

For a one-variable rule r in $\mathbb{FDNC}$ syntax, let $r_{\downarrow t}$ denote the rule obtained by substituting every occurrence of the variable in r with a term t. Without loss of generality, we assume that in rules

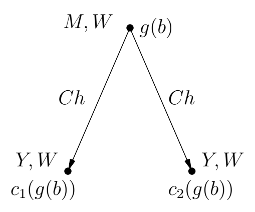
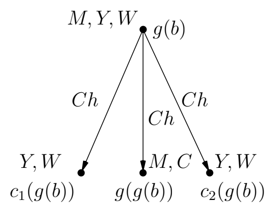
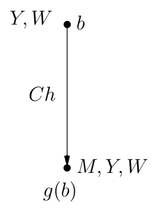
K_1	K_2	K_3
$M(g(b))$, $W(g(b))$, $Ch(g(b), c_1(g(b)))$, $Y(c_1(g(b)))$, $W(c_1(g(b)))$, $Ch(g(b), c_2(g(b)))$, $Y(c_2(g(b)))$, $W(c_2(g(b)))$	$M(g(b))$, $Y(g(b))$, $W(g(b))$ $Ch(g(b), c_1(g(b)))$, $Ch(g(b), c_2(g(b)))$, $Ch(g(b), g(g(b)))$, $Y(c_1(g(b)))$, $W(c_1(g(b)))$, $Y(c_2(g(b)))$, $W(c_2(g(b)))$, $M(g(g(b)))$, $C(g(g(b)))$,	$Y(b)$, $W(b)$, $Ch(b, g(b))$, $M(g(b))$, $Y(g(b))$, $W(g(b))$
 M, W $g(b)$ Ch Ch Y, W Y, W $c_1(g(b))$ $c_2(g(b))$	 M, Y, W $g(b)$ Ch Ch Ch Y, W M, C Y, W $c_1(g(b))$ $g(g(b))$ $c_2(g(b))$	

Figure 2: Example knots

r of type (R2) or (R4), the tuple of variables in binary atoms is always $\langle x, y\rangle$, and denote by $r_{\downarrow s,t}$ the rule obtained by substituting every occurrence of x and y with a term s and t, respectively.

Definition 3.3 (Local Program)**.** Let U^t be a state. The *local program* $P(U^t)$ is the smallest program containing the following rules:

- $A(t) \leftarrow$, for each $A(t) \in U^t$,
- $r_{\downarrow t}$, for each $r \in P$ of type (R3), (R5), or (R6),
- $r_{\downarrow t, f(t)}$, for each $r \in P$ of type (R2) or (R4) and function symbol f of P, and
- $r_{\downarrow f(t)}$, for each $r \in P$ of type (R1) and function symbol f of P.

Suppose I is a forest-shaped interpretation for P, $t \hat{\in} I$, and U is the state of t in I, i.e., $U = \mathsf{st}(I, t)$. Intuitively, the stable models of $P(U)$ define the set of possible immediate successor structures for t in I. In other words, if I is a stable model of P, then I must contain a stable model of $P(U)$. Stable models of local programs have a simple structural property, captured by the notion of *knots*.

Definition 3.4 (Knots)**.** A *knot with a root term* t is a set of atoms K such that

(i) each atom in K has form $A(t)$, $R(t, f(t))$, or $A(f(t))$ where A, R, and f are arbitrary, and

(ii) for each term $f(t) \hat{\in} K$, there exists $R(t, f(t)) \in K$ (connectedness).

We say K is *over (the signature of)* a program P, if each predicate and function symbol occurring in K also occurs in P (t need not be from $\mathcal{HU}^P$). Let $\mathsf{succ}(K)$ denote the set of all terms $f(t) \hat{\in} K$.

A knot with a root term t can be viewed as a labeled tree of depth at most 1, where $\mathsf{succ}(K)$ are the leaves. The nodes are labeled with unary predicate symbols, while the edges are labeled with binary predicate symbols. Note that $\emptyset$ is a knot whose root term can be arbitrary. Figure 2 shows an example of knots over the signature of the program P^{ex} in Example 2.

It is easy to see that due to the structure of local programs, their stable models satisfy the conditions in the definition of knots, and therefore are knots. On the other hand, knots are also the structures that appear in the trees of the forest-shaped interpretations. To “extract” a knot occurring in a forest-shaped interpretation, the following will be helpful.

For a term t, let $\mathcal{HB}_t$ denote the set of all atoms that can be built from unary and binary predicate symbols using t and terms of the form $f(t)$. For any forest-shaped interpretation I for P and $t \hat{\in} I$, the set $K := I \cap \mathcal{HB}_t$ is a knot.

The following notion of *stable knot* is central. They are self-contained building blocks for stable models of $\mathbb{FDNC}$ programs.

Definition 3.5 (Stable Knot)**.** Let K be a knot with a root term t and $U^t = \mathsf{st}(K, t)$. Then K is *stable* w.r.t. the program P iff $K \in SM(P(U^t))$.

Intuitively, stable knots encode an assumption and a solution. Suppose a knot K with a root term t and $U^t = \mathsf{st}(K, t)$ is stable w.r.t. P. Moreover, suppose t occurs in a forest-shaped interpretation I for P, as a “leaf node”, i.e., there are no atoms of the form $R(t, f(t))$ in I. If the states of t in I and K coincide, i.e., $\mathsf{st}(I, t) = U^t$, then intuitively K becomes an eligible set of atoms that can be introduced in I to give t the necessary successors.

Example 3. (Continued) Consider the knots K_1, K_2 and K_3 from Figure 2. It is easy to see that there exists a stable model of P^{ex} where K_1 occurs, i.e., a stable model I such that $I \cap \mathcal{HB}_{g(b)} = K_1$. In fact, such a stable model is depicted in Figure 1. On the other hand, K_2 and K_3 do not occur in any stable model of P^{ex}, since the rules of P^{ex} do not force a domain element to satisfy both M and Y.

The knot K_1 is a stable knot. As easily checked, K_1 is a stable model of the local program $P^{ex}(\{M(g(b)), W(g(b))\})$. Even though K_2 does not occur in stable model of P^{ex}, it is a stable model of $P^{ex}(\{M(g(b)), Y(g(b)), W(g(b))\})$, and therefore is stable. Intuitively, K_2 is an eligible building block for a stable model of P^{ex} only if $g(b)$ satisfies exactly W and both M and Y. The knot K_3 is not a stable knot, since the stable models of $P^{ex}(\{Y(b), W(b)\})$ are $K_3 \setminus \{Y(g(b))\}$ and $K_3 \setminus \{Y(g(b)), W(g(b))\} \cup \{C(g(b))\}$.

After introducing the necessary notions for the tree-part of forest-shaped interpretations, we turn to the graph part.

Definition 3.6 (P^{G})**.** By P^{G} we denote the program $\mathsf{Ground}(P')$, where P' is obtained from P by removing all the rules containing function symbols.[2]

The following theorem characterizes the stable models of P. For an interpretation I, let I^c be the set of all atoms $A(\vec{c}) \in I$ such that $\vec{c}$ is a tuple of constants.

Theorem 1. *Let I be an interpretation for P. The following two are equivalent.*

(A) I is a stable model of P.

(B) I is a forest-shaped interpretation such that

(i) I^c is a stable model of P^{G}, and

[2]Note that P^{G} is finite since its Herbrand universe contains only the constants of P.

(ii) for each term $t \hat{\in} I$, $I \cap \mathcal{HB}_t$ is a knot that is stable w.r.t. P.

Proof. (A) $\Rightarrow$ (B). Assume I is a stable model of P. We show that (i) follows. Let $J := I^c$ and $Q := P^{\mathsf{G}}$. Suppose J is a not a stable model of the program Q, i.e., J not a minimal model of Q^J. There are two possibilities:

- J is not a model of Q^J. Then there exists a rule r in Q^J such that $\mathsf{body}(r) \subseteq J$ and $\mathsf{head}(r) \cap J = \emptyset$. This rule, by the definition, is grounded to constants only. Since, J is a restriction of I to the atoms that only have constants as arguments, J and I agree on the reduct for the rules ground with constants only. Hence, $r \in P^I$. For the same reason, $\mathsf{body}(r) \subseteq I$ and $\mathsf{head}(r) \cap I = \emptyset$, which implies that I is not a model of the reduct P^I, i.e., I is not a stable model of P.

- J is a model of Q^J, but is not minimal, i.e., there exists $H \subset J$ such that H is a model of Q^J. First, notice that by the definition of P^{G} and due to the fact that J is I^c, it holds that $(*)$ $r \in Q^J$ iff $r \in P^I$ and r is ground to constants only. Define an interpretation $M := H \cup (I \setminus J)$. Obviously, M is such that $M \subset I$. We verify that M is a model of P^I. Indeed, M represents an interpretation obtained from I by removing some constant-ground atoms. Due to the syntax of $\mathbb{FDNC}$ programs, M could potentially violate only a rule r in P^I that is ground to constants only. However, by $(*)$, $r \in Q^J$ and we assumed that H is a model of Q^J. We arrive at a contradiction.

In the similar fashion we show that (ii) follows. Suppose $t \hat{\in} I$ and $K := I \cap \mathcal{HB}_t$. That K is a knot over the signature of P follows from the fact that I is a forest-shaped (see Proposition 1). Suppose K is not stable w.r.t. P, i.e., K is not a stable model of $P(U)$, where $U = \mathsf{st}(K, t)$. There are two possibilities:

- K is not a model of $P(U)^K$. Then there exists a rule $r \in P(U)^K$ such that $\mathsf{body}(r) \subseteq K$ and $\mathsf{head}(r) \cap K = \emptyset$. It cannot be a fact since, by definition, each fact $A(t) \leftarrow$ is in $P(U)$ iff $A(t) \in K$. Then $r \in P_t$, where P_t denotes the program obtained from $P(U)$ by removing the facts. By the construction of local programs, $P_t \subseteq \mathsf{Ground}(P)$. Since $K = I \cap \mathcal{HB}_t$, K and I agree on the reduct for the rules in P_t and the interpretation of their atoms. This implies $r \in P^I$, $\mathsf{body}(r) \subseteq I$ and $\mathsf{head}(r) \cap I = \emptyset$. Therefore, I is not a stable model of P.

- K is a model of $P(U)^K$, but is not minimal, i.e., there exists $H \subset K$ such that H is a model of $P(U)^K$. Define a new interpretation $M := H \cup (I \setminus K)$. Obviously, $M \subset I$. In the following we show that M is a model of P^I, and, hence, I is not a stable model of P. Suppose M is not a model of P^I. Then there is a violated rule $r \in P^I$ such that $\mathsf{body}(r) \subseteq M$ and $\mathsf{head}(r) \cap M = \emptyset$. Since I is a model of P^I and M is not, then r is of one of the following forms:

  (a) $A_1(t) \vee \ldots \vee A_n(t) \leftarrow B_0(t), \ldots, B_m(t)$,
  (b) $R_1(t, f_0(t)) \vee \ldots \vee R_n(t, f_n(t)) \leftarrow P_0(t, g_0(t)), \ldots, P_m(t, g_m(t))$,
  (c) $A_1(f(t)) \vee \ldots \vee A_n(f(t)) \leftarrow B_0(f(t)), \ldots, B_m(f(t))$,
  (d) $A_1(f(t)) \vee \ldots \vee A_n(f(t)) \leftarrow B_0(Z_0), \ldots, B_m(Z_m), R_0(t, f(t)), \ldots, R_k(t, f(t))$, or
  (e) $R_1(t, f_0(t)) \vee \ldots \vee R_n(t, f_n(t)) \leftarrow B_0(t), \ldots, B_m(t)$,

  where each $Z_i \in \{t, f(t)\}$, $n \geq 0$, and $m, k > 0$. Suppose r is of the form (a). Then $K \setminus H$ contains an atom $A(t)$, for some unary predicate symbol A. It follows that H is not a model of $P(U)^K$. This is due to the fact that $P(U)^K$ contains $A(t) \leftarrow$ by the definition of local programs.

Therefore r is of type (b), (c), (d), or (e). Due to $K = I \cap \mathcal{HB}_t$ and the definition of $P(U)$, it follows that $r \in P(U)^K$. Due to $\mathsf{body}(r) \subseteq M$, $M = H \cup (I \setminus K)$, and the atoms that may occur in the body of r, we have $\mathsf{body}(r) \subseteq H$. Furthermore, due to $\mathsf{head}(r) \cap M = \emptyset$, we have $\mathsf{head}(r) \cap H = \emptyset$. Contradiction to the assumption that H is a model of $P(U)^K$.

(B) $\Rightarrow$ (A). Suppose (B) holds, but I is not a stable model of P, i.e., I is not a minimal model of P^I. Again, there are two possibilities:

- I is not a model of P^I. Then there exists a rule r in P^I such that $\mathsf{body}(r) \subseteq I$ and $\mathsf{head}(r) \cap I = \emptyset$. It cannot be the case that r is ground to constants only. This is due to the fact that I^c is a stable model of P^{G} and that r is included in the reduct of P^{G} w.r.t. I^c. Satisfaction of the rest of the rules follows directly from the fact that, for each term $t \hat{\in} I$, $K := I \cap \mathcal{HB}_t$ is a knot that is stable w.r.t. P.

- I is a model of P^I, but is not minimal, i.e., there exists $H \subset I$ such that H is a minimal model of P^I. Due to forest-shaped model property, H is forest-shaped. If $H^c \subset I^c$, then I^c is not a stable model of P^{G}. Then it has to be the case that $H^c = I^c$ and there exists some term t satisfying the following 2 conditions.

  (a) It holds that:

    (I) $A(t) \in I$ and $A(t) \notin H$, for some unary predicate symbol A, and t is not a constant, or

    (II) $R(t,s) \in I$ and $R(t,s) \notin H$, for some binary predicate symbol R and a term s,

  (b) Each subterm v of t violates (a).

  Intuitively, t is some (smallest w.r.t. depth) term where I and H disagree on the interpretation of atoms. Suppose t satisfies (I) (and possibly (II)), and is of the form $f(s)$. By assumption, $K := I \cap \mathcal{HB}_s$ is stable w.r.t. P. Due to the selection of t, $K' := H \cap \mathcal{HB}_s$ is a knot such that $K' \subset K$ and $\mathsf{st}(K', s) = \mathsf{st}(K, s) =: U$. It is easy to verify that since H is a model of P^I, then K' is a model of $P(U)^K$, and, hence, K is not stable w.r.t. P. Contradiction. Suppose t does not satisfy (I) but satisfies (II). Again, by assumption, $K := I \cap \mathcal{HB}_t$ is stable w.r.t. P. Due to the selection of t and failure of (II), $K' := H \cap \mathcal{HB}_t$ is a knot such that $K' \subset K$ and $\mathsf{st}(K', t) = \mathsf{st}(K, t) =: U$. Again, if H is a model of P^I, then K' is a model of $P(U)^K$, and, hence, K is not stable w.r.t. P. Contradiction.

□

3.2 Finite Representation of Stable Models

By the semantic characterization of the stable models of an $\mathbb{FDNC}$ program from above, we may view them as being composed of stable knots. More precisely, we show that Theorem 1 allows us to obtain a finite representation of the stable models, which is based on the observation that although infinitely many knots might occur in some stable model of a program, only finitely many of them are non-isomorphic modulo the root term.

Definition 3.7 ($K_{\downarrow u}$)**.** Let K be a knot with a root term t. By $K_{\downarrow u}$ we denote the knot obtained from K by replacing each occurrence of t in K with a term u.

Indeed, if the program P has an infinite stable model I, then the set of knots $L := \{(I \cap \mathcal{HB}_t) \mid t \hat{\in} I\}$ is infinite. However, for a fixed term t, the set $L' := \{K_{\downarrow t} \mid K \in L\}$ is finite as there are only finitely many knots with the root term t over the signature of P. Intuitively, if we view t as a variable, then each $K \in L$ can be viewed as an instance of some knot in L'.

To talk about sets of knots with a common root term, we assume a special constant $\mathbf{x}$ not occurring in any $\mathbb{FDNC}$ program. We call a set L of knots $\mathbf{x}$-*grounded*, if all its knots have the root term $\mathbf{x}$. The following notion collects the knots occurring in a stable model and abstracts them using $\mathbf{x}$.

Definition 3.8 (Scanning)**.** Let I be a forest-shaped interpretation for P. We define the set of $\mathbf{x}$-grounded knots as $\mathbb{K}(I) := \{(I \cap \mathcal{HB}_t)_{\downarrow \mathbf{x}} \mid t \hat{\in} I\}$.

In the following, we show that $\mathbf{x}$-grounded sets of knots can be used to represent the stable models of an $\mathbb{FDNC}$ program. First, we observe that the stability of a knot is preserved under substitutions.

Proposition 2. *If K is a knot that is stable w.r.t. P, and u is an arbitrary term, then $K_{\downarrow u}$ is stable w.r.t. P.*

Proof. Indeed, if we consider a stable model of a propositional program, then the global renaming of propositional atoms in the model and in the program preserves the property. □

We introduce a notion of *founded* sets of $\mathbf{x}$-grounded knots. The intention is to capture the properties of the set $\mathbb{K}(I)$ when I is a stable model of P. To this end, we need a notion of *state equivalence* as a counterpart for substitutions in knots. Formally, states U^t and V^s are *equivalent* (in symbols, $U^t \approx V^s$), if $U^t = \{A(t) \mid A(s) \in V^s\}$, i.e., in both states terms satisfy the same unary predicates.

Definition 3.9 (Founded Knot Set)**.** Let $S \neq \emptyset$ be a set of states. A set L of $\mathbf{x}$-grounded knots that are stable w.r.t. the program P is *founded* w.r.t. P and S, if the following hold:

1. For each $U \in S$, there exists $K \in L$ such that $U \approx \mathsf{st}(K, \mathbf{x})$.
2. For each $K \in L$, the following hold:
   a. for each $s \in \mathsf{succ}(K)$, there exists $K' \in L$ s.t. $\mathsf{st}(K, s) \approx \mathsf{st}(K', \mathbf{x})$, and
   b. there exists a sequence $\langle K_0, \dots, K_n \rangle$ of knots in L such that:
      - $K_n = K$,
      - K_0 is such that $\mathsf{st}(K, \mathbf{x}) \approx U$ for some $U \in S$, and
      - for each $0 \leq i < n$, there exists $s \in \mathsf{succ}(K_i)$ s.t. $\mathsf{st}(K_i, s) \approx \mathsf{st}(K_{i+1}, \mathbf{x})$.

For an interpretation I, let $S(I)$ denote the set of states of constants occurring in I, i.e., $S(I) := \{\mathsf{st}(I, c) \mid c \hat{\in} I \text{ is a constant}\}$. The following is easy to verify.

Proposition 3. *Let $I \in SM(P)$. Then $\mathbb{K}(I)$ is a set of knots that is founded w.r.t. P and $S(I^c)$.*

In what follows, we provide a construction of stable models out of knots in a founded set. Moreover, we show that for a given consistent program there exists a founded set of knots that captures all the stable models.

Generating Stable Models out of Knots

We state formally the construction of forest-shaped interpretations out of knots in a founded set. To this end, we first state the construction of trees which are represented in the standard way by prefix-closed sets of words. For a sequence of elements $p = [e_1, \ldots, e_n]$, let $\tau(p)$ denote the last element e_n, and $[p|e_{n+1}]$ denote the sequence $[e_1, \ldots, e_n, e_{n+1}]$.

Definition 3.10 (Tree Construction)**.** Let L be a set of knots that is founded w.r.t. P and a set of states S, and let U^t be a state such that $U^t \approx V$, for some $V \in S$. A set T of sequences, where each element in a sequence is a tuple of a knot and a term, is called a *tree induced by L starting at U^t*, if the following hold:

(a) $[\langle K, t\rangle] \in T$, where $K \in L$ is s.t. $\mathsf{st}(K, \mathbf{x}) \approx U^t$.

(b) If there exists $p \in T$ with $\tau(p) = \langle K, t\rangle$ and $f(\mathbf{x}) \in \mathsf{succ}(K)$, then there exists $[p|\langle K', f(t)\rangle] \in T$, where K' is a knot in L s.t. $\mathsf{st}(K, f(\mathbf{x})) \approx \mathsf{st}(K', \mathbf{x})$.

(c) T is minimal, i.e., each $T' \subset T$ violates (a) or (b).

We state the transformation of trees into Herbrand interpretations.

Definition 3.11 ($T_\downarrow$)**.** Let T be a tree induced by a founded set of knots L starting at some state. We define the set of atoms $T_\downarrow := \{K_{\downarrow t} \mid p \in T \text{ with } \tau(p) = \langle K, t\rangle\}$.

We generalize the construction of trees to forest-shaped interpretations.

Definition 3.12 (Forest Construction)**.** Let G be a set of atoms ground with constants of program P only, and L be a set of knots founded w.r.t. P and a set of states $S \supseteq S(G)$. Then $\mathcal{F}(G, L)$ is the largest set of forest-shaped interpretations

$$I = G \cup (T^{c_1})_\downarrow \cup \ldots \cup (T^{c_n})_\downarrow,$$

where $\{c_1, \ldots, c_n\}$ is the set of all constants occurring in G and each T^{c_i} a tree induced by L starting at $\mathsf{st}(G, c_i)$.

$\mathcal{F}(G, L)$ represents all the interpretations that can be build from G by attaching, for each of the constants, a tree induced by L.

Theorem 2. *If $G \in SM(P^{\mathsf{G}})$, L is a set of knots that is founded w.r.t. P and some $S \supseteq S(G)$, then $\mathcal{F}(G, L) \neq \emptyset$ and each $I \in \mathcal{F}(G, L)$ is a stable model of P.*

Proof. Indeed, $\mathcal{F}(G, L) \neq \emptyset$ due to foundedness of L. Assume some $I \in \mathcal{F}(G, L)$. Each $K \in L$ is stable w.r.t. P. Then due to Proposition 2, for each term $t \hat{\in} I$, $I \cap \mathcal{HB}_t$ is a knot that is stable w.r.t. P. Keeping in mind that $G \in SM(P^{\mathsf{G}})$, Theorem 1 implies that I is a stable model of P. □

We showed that stable model existence can be proved by checking that some founded set of knots exists. As we see next, the properties of founded sets of knots imply that we can obtain a set capturing all the stable models of a program.

Capturing Stable Models

The following property of founded sets of knots is obvious.

Proposition 4. *Let L_1 and L_2 be sets of knots founded w.r.t. P and sets of states S_1 and S_2 respectively. Then $L_1 \cup L_2$ is founded w.r.t. P and $S_1 \cup S_2$.*

At this point, we introduce a founded set of knots, which will capture all the stable models. First, let $S(P)$ denote the set of states that occur in the stable models of P^{G}, i.e., $S(P) := \{\mathsf{st}(G, c) \mid G \in SM(P^{\mathsf{G}}) \wedge c \hat{\in} G\}$.

Definition 3.13 ($\mathbb{K}_P$)**.** We denote by $\mathbb{K}_P$ the smallest set of knots which contains every set of knots L that is founded w.r.t. P and some $S \subseteq S(P)$.

Due to Proposition 4 and Definition 3.13, the following is immediate.

Proposition 5. *For the program P, the following hold:*

(a) If $\mathbb{K}_P \neq \emptyset$, then $\mathbb{K}_P$ is founded w.r.t. P and some $S \subseteq S(P)$.

(b) If L is a set of knots that is founded w.r.t. P and some $S \subseteq S(P)$, then $\mathbb{K}_P$ is founded w.r.t. P and some $S' \supseteq S$.

(c) Each $L \supset \mathbb{K}_P$ is not founded w.r.t. P and any $S \subseteq S(P)$.

It is easy to verify that a stable model I can be reconstructed out of knots in $\mathbb{K}(I)$. Naturally, the same holds for any superset of $\mathbb{K}(I)$ satisfying Definition 3.9.

Proposition 6. *If I is a stable model of P, then $I \in \mathcal{F}(I^c, L)$ for each set of knots $L \supseteq \mathbb{K}(I)$ s.t. L is founded w.r.t. P and some set of states $S \supseteq S(I^c)$.*

The following will be helpful.

Definition 3.14 (Compatible $\mathbb{K}_P$)**.** We say $\mathbb{K}_P$ is *compatible* with a set of states S, if for each state $U \in S$, there exists $K \in \mathbb{K}_P$ s.t. $U \approx \mathsf{st}(K, \mathbf{x})$.

The crucial property of $\mathbb{K}_P$ is that it captures the tree-structures of all the stable models of P. Together with the stable models of P^{G}, it represents the stable models of P.

Theorem 3. *Let I be an interpretation for P. Then, $I \in SM(P)$ iff $I \in \mathcal{F}(G, \mathbb{K}_P)$, for some $G \in SM(P^{\mathsf{G}})$ s.t. $\mathbb{K}_P$ is compatible with $S(G)$.*

Proof. If $I \in SM(P)$, then, by Proposition 3, $\mathbb{K}(I)$ is founded w.r.t. P and $S(I^c)$. By definition, $\mathbb{K}(I) \subseteq \mathbb{K}_P$. By Proposition 5, $\mathbb{K}_P$ is founded w.r.t. P and some $S \supseteq S(I^c)$. By Proposition 6, $I \in \mathcal{F}(I^c, \mathbb{K}_P)$. The other direction is proved by Theorem 2. ☐

We have obtained a finite representation of the stable models of a $\mathbb{FDNC}$ program P. Indeed, each of its stable models can be generated out of some stable model of P^{G} and a set of knots $\mathbb{K}_P$. We can view P^{G} together with $\mathbb{K}_P$ as a compilation of the logic program P that can be exploited for reasoning and stable model building. We will discuss this further in Sections 5 and 6, where computational cost is addressed.

Problem	$\mathbb{F}$	$\mathbb{FD}$	$\mathbb{FC}$	$\mathbb{FDC}$, $\mathbb{FN}$, $\mathbb{FNC}$, $\mathbb{FDNC}$
Consistency	Trivial	Trivial	PSPACE (6.2)	EXPTIME (5.2, 6.1)
$P \models_b A(\vec{t})$	P (6.3)	Σ_2^P (6.3)	PSPACE (6.2)	EXPTIME (5.3)
$P \models_b \exists\vec{x}.A(\vec{x})$	PSPACE (6.3)	PSPACE (6.3)	PSPACE (6.2)	EXPTIME (5.3)
$P \models_c A(\vec{t})$	P (6.3)	CO-NP (6.3)	PSPACE	EXPTIME
$P \models_c \exists\vec{x}.A(\vec{x})$	PSPACE	EXPTIME	PSPACE	EXPTIME
$P \models_c \lambda\vec{x}.A(\vec{x})$	PSPACE	EXPSPACE (5.4)	PSPACE	EXPSPACE (5.4)

Table 1: Complexity of $\mathbb{FDNC}$ and Fragments (Completeness Results)

4 Complexity Results

This section gives a brief overview of our results on the complexity of the main reasoning tasks in $\mathbb{FDNC}$ and its fragments, which are compactly summarized in Table 1. An in-depth analysis and the reasoning techniques for the derivation are given in the following two sections. Here, we give some intuition behind the results and discuss how some of them can be derived from a core of results.

As shown in the previous section, $\mathbb{FDNC}$ programs have forest-shaped stable models. Naturally, reasoning in $\mathbb{FDNC}$ involves construction of forest-shaped interpretations (in the following, *forests*). Consistency testing involves building a forest-shaped stable model, while brave/cautious reasoning requires checking whether some property holds in some/all stable models that can be built. However, an $\mathbb{FDNC}$ program may have infinitely large stable models, and therefore the construction has to employ some direct or indirect blocking technique to stop the construction after sufficient information is acquired.

The forest-shape model property implies that blocking of the model construction is feasible and, hence, the decidability of $\mathbb{FDNC}$ for major reasoning tasks can be established. Indeed, a continuous construction of a forest will lead to re-occurrences of patterns, e.g., states of terms, non-isomorphic labeled arcs, or non-isomorphic trees of depth 1, etc. To give the algorithms for $\mathbb{FDNC}$, we could resort to the methods of Description Logics (DLs), which usually have a forest-shape model property, and are usually decided by Tableaux methods with blocking. Unfortunately, such methods are not very suitable for our case. First, they cannot easily handle minimality testing, and are generally not worst-case optimal. Second, Tableaux methods are designed for consistency testing, while some important tasks from non-monotonic reasoning, e.g., brave reasoning, cannot be reduced to consistency testing.

Therefore, our algorithms for $\mathbb{FDNC}$ rely on the finite representation of stable models in terms of maximal founded sets of knots. In Section 5.1, we show how to derive the set $\mathbb{K}_P$ of knots for a given $\mathbb{FDNC}$ program P in single exponential time in the size of P. This is possible as the number of distinct $\mathbf{x}$-grounded knots is bounded by a single exponential. Given $\mathbb{K}_P$, several standard reasoning tasks can be solved in time polynomial in the size of $\mathbb{K}_P$; hence, overall they are in EXPTIME. This includes consistency testing (Section 5.2), brave entailment of ground and existential queries (Section 5.3), as well as cautious entailment of ground and existential queries (which is easily reduced to consistency testing). These upper bounds are tight for $\mathbb{FDNC}$. It is easy to see that a decision procedure needs to explore forests whose depths are bounded by a single exponential in the size of the input program. However, due to the disjunction or non-monotonic negation in an $\mathbb{FDNC}$ program, the number of such candidate forests may be too high for a procedure to traverse them in polynomial space.TE: Intuition for EXPTIME-hardness is not fully convincing. The ExpTime-hardness of consistency testing is proved in Section 5.2 by an encoding of an ExpTime-hard Description Logic $\mathcal{ALC}$, which is extended to $\mathbb{FN}$ in Section 6.1. The hardness of consistency testing directly provides lower bounds for brave and cautious entailment of ground and existential queries.

For the fragment $\mathbb{FC}$ of $\mathbb{FDNC}$, the picture is different. Such programs have the unique model property, i.e., if a stable model exists, it is unique. For the standard reasoning tasks, this implies that a procedure needs to navigate a unique forest searching for a node with a certain property, e.g., the one that causes an inconsistency, or satisfies a query. Furthermore, the procedure needs to navigate only the depths bounded by a single exponential. Our algorithms navigate the forest by non-deterministically guessing the paths through function symbols and building necessary parts of a stable model. They run in polynomial space and can, by Savitch's result [46], turned into deterministic polynomial space algorithms. The PSpace-hardness of consistency testing is shown by an encoding of PSpace Turing machines, which is extended to other standard reasoning tasks (see Section 6.2 for more details).

If we disallow non-monotonic negation and constraints, the complexity drops even more. Consistency testing in both $\mathbb{F}$ and $\mathbb{FD}$ is trivial, while the complexity of ground entailment drops to lower levels of the polynomial hierarchy, and corresponds to the complexity of propositional logic programming. This is because the consistency needs not be ensured, and the necessary conditions can be verified locally within polynomial distance from the graph part of the input program. Section 6.3 discusses the results for $\mathbb{F}$ and $\mathbb{FD}$.

The last row in Table 1 lists the complexity of open queries. Deciding cautious entailment of open queries in $\mathbb{FDNC}$ is ExpSpace-complete and thus harder than cautious entailment of existential queries. Intuitively, this is because to search for a term that satisfies a property in each stable model of a program, we must look at branches beyond single exponential length. However, the length can be bounded by a double exponential, and we can thus manage to answer the query in single exponential space; Section 5.4 provides the details.

The main entries in Table 1 are presented with the reference to the section that discusses the problem in detail. The remaining entries can be justified as follows:

(i) $\mathbb{F}$ and $\mathbb{FD}$ are Horn programs, and therefore are always consistent.

(ii) PSpace-hardness (resp. ExpTime-hardness) of $P \models_c \exists\vec{x}.A(\vec{x})$ in $\mathbb{F}$ and $\mathbb{FC}$ (resp. in $\mathbb{FD}$ and $\mathbb{FDC}$) follows since consistency checking with constraints in $\mathbb{FC}$ (resp. $\mathbb{FDC}$) can be reduced to cautious inference. On the other hand, completeness also follows because cautious inference can be reduced to inconsistency testing in the standard way.

(iii) Similarly, PSpace (resp. ExpTime) membership of $P \models_c A(\vec{t})$, where P is an $\mathbb{FC}$ program (resp. $\mathbb{FDC}$, $\mathbb{FN}$, $\mathbb{FNC}$ or $\mathbb{FDNC}$), is due to the fact that the task can be reduced to checking consistency of $P \cup \{\leftarrow A(\vec{t})\}$. On the other hand, hardness follows from the fact that P is inconsistent iff $P \models_c A'(t)$ where A' is a symbol not occurring in P and t is arbitrary.

(iv) PSpace-completeness of $P \models_c \lambda\vec{x}.A(\vec{x})$ in $\mathbb{F}$ and $\mathbb{FC}$ follows because these fragments have the unique stable model property, and hence cautions entailment of open and existential queries coincide; the latter is PSpace-complete.

To ease presentation, we use a lemma that allows us to concentrate on unary queries.

Lemma 1. *Let C be a complexity class in Table 1, and let $\mathcal{L}$ be from the $\mathbb{F}$ family. Then:*

(i) If deciding the consistency of a given program in $\mathcal{L}$ is C-hard, then deciding brave entailment of queries (ground or existential, unary or binary) is C-hard in $\mathcal{L}$ as well.

(ii) Brave entailment of unary existential (resp., ground) queries is C-complete for $\mathcal{L}$ iff brave entailment of binary existential (resp., ground) queries is C-complete for $\mathcal{L}$.

(iii) Cautious entailment of unary open queries is C-complete for $\mathcal{L}$ iff cautious entailment of binary open queries is C-complete for $\mathcal{L}$.

5 Complexity of $\mathbb{FDNC}$

This section discusses the complexity of reasoning in $\mathbb{FDNC}$ and provides worst-case optimal algorithms together with the matching hardness results. The methods for consistency testing, deciding brave entailment of ground and existential queries and cautious entailment of open queries rely on the finite representation of stable models in terms of the set $\mathbb{K}_P$ of knots which, together with the set $SM(P^{\mathsf{G}})$, captures all the stable models of P (see Theorem 3).

5.1 Deriving Maximal Founded Set of Knots

To derive $\mathbb{K}_P$, we proceed in two phases. In the first phase, we generate the set of knots $\mathsf{All}(P)$ that surely contains $\mathbb{K}_P$. In the second phase, we remove some knots from it to ensure that it satisfies Definition 3.13.

To ease the presentation, for a knot set L, let $\mathsf{states}(L) := \{\mathsf{st}(K, s) \mid K \in L,\ s \in \mathsf{succ}(K)\}$, i.e., $\mathsf{states}(L)$ is the set of all states of the successor terms of knots in L.

Definition 5.1 ($\mathsf{All}(P)$)**.** For an $\mathbb{FDNC}$ program P, let $\mathsf{All}(P)$ be the smallest set of $\mathbf{x}$-grounded knots satisfying the following conditions:

a) If $U \in S(P)$ and $K \in SM(P(U))$, then $K_{\downarrow\mathbf{x}} \in \mathsf{All}(P)$.

b) If $U \in \mathsf{states}(\mathsf{All}(P))$ and $K \in SM(P(U))$, then $K_{\downarrow\mathbf{x}} \in \mathsf{All}(P)$.

Intuitively, $\mathsf{All}(P)$ contains by construction each set of knots that is founded w.r.t. P and some set of states $S \subseteq S(P)$. The problem is that $\mathsf{All}(P)$ might contain a knot K such that some $s \in \mathsf{succ}(K)$ has no potential successor knot (see (2.a) in Definition 3.9). Such knots should be removed from $\mathsf{All}(P)$.The second phase deals with this problem.

Definition 5.2 ($\mathsf{reach}(L, S)$)**.** For any set of $\mathbf{x}$-grounded knots L and set of states S, $\mathsf{reach}(L, S)$ is the smallest set of knots such that:

a) if $U \in S$, $K \in L$ and $U \approx \mathsf{st}(K, \mathbf{x})$, then $K \in \mathsf{reach}(L, S)$, and

b) if $U \in \mathsf{states}(\mathsf{reach}(L, S))$, $K \in L$ and $U \approx \mathsf{st}(K, \mathbf{x})$, then $K \in \mathsf{reach}(L, S)$.

Intuitively, $\mathsf{reach}(L, S)$ are the knots in L reachable from the states in S. Indeed, if $\mathsf{reach}(L, S) = L$, then L fulfills condition (2.b) of Definition 3.9 to be founded w.r.t. S.

Theorem 4. *If P is an $\mathbb{FDNC}$ program, then* $\mathbb{K}_P = \mathsf{reach}(\mathsf{All}(P), S(P))$.

Proof. Let $L := \mathsf{reach}(\mathsf{All}(P), S(P))$. We verify that L satisfies the conditions in Definition 3.13, i.e., L is the single $\subseteq$-minimal set which contains each knot set L' that is founded w.r.t. P and some $S \subseteq S(P)$.

Indeed, $L' \subseteq \mathsf{All}(P)$ by construction. Moreover, due to the definition of reach, we have $L' \subseteq L$. Suppose L is not minimal, i.e., there exists some $N \subset L$ that contains every knot set L that is founded w.r.t. P and some $S \subseteq S(P)$. Then L must be nonempty. It follows that L is founded w.r.t. P and some $S \subseteq S(P)$. First, the definition of $\mathsf{All}(P)$ ensures that all the knots in L are stable. Second, reach ensures that every knot in L has proper successors to satisfy (2.a) in Definition 3.9, and has a proper sequence of predecessors to satisfy (2.b) reaching a state in $S(P)$. By assumption on N and the foundedness of L, we have $L \subseteq N$. This, however, contradicts $N \subset L$. Thus L satisfies Definition 3.13, i.e., $\mathbb{K}_P = L$. □

It is easy to see that to compute $\mathbb{K}_P$ we need time at most single exponential in the size of an $\mathbb{FDNC}$ program P. The claim is immediate from the following observations:

– The number of $\mathbf{x}$-grounded knots over P is bounded by a single exponential in the size of P. More precisely, the number is bounded by $max = 2^{n+k\cdot(n+m)}$, when P has k function, n unary, and m binary predicate symbols.

– Computing $\mathsf{All}(P)$ requires adding at most max $\mathbf{x}$-grounded knots. Each such knot has polynomial size and its stability is verifiable using an $\Sigma_2^P = \mathrm{NP}^{\mathrm{NP}}$ oracle. Thus, $\mathsf{All}(P)$ is computable in time single exponential in the size of P.

– The size of $S(P)$ is bounded by a single exponential in the size of P.

– Computing $\mathsf{reach}(L, S)$ is polynomial in the combined size of L and S. Hence, $\mathsf{reach}(\mathsf{All}(P), S(P))$ can be computed in time that is polynomial in the size of $\mathsf{All}(P)$ and $S(P)$.

5.2 Deciding Consistency

Once the set $\mathbb{K}_P$ for an $\mathbb{FDNC}$ program P is derived, it can be readily used for consistency testing. We will see that the resulting algorithm is worst-case optimal.

Theorem 5. *For every $\mathbb{FDNC}$ program P, the following are equivalent:*

(i) P is consistent.

(ii) For some $G \in SM(P^{\mathsf{G}})$, the set $\mathbb{K}_P$ is compatible w.r.t. $S(G)$.

Proof. If I is a stable model of P, then by Theorem 3 there exists some $G \in SM(P^{\mathsf{G}})$ such that $\mathbb{K}_P$ is compatible w.r.t. $S(G)$. The other direction is proved by Theorem 2. □

By this theorem, to decide consistency of P we can search for a stable model G of the program P^{G} such that for each constant of P, $\mathbb{K}_P$ can start the tree construction (i.e., $\mathbb{K}_P$ is compatible with $S(G)$). We obtain the following result.

Theorem 6. *Deciding whether a given $\mathbb{FDNC}$ program is consistent is in* ExpTime.

Proof. Deciding whether $\mathbb{K}_P$ is compatible w.r.t. $S(G)$, for some $G \in SM(P^{\mathsf{G}})$, is feasible in time polynomial in $n + m$, where m is the size of $\mathbb{K}_P$ and n is the size of $SM(P^{\mathsf{G}})$. Overall, this can be done in time single exponential in the size of P, since both m and n are single exponential in the size of P. Since $SM(P^{\mathsf{G}})$ is computable in single exponential time, the result follow from Theorem 5. □

As we have pointed earlier already, we can see $\mathbb{K}_P$ together with P^{G} as a compilation of the $\mathbb{FDNC}$ program P. Out of this compilation, we can gradually build a stable model of P by continuing the tree construction for some stable model G of P^{G} using knots from $\mathbb{K}_P$ (and every stable model of P results by proper choices). Here the hard part is computing a stable model $G \in SM(P^{\mathsf{G}})$, which depending on the complexity of function-free logic programs is Σ_2^P-hard already for $\mathbb{FD}$, NP-hard already for $\mathbb{FN}$, and polynomial for $\mathbb{F}$ and $\mathbb{FC}$. Checking the compatibility of $\mathbb{K}_P$ with $S(G)$ is polynomial, and each tree expansion step using a knot from $\mathbb{K}_P$ is feasible with low computational (clearly polynomial) cost. Note that this model-building technique is complementary to computing a stable model of an ordinary (function-free) logic program, and may be realized on top of traditional stable model engines (like DLV or Smodels).

In the following, we show that the algorithm emerging from Theorem 5 is worst-case optimal. The proof is by a polynomial-time translation of consistency testing in the Description Logic $\mathcal{ALC}$, which is ExpTime-hard, to consistency testing in $\mathbb{FDC}$. The translation is interesting in its own right, as it provides a translation of the core of expressive Description Logics into logic programming.

Definition 5.3 ($\mathcal{ALC}$ Syntax)**.** Let $\mathbf{C} \supseteq \{\top, \bot\}$, $\mathbf{R}$, and $\mathbf{I}$ denote the sets of *concept names*, *role names*, and *individual names*, respectively. *Concepts* are define inductively: (a) every concept name in $\mathbf{C}$ is a concept, and (b) if C, D are concepts and R is a role, then $C \sqcap D$, $C \sqcup D$, $\neg C$, $\forall R.C$, and $\exists R.C$ are also concepts. If A is an atomic concept, then A and $\neg A$ are *literal concepts.*

A *general concept inclusion axiom* (GCI) is an expression $C \sqsubseteq D$ where C, D are concepts. An *assertion* is an expression $C(a)$ or $R(a, b)$, where $a, b \in \mathbf{I}$, R is a role name, and C is a concept name. An *$\mathcal{ALC}$ knowledge base* is a finite set of GCIs and assertions.

Each $\mathcal{ALC}$ knowledge base $\mathcal{K}$ has a model-theoretic semantics, which is given via a mapping of $\mathcal{K}$ into a set of sentences in first-order logic, shown in Figure 3 (see e.g. [33] for details). The major reasoning task in $\mathcal{ALC}$ is deciding the *consistency* of a given $\mathcal{K}$, i.e., of the first-order theory $\Theta(\mathcal{K})$.

We now provide a polynomial time translation of *normalized* $\mathcal{ALC}$ knowledge bases $\mathcal{K}$ into $\mathbb{FDC}$ programs $P^{\mathcal{K}}$ such that $\mathcal{K}$ is consistent iff $P^{\mathcal{K}}$ is consistent. Normalized knowledge bases obey certain structural constraints which makes presenting the translation easier.

Definition 5.4 ($\mathcal{ALC}$ Normal Form)**.** An $\mathcal{ALC}$ knowledge base $\mathcal{K}$ is in *normal form*, if its GCI axioms are of one of the following forms:

(T1) $A_0 \sqcap \ldots \sqcap A_n \sqsubseteq B_0 \sqcup \ldots \sqcup B_m$,

(T2) $A_0 \sqcap \ldots \sqcap A_n \sqsubseteq \bot$,

(T3) $\top \sqsubseteq B_0 \sqcup \ldots \sqcup B_m$,

(T4) $A_0 \sqsubseteq \exists R.B_0$, or

(T5) $A_0 \sqsubseteq \forall R.B_0$,

Mapping knowledge base $\mathcal{K}$	
$\Theta(\mathcal{K})$	$=\bigcup_{\alpha\in\mathcal{K}}\{\Pi(\alpha)\}$
Mapping axioms	
$\Pi(C \sqsubseteq D)$	$=(\forall x)(\pi(C,x) \rightarrow \pi(D,x))$
$\Pi(C(a))$	$=p_C(a)$
$\Pi(R(a,b))$	$=p_R(a,b)$
Mapping concepts and roles	
$\pi(A,X)$	$=p_A(X)$
$\pi(\neg C,X)$	$=\neg\pi(C,X)$
$\pi(\top,X)$	$=\top$
$\pi(\bot,X)$	$=\bot$
$\pi(C \sqcup D,X)$	$=\pi(C,X) \vee \pi(D,X)$
$\pi(C \sqcap D,X)$	$=\pi(C,X) \wedge \pi(D,X)$
$\pi(\forall R.C,X)$	$=(\forall y)(p_R(X,y) \rightarrow \pi(C,y))$
$\pi(\exists R.C,X)$	$=(\exists y)(p_R(X,y) \wedge \pi(C,y))$

Note: X is a meta-variable that is replaced by an actual variable.

Figure 3: Semantics of the DL $\mathcal{ALC}$ by mapping to first-order logic

where $n, m > 0$, and each A_i and B_j is atomic, but is neither $\top$ nor $\bot$.[3] Additionally, if $\mathcal{K}$ is in normal form and does not contain axioms of type (T3), then $\mathcal{K}$ is *safe*.

Importantly, we can normalize any $\mathcal{ALC}$ knowledge base $\mathcal{K}$ efficiently.

Proposition 7. *Given any $\mathcal{ALC}$ knowledge base $\mathcal{K}$, we can obtain in linear time a safe knowledge base $\mathcal{K}'$ in normal form such that $\mathcal{K}$ is consistent iff $\mathcal{K}'$ is consistent.*

The proof, which is based on well-known *definitional form transformations*, is given in the appendix.

Applying Θ to a knowledge base $\mathcal{K}$ in normal form leads us close to the syntax of $\mathbb{FDC}$ programs. However, the rules of type (T3), in which $\top$ is the only concept in the antecedent of an axiom, causes a problem. Indeed, if $\mathcal{K}$ contains some axiom $\top \sqsubseteq A$, then $\Theta(\mathcal{K})$ contains the formula $\forall x.A(x)$, which leads to $A(x) \leftarrow$ in the rule representation. Non-ground rules containing empty bodies, which are unsafe, are not allowed in $\mathbb{FDNC}$ programs, and therefore we require safety of $\mathcal{ALC}$ knowledge bases.

We are now ready to define the translation. For any safe knowledge base $\mathcal{K}$ in normal form, let $P^{\mathcal{K}}$ denote the $\mathbb{FDNC}$ program that results after applying the translation rules in Table 2.

Proposition 8. *Let $\mathcal{K}$ be a safe knowledge base in normal form. Then $\mathcal{K}$ is consistent iff $P^{\mathcal{K}}$ is consistent.*

Proof. It is easy to verify that $P^{\mathcal{K}}$ is a rule-representation of the first-order theory that is obtained from $\Theta(\mathcal{K})$ by applying skolemization and a satisfiability preserving transformation for the axioms of type (T4). By Herbrand's Theorem [28], $P^{\mathcal{K}}$ is consistent iff $\Theta(\mathcal{K})$ consistent. □

[3]A similar normal form for the weaker Description Logic $\mathcal{EL}^{++}$ has been described in [4].

	Axioms of $\mathcal{K}$	Rules of $P^{\mathcal{K}}$
(T1)	$A_0 \sqcap \ldots \sqcap A_n \sqsubseteq B_0 \sqcup \ldots \sqcup B_m$	$B_0(x) \vee \ldots \vee B_m(x) \leftarrow A_0(x), \ldots, A_n(x)$
(T2)	$A_0 \sqcap \ldots \sqcap A_n \sqsubseteq \bot$	$\leftarrow A_0(x), \ldots, A_n(x)$
(T4)	$A \sqsubseteq \exists R.C$	$R'(x, f(x)) \leftarrow A(x)$ $R(x, y) \leftarrow R'(x, y)$ $C(y) \leftarrow R'(x, y)$
(T5)	$A \sqsubseteq \forall R.C$	$C(y) \leftarrow A(x), R(x, y)$
	$A(a)$	$A(a) \leftarrow;$
	$R(a, b)$	$R(a, b) \leftarrow;$
where $n \geq 0$, f is fresh function symbol, R' is a fresh binary predicate symbol.		

Table 2: Translating $\mathcal{ALC}$ into $\mathbb{FDNC}$

Note that $P^{\mathcal{K}}$ is in fact positive and constructible in linear time from $\mathcal{K}$. Hence, Propositions 7 and 7 and the well-known ExpTime-hardness of $\mathcal{ALC}$ [47] imply that deciding consistency of $\mathbb{FDC}$ and $\mathbb{FDNC}$ programs is ExpTime-hard. Combined with Theorem 6, we establish the completeness result.

Theorem 7. *For $\mathbb{FDC}$ and $\mathbb{FDNC}$ programs, checking consistency is* ExpTime*-complete.*

5.3 Brave Entailment of Queries

As we did for consistency checking, we exploit the set $\mathbb{K}_P$ for a program P to provide algorithms for brave reasoning. We first discuss entailment of existential unary atomic queries, and then move to ground queries. The general intuition behind the method is to perform some "back-propagation" of unary predicate symbols in the set of knots.

Definition 5.5 ($\mathcal{E}_L$)**.** Let L be a set of knots founded w.r.t. a $\mathbb{FDNC}$ program P and a set of states S. Let C be the set of unary predicate symbols occurring in P. By $\mathcal{E}_L$ we denote the smallest relation over $L \times C$ closed under the following rules:

(a) if $K \in L$ and some $A(\mathbf{x}) \in K$, then $\langle K, A\rangle \in \mathcal{E}_L$, and

(b) if $K' \in L$ is a possible successor of $K \in L$, and $\langle K', A\rangle \in \mathcal{E}_L$, then $\langle K, A\rangle \in \mathcal{E}_L$.

Intuitively, $\langle K, A\rangle \in \mathcal{E}_L$ means that, starting from K, a sequence of possible successor knots will eventually reach a knot containing $A(\mathbf{x})$. Since $\mathbb{K}_P$ together with $SM(P^{\mathsf{G}})$ capture the stable models of P, we have the following:

Theorem 8. *Let P be an $\mathbb{FDNC}$ program. The following two are equivalent.*

(A) $P \models_b \exists x.A(x)$.

(B) There exists some $G \in SM(P^{\mathsf{G}})$ such that the following hold:

(i) $\mathbb{K}_P$ is compatible w.r.t. $S(G)$, and

(ii) for some constant c and $K \in \mathbb{K}_P$, $\mathsf{st}(G, c) \approx \mathsf{st}(K, \mathbf{x})$ and $\langle K, A\rangle \in \mathcal{E}_{\mathbb{K}_P}$ holds.

Theorem 8 provides us with an algorithm, since brave entailment of existential queries can be decided by verifying the condition (B) of the theorem. As easily seen, the condition is verifiable in time single exponential in the size of the P. Indeed, computing $\mathcal{E}_{\mathbb{K}_P}$ requires time quadratic in the size of $\mathbb{K}_P$, or single exponential in the size of P. Once $\mathbb{K}_P$, $\mathcal{E}_{\mathbb{K}_P}$, and $SM(P^{\mathsf{G}})$ are computed, the conditions in (B) are verifiable in time polynomial in the combined size of $\mathbb{K}_P$, $\mathcal{E}_{\mathbb{K}_P}$, and $SM(P^{\mathsf{G}})$. Hence, (B) can be verified in time single exponential in the size of P, that is, for a given $\mathbb{FDNC}$ program, the problem of deciding whether it bravely entails a unary existential query is in ExpTime. On the other hand, due to Theorem 7 and Lemma 1, we know that brave entailment of unary existential queries is ExpTime-hard already for $\mathbb{FDC}$. Keeping that in mind, we conclude the following.

Theorem 9. *For $\mathbb{FDC}$ and $\mathbb{FDNC}$ programs, brave entailment of an existential unary query is* ExpTime*-complete. The same holds for binary existential queries (see Lemma 1).*

The method for deciding brave entailment of ground queries is based on an adaptation of the algorithm for the existential queries.

Definition 5.6 ($\mathcal{G}_L^q$). Let $q = A(t)$ be a ground atom and L be a set of knots founded w.r.t. an $\mathbb{FDNC}$ program P and a set of states S. Let T be the set of subterms of the term t. Then $\mathcal{G}_L^q$ is the smallest relation over $L \times T$ such that:

(a) if $K \in L$ and $A(\mathbf{x}) \in K$, then $\langle K, t\rangle \in \mathcal{G}_L^q$, and

(b) if there exist (i) $K \in L$ with $f(\mathbf{x}) \in \mathsf{succ}(K)$ and (ii) $K' \in L$ s.t. $\mathsf{st}(K, f(\mathbf{x})) \approx \mathsf{st}(K', \mathbf{x})$ and $\langle K', f(v)\rangle \in \mathcal{G}_L^q$, then $\langle K, v\rangle \in \mathcal{G}_L^q$.

Suppose we have a query $q = A(f(g(f(c))))$ and a knot K in L such that $\langle K, c\rangle \in \mathcal{G}_L^q$. Intuitively, this means there exists a tree construction starting with a root term c that will eventually contain the atom $A(f(g(f(c))))$. Due to the properties of $\mathbb{K}_P$, we have the following.

Theorem 10. *Let P be an $\mathbb{FDNC}$ program and q a ground unary query. Suppose c is the single constant occurring in q. The following two are equivalent:*

(A) $P \models_b q$.

(B) There exists some $G \in SM(P^{\mathsf{G}})$ such that

(i) $\mathbb{K}_P$ is compatible w.r.t. $S(G)$, and

(ii) there exists some $K \in \mathbb{K}_P$ such that $\mathsf{st}(G, c) \approx \mathsf{st}(K, \mathbf{x})$ and $\langle K, c\rangle \in \mathcal{G}_{\mathbb{K}_P}^q$.

By similar arguments as for existential queries, we can see that checking condition (B) is feasible in time single exponential in the size of P and q. Note that computing $\mathcal{G}_{\mathbb{K}_P}^q$ is feasible in time polynomial in the size of $\mathbb{K}_P$ and q, or single exponential in the size of P and q. Once $\mathbb{K}_P$, $\mathcal{G}_{\mathbb{K}_P}^q$, and $SM(P^{\mathsf{G}})$ are computed, the conditions in (B) can be verified in time polynomial in the combined size of $\mathbb{K}_P$, $\mathcal{G}_{\mathbb{K}_P}^q$, and $SM(P^{\mathsf{G}})$, each of which is single exponential in the size of P and q. It follows that for a given $\mathbb{FDNC}$ program, the problem of deciding whether it bravely entails a unary ground query is in ExpTime. Combined with Theorem 7 and Lemma 1, we establish the completeness result.

Theorem 11. *For $\mathbb{FDC}$ and $\mathbb{FDNC}$ programs, brave entailment of a unary ground query is* ExpTime*-complete. The same holds for binary ground queries (see Lemma 1).*

5.4 Cautious Entailment of Open Queries

In the previous sections, we presented methods for brave entailment of existentially quantified or ground queries. As shown in Section 4, cautious reasoning can be easily reduced to consistency testing. All these tasks are ExpTime-complete for $\mathbb{FDNC}$. This section deals with cautious entailment of open queries, which turns out to be harder (under widely adopted beliefs in complexity theory).

Like for other reasoning methods discussed, we base our method on the set $\mathbb{K}_P$ of an $\mathbb{FDNC}$ program P. As we have seen, each stable model of P can be constructed by taking a compatible graph and building a tree for each constant. Indeed, if $P \models_c A(t)$ holds, each tree construction starting at the constant of t from knots in $\mathbb{K}_P$ must eventually reach t satisfying A. To talk about such forcing, we introduce the following notion.

Definition 5.7. (Converging Sequence) Let A be a unary predicate symbol of P. A nonempty sequence $[\frac{L_0}{c}, \frac{L_1}{f_1}, \ldots, \frac{L_n}{f_n}]$, were each $L_i \subseteq \mathbb{K}_P$ is nonempty, c is a constant, and $f_1, \ldots, f_n$ are function symbols, is called a *converging sequence for A (w.r.t. $\mathbb{K}_P$)* if the following hold:

(1) for each $K \in L_{j-1}$, where $1 \leq j \leq n$, $f_j(\mathbf{x}) \in \mathsf{succ}(K)$;

(2) for each $K \in L_{j-1}$, where $1 \leq j \leq n$, and each $K' \in \mathbb{K}_P$, $\mathsf{st}(K, f_j(\mathbf{x})) \approx \mathsf{st}(K', \mathbf{x})$ implies $K' \in L_j$;

(3) if $K \in L_n$, then $A(\mathbf{x}) \in K$.

Furthermore, to talk about knots that can start model construction, we use the following notion. For a constant c, let $\mathsf{seeds}(c, P) := \{K \in \mathbb{K}_P \mid \mathsf{st}(\mathbf{x}, K) \approx \mathsf{st}(c, G) \wedge G \in SM(P^{\mathsf{G}})\}$.

Proposition 9. *Let P be a consistent $\mathbb{FDNC}$ program and let $\lambda x.A(x)$ be an open query. Then $P \models_c \lambda x.A(x)$ iff there exists a converging sequence $s = [\frac{L_0}{c}, \frac{L_1}{f_1}, \ldots, \frac{L_n}{f_n}]$ for A, where $L_0 = \mathsf{seeds}(c, P)$.*

The proposition above characterizes cautious entailment of open queries in terms of existence of converging sequences. To provide an algorithm, we next show that the length of converging sequences can be bounded by a double exponential in the size of the initial program.

Proposition 10. *For every converging sequence $[\frac{L_0}{c}, \frac{L_1}{f_1}, \ldots, \frac{L_n}{f_n}]$ for A there exists a converging sequence $[\frac{L_0}{c}, \frac{L'_1}{f'_1}, \ldots, \frac{L'_m}{f'_m}]$ for A such that $m \leq |F| \times 2^{|\mathbb{K}_P|} + 1$, where F is the set of function symbols occurring in P.*

Proof. There are only $|F| \times 2^{|\mathbb{K}_P|}$ distinct pairs $\frac{L}{f}$ of a function symbol f and a set of knots $L \subseteq \mathbb{K}_P$. Suppose $s = [\frac{L_0}{c}, \frac{L_1}{f_1}, \ldots, \frac{L_n}{f_n}]$ is a converging sequence for A and $\frac{L}{f}$ is an element of s that occurs more than once, first at position $0 < k$ and last at position $k < l$. It is easy to verify that $[\frac{L_0}{c}, \frac{L_1}{f_1}, \ldots, \frac{L_k}{f_k}, \frac{L_{l+1}}{f_{l+1}}, \ldots, \frac{L_n}{f_n}]$ is a converging sequence for A where $\frac{L}{f}$ occurs only once. Note that the first element of the sequence is preserved. It follows that there exists a converging sequence s' for A that does not contain duplicates of its elements, while its first element is $\frac{L_0}{c}$. Indeed, s' cannot be longer than $|F| \times 2^{|\mathbb{K}_P|} + 1$, and thus the claim holds. □

The following theorem follows directly from Proposition 9 and Proposition 10.

```
**Algorithm** *openQueries* ($\mathbb{FDNC}$ program $P$, open query $\lambda x.A(x)$)
**Output:** *true* iff there exists $t$ s.t. $P \models_c A(t)$
if $P$ is inconsistent then
  return *true*
end if
Guess some constant $c$ of $P$;
$L := \mathsf{seeds}(c, P)$;
repeat
  if $A(\mathbf{x}) \in K$ for each $K \in L$ then
    return *true*
  end if
  Guess some $f \in F$;
  if there exists $K \in L$ such that $f(\mathbf{x}) \notin K$ then
    return false
  else
    $L^{aux} := \{K' \in \mathbb{K}_P \mid \mathsf{st}(\mathbf{x}, K') \approx \mathsf{st}(f(\mathbf{x}), K) \wedge K \in L\}$;
    $L := L^{aux}$;
    $i := i + 1$
  end if
until $i = |F| \times 2^{|\mathbb{K}_P|} + 1$
return *false*
```

Figure 4: Non-deterministic procedure for cautious entailment of open queries; $\mathbb{K}_P$ is assumed to be precomputed, F is the set of function symbols of P.

Theorem 12. *Let P be an $\mathbb{FDNC}$ program and $\lambda x.A(x)$ be an open query. Then $P \models_c \lambda x.A(x)$ iff P is inconsistent or there exists a converging sequence $[\frac{L_0}{c}, \frac{L_1}{f_1}, \ldots, \frac{L_n}{f_n}]$ for A, where $L_0 = \mathsf{seeds}(c, P)$ and $n \leq |F| \times 2^{|\mathbb{K}_P|} + 1$.*

Based on this theorem, we present in Figure 5.4 an algorithm that decides cautious entailment of open queries by checking the existence of a converging sequence of at most double exponential length. We assume that the set $\mathbb{K}_P$ for the input program P is precomputed. The procedure non-deterministically guesses a sequence of functions symbols and verifies the conditions in Definition 5.7. Furthermore, the procedure can be implemented to run in non-deterministic exponential space. Indeed, storing the set $\mathbb{K}_P$ and the double exponential counter requires at most exponential space, while the rest of the constructs require at most linear space. By Savitch's result [46], we can turn the algorithm into an ExpSpace-algorithm, which establishes the ExpSpace-membership. By a generic Turing machine encoding, we show that the problem is also ExpSpace-hard, even for $\mathbb{FD}$ and $\mathbb{FN}$ programs (see appendix).

Theorem 13. *Cautious entailment of open queries in $\mathbb{FD}$, $\mathbb{FN}$, $\mathbb{FNC}$, $\mathbb{FDC}$ and $\mathbb{FDNC}$ programs is* ExpSpace*-complete.*

6 Complexity of Fragments

In this section, we consider the complexity of reasoning in the fragments of $\mathbb{FDNC}$. Some reasoning tasks are already covered by the results of the previous section and the discussion in Section 4, including cautious entailment of existential queries in $\mathbb{FD}$ (cf. Theorem 7) and cautious entailment of open queries (cf. Theorem 13).

We first show that in $\mathbb{FN}$, all reasoning tasks remain as hard as in full $\mathbb{FDNC}$. All other reasoning tasks that remain to be considered are at most PSPACE-complete, and in some cases at low levels of the polynomial hierarchy.

6.1 Reasoning in $\mathbb{FN}$ and $\mathbb{FNC}$

We show that the consistency problem for $\mathbb{FDC}$ reduces in polynomial time to the consistency problem for $\mathbb{FN}$. Since the reasoning tasks that we considered (consistency and brave entailment) are EXPTIME-complete for $\mathbb{FDC}$, the reduction implies that they are all EXPTIME-complete for $\mathbb{FN}$ and $\mathbb{FNC}$.

The plan is as follows. We first construct, given an arbitrary $\mathbb{FDC}$ program P, an $\mathbb{FN}$ program $F(P)$ whose set of stable models coincides intuitively with the set of all possible (forest-shaped) Herbrand interpretations for P. We then structurally transform P into an $\mathbb{FN}$ program P' such that the $\mathbb{FN}$ program $P' \cup F(P)$ is consistent iff P is consistent.

We assume that for each unary (resp., binary) predicate symbol Q of P there is a unary (resp., binary) predicate symbol $\bar{Q}$ available which does not occur in P. Moreover, let Dom and A^c be fresh unary predicate symbols, and let S be a fresh binary predicate symbol not occurring in P.

Definition 6.1 ($F(P)$)**.** By $F(P)$ we denote the smallest $\mathbb{FN}$ program that consists of the following rules:

$$\begin{array}{lrl}
\text{(F1)} & Dom(c) & \leftarrow, \\
\text{(F2)} & S(c,d) & \leftarrow, \\
\text{(F3)} & S(x,f(x)) & \leftarrow Dom(x), \\
\text{(F4)} & Dom(y) & \leftarrow S(x,y), \\
\text{(F5)} & A(\vec{x}) & \leftarrow Dom(\vec{x}), not\ \bar{A}(\vec{x}), \\
\text{(F6)} & \bar{A}(\vec{x}) & \leftarrow Dom(\vec{x}), not\ A(\vec{x}), \\
\text{(F7)} & A^c(\vec{x}) & \leftarrow A(\vec{x}), \bar{A}(\vec{x}), not\ A^c(\vec{x}),
\end{array}$$

for each pair c, d of constants of P, each function symbol f of P, each predicate symbol A of P. Above, $\vec{x} = \langle x \rangle$ if A is unary, and $\vec{x} = \langle x, y \rangle$ if A is binary.

The properties of $F(P)$ are stated next.

Proposition 11. *An interpretation I is a stable model of $F(P)$ iff I is a forest-shaped interpretation such that the following hold:*

(1) $S(c,d) \in I$, for each pair c, d of constants of P,

(2) $Dom(t) \in I$, for each term $t \hat{\in} I$,

(3) $S(t, f(t)) \in I$, for each $t \hat{\in} I$ and each function symbol f of P,

(4) $|\{A(t), \bar{A}(t)\} \cap I| = 1$*, for each* $t \hat{\in} I$ *and each unary predicate* A *of* P*, and*

(5) $S(s,t) \in I$ *implies* $|\{R(s,t), \bar{R}(s,t)\} \cap I| = 1$*, for each pair* $s, t \hat{\in} I$ *and each binary predicate* R *of* P*.*

Intuitively, $F(P)$ generates a set of forest-shaped interpretations for P. Next we show how to filter out the interpretations that do not satisfy the rules in P. If some interpretation I remains, then P is consistent. Note that such I would not necessarily correspond to a minimal model of P. For technical reasons, we assume that the rules of type (R6) occurring in P are not disjunctive, i.e., there is at most one literal in the head of the rule. It is easy to see that, in case P has such rules, they can be eliminated in linear time while preserving consistency. Indeed, $R_1(x, f_1(x)) \vee \ldots \vee R_n(x, f_k(x)) \leftarrow B_0(x), \ldots, B_l(x)$ occurring in P can be replaced by $A_1(x) \vee \ldots \vee A_n(x) \leftarrow B_0(x), \ldots, B_l(x)$ and $R_i(x, f_i(x)) \leftarrow A_i(x)$ for each $i \in \{1, \ldots, n\}$. This transformation clearly preserves consistency.

Definition 6.2 ($TR(P)$)**.** For a $\mathbb{FDC}$ program P as described, we denote by $TR(P)$ the $\mathbb{FN}$ program $F(P) \cup P'$, where P' is the $\mathbb{FN}$ program obtained from P by replacing each rule

$$W_1(\vec{t_1}) \vee \ldots \vee W_n(\vec{t_n}) \leftarrow Q_1(\vec{v_1}), \ldots, Q_m(\vec{v_m}) \in P$$

with a rule

$$C(\vec{t_1}) \leftarrow Q_1(\vec{v_1}), \ldots, Q_m(v_m), \bar{W_1}(\vec{t_1}), \ldots, \bar{W_n}(\vec{t_n}), not\ C(\vec{t_1}),$$

where C is a fresh predicate symbol with the arity of $\vec{t_1}$, and $n, m > 0$.

Indeed, $TR(P)$ is an $\mathbb{FN}$ program; literals $W_i(\vec{t_i})$ in the head of an initial rule can be shifted to their "complements" $\bar{W_i}(\vec{t_i})$ in the body without violating the syntax of $\mathbb{FN}$ programs. This would not be the case if disjunctive heads were allowed for the rules of type (R6). The following is easy to see (the proof is given in the appendix).

Proposition 12. *The program* P *is consistent iff* $TR(P)$ *is consistent.*

We showed how to transform an $\mathbb{FDC}$ program into an $\mathbb{FN}$ program while preserving consistency. As easily verified, the translation is polynomial in the size of the initial program P (more precisely, quadratic in the size of P due to the facts (F2) of $F(P)$; the rest is linear). Therefore, recalling ExpTime-completeness of consistency testing in $\mathbb{FDNC}$ (Theorem 7), we conclude.

Theorem 14. *For both* $\mathbb{FN}$ *and* $\mathbb{FNC}$ *programs, checking consistency is* ExpTime*-complete.*

The ExpTime-completeness of consistency checking for $\mathbb{FN}$ allows us to obtain similar results for brave query entailment. Since consistency testing is reducible to brave entailment (see Lemma 1), and since brave entailment of existential and ground queries is ExpTime-complete (see Theorems 9 and 11), we obtain:

Theorem 15. *For* $\mathbb{FN}$ *and* $\mathbb{FNC}$ *programs, brave entailment of a unary ground or existential query is* ExpTime*-complete. The same holds for binary queries (see Lemma 1).*

6.2 Reasoning in $\mathbb{FC}$

We show that reasoning in $\mathbb{FC}$ is easier than in $\mathbb{FDNC}$: consistency and brave reasoning reduce to PSpace-completeness. To obtain these results, we cannot exploit the maximal founded set of knots of a program as its size can be exponentially larger. Nevertheless, the semantic characterization centering around Theorem 1 enables reasoning from $\mathbb{FC}$ programs by iterative construction of knots. The following result, which holds for full $\mathbb{FDNC}$, provides a basis for reasoning in $\mathbb{FC}$.

Theorem 16. *Let P be a $\mathbb{FDNC}$ program. The following two are equivalent:*

(i) There exists a stable model of P.

(ii) There exists a stable model G of P^{G} such that, for each constant c of P, there exists a set of knots that is founded w.r.t. P and the singleton set of states $\{\mathsf{st}(G, c)\}$.

Proof. For the "(i) to (ii)" direction, assume that I is a stable model of P. By Theorem 1, I^c is a stable model of P^{G}. So let $G := I^c$. By Proposition 3, we know that $\mathbb{K}(I)$ is a set of knots that is founded w.r.t. P and $S(G)$. Simply take some set-inclusion minimal set L of knots closed under the following rules:

a) L contains some $K \in \mathbb{K}(I)$ such that $\mathsf{st}(G, c) \approx K, \mathbf{x}$, and

b) if $K \in L$ and $s \in \mathsf{succ}(K)$, then L contains some $K' \in L$ such that $\mathsf{st}(K, s) \approx K', \mathbf{x}$.

Indeed, due to foundedness of $\mathbb{K}(I)$, the set L can be constructed and is founded w.r.t. P and $\{\mathsf{st}(G, c)\}$.

For the other direction, assume (ii) holds. Let L_c denote a set of knots that is founded w.r.t. P and $\{\mathsf{st}(G, c)\}$. Let C be the set of constants of P. Due to Proposition 4, the set $L := \bigcup_{c \in C} L_c$ is a set of knots that is founded w.r.t. P and $S(G)$. Then Theorem 2 proves the claim. □

The key feature of $\mathbb{FC}$ is the unique model property, i.e., if there exists a minimal model for an $\mathbb{FC}$ program, then it is unique. From Theorem 16, we know that to decide whether a $\mathbb{FC}$ program is consistent we can proceed in two steps:

(1) Check the existence of the single minimal model G of P^{G}. If it exists, then proceed to the next step. Otherwise, P is not consistent.

(2) Check whether for each constant c of P, there exists a set of knots that is founded w.r.t. P and $\{\mathsf{st}(G, c)\}$. If the answer is "yes", then P is consistent. Otherwise it is not.

Indeed, G is computable in time polynomial in the size of P. For the second step, notice that the local programs for P also have the unique-model property. This implies the uniqueness of a set L that is founded w.r.t. P and $\{U\}$, where U is a state.

To decide the second step, in Figure 5 we present a generic non-deterministic procedure *check-Condition*. The procedure takes as input an $\mathbb{FDNC}$ program P, a state U, and a Boolean function that maps states to Boolean values. In the procedure, the value *max* is the number of distinct $\mathbf{x}$-grounded knots over the signature of P. As it was already argued, $max = 2^{n+k\cdot(n+m)}$, where n and m are the numbers of unary and binary predicate symbols of P, respectively, and k is the number of function symbols in P.

Let $cond_1$ be a Boolean function that maps each state U to *true* if the program $P(U)$ is inconsistent, and to $false$ otherwise.

Proposition 13. *Let P be an $\mathbb{FC}$ program, and let U be a state. There exists a set of knots that is founded w.r.t. P and $\{U\}$ iff no run of the procedure checkCondition$(P, U, cond_1)$ returns true.*

Proof. The "only if" direction is trivial, while for the other direction, we can simply collect all the knots that appeared at any run of the algorithm. It is easy to verify that such a collection is a set that is founded w.r.t. P and $\{U\}$. □

```
func checkCond (program $P$, state $U$, function $cond$)
repeat
   if $cond(U) = true$ then
      return $true$
   end if;
   Choose $K \in MM(P(U))$ and $s \in \mathsf{succ}(K)$;
   Let $U$ be a state obtained from $\mathsf{st}(K, s)$ by substituting $s$ with $\mathbf{x}$;
   $i := i + 1$
until $i = max$;
return $false$
```

Figure 5: Non-deterministic procedure for PSpace algorithms

The algorithm *checkCondition*$(P, U, cond_1)$ runs in polynomial space. The procedure keeps only a counter that counts up to a single exponential; this requires only polynomial space. Note that the procedure at each iteration works only on a single local program that is of polynomial size. This local program has a unique model property and, hence, representing its models requires polynomial space also.

Indeed, to decide the second step, we need to make only a linear number of calls to *checkCondition*. Summing up, both steps to decide consistency of P are feasible in co-NPSpace w.r.t. to the size of P. By Savitch's Theorem [46], we know co-NPSpace = PSpace.

Theorem 17. *Deciding whether a given* $\mathbb{FC}$ *program is consistent is in* PSpace.

We show PSpace-hardness of the problem by a simulation of PSpace Turing machines.

Definition 6.3. A *deterministic Turing machine (DTM)* is a quadruple (S, Σ, δ, s_0), where S is a finite set of states, Σ is a finite alphabet, δ is a *transition function*, and $s_o \in S$ is the *initial state.* The transition function δ is a partial function

$$\delta : S \times \Sigma \to (S \cup \{accept\}) \times \Sigma \times \{-1, 0, +1\},$$

where *accept* is a new state not occurring in S. We assume a special symbol b in Σ that stands for *blank* symbol. By I_k we denote the kth symbol in the input string $I = I_0, \ldots, I_{|I|-1}$.

Let $\mathcal{L}$ be a language in PSpace, and let T be a DTM which decides whether a given word I is in $\mathcal{L}$ within space $sb(I)$ that is polynomial in $|I|$. The computation of T on I can be simulated by an $\mathbb{F}$ program $P(T, I)$ (see Figure 6.2). Due to construction, we can use a single constraint to decide whether $I \in \mathcal{L}$. It is easy to see that $I \in \mathcal{L}$ iff $P(T, I) \cup \{\leftarrow St_{accept}(x)\}$ is inconsistent.

Keeping in mind that the translation is clearly polynomial in the size of T and I, we have that checking if $I \in \mathcal{L}$ is reducible in polynomial time to consistency checking of an $\mathbb{FC}$ program. Keeping in mind Theorem 17, we conclude the following.

Theorem 18. *For* $\mathbb{FC}$ *programs, checking consistency* PSpace*-complete.*

Since $\mathbb{FC}$ programs have the single-stable model property, the problem of brave entailment of existential queries can be easily expressed by constraints that are allowed in $\mathbb{FC}$.

Generating time:
$Time(st) \leftarrow$ $N(x, f(x)) \leftarrow Time(x)$ $Time(y) \leftarrow N(x, y)$
Initial configuration:
$Sym_{\alpha,\pi}(st) \leftarrow$ for $0 \leq \pi < \lvert I \rvert$ such that $\alpha = I_\pi$ $Sym_{b,\pi}(st) \leftarrow$ for $\lvert I \rvert \leq \pi \leq sb(I)$ $Cur_0(st) \leftarrow$ $St_{s_0}(st) \leftarrow$
Transition $\delta(s, \sigma) = \langle s', \sigma', d\rangle$, where $0 \leq \pi \leq sb(I)$
$Sym_{\sigma',\pi}(y) \leftarrow N(x, y), St_s(x), Sym_{\sigma,\pi}(x), Cur_\pi(x)$ $St_{s'}(y) \leftarrow N(x, y), St_s(x), Sym_{\sigma,\pi}(x), Cur_\pi(x)$ $Cur_{\pi+d}(y) \leftarrow N(x, y), St_s(x), Sym_{\sigma,\pi}(x), Cur_\pi(x)$
Inertia rules, where $0 \leq \pi < \pi' \leq sb(I)$:
$Sym_{\sigma,\pi}(y) \leftarrow N(x, y), Sym_{\sigma,\pi}(x), Cur_{\pi'}(x)$ $Sym_{\sigma,\pi'}(y) \leftarrow N(x, y), Sym_{\sigma,\pi'}(x), Cur_\pi(x)$

Figure 6: Reduction of DTM T with input I to $\mathbb{FC}$ program $P(T, I)$.

Proposition 14. *Let P be an $\mathbb{FC}$ program. Then $P \models_b \exists x.A(x)$ iff P is consistent and $P \cup \{\leftarrow A(x)\}$ is not consistent.*

The proposition implies that brave entailment of an existential unary query in $\mathbb{FC}$ can be polynomially reduced to consistency checking in $\mathbb{FC}$. Keeping in mind that the task is PSpace-hard (Lemma 1), we conclude the following.

Theorem 19. *For $\mathbb{FC}$ programs, brave entailment of unary existential queries is* PSpace*-complete. The same holds for binary existential queries (see Lemma 1).*

In a similar fashion, we prove PSpace-completeness for ground queries. The following proposition is helpful.

Proposition 15. *Let P be a $\mathbb{FC}$ program, and let $A(t)$ be a ground atom with $t = f_n(\ldots f_1(c_0)\ldots)$. Let P' be the program obtained by adding to P the following rules:*

(a) $C_0(c_0) \leftarrow$,

(b) $R(x, f_{i+1}(x)) \leftarrow C_i(x)$, *where* $0 \leq i < n$,

(c) $C_{i+1}(y) \leftarrow C_i(x), R(x, y)$, *where* $0 \leq i < n$, *and*

(d) $D(x) \leftarrow C_n(x), A(x)$,

(e) $\leftarrow D(x)$,

where $C_0, \ldots, C_n$, R and D are fresh predicates not occurring in P. Then $P \models_b A(t)$ iff P is consistent and P' is not consistent.

The proposition implies that brave entailment of a ground unary query in $\mathbb{FC}$ program P can be decided by adding polynomially many rules to P and making two consistency checks, and hence is polynomially reducible to consistency checking in $\mathbb{FC}$. Since the latter is PSPACE-hard by Lemma 1, we have the following result.

Theorem 20. *For $\mathbb{FC}$ programs, brave entailment of unary ground queries is* PSPACE*-complete. The same holds for binary queries (see Proposition 1).*

6.3 Reasoning in $\mathbb{F}$ and $\mathbb{FD}$

$\mathbb{F}$ and $\mathbb{FD}$ programs are Horn programs, and therefore are always consistent. We discuss here brave entailment of existential queries together with brave and cautious entailment of ground queries. PSPACE and EXPTIME completeness of cautious entailment in $\mathbb{F}$ and $\mathbb{FD}$ respectively follows from completeness results for consistency testing in $\mathbb{FC}$ and $\mathbb{FDC}$ (see observation (i) in Section 4).

As known, for a given $\mathbb{F}$ program P, deciding $P \models_b \exists x.A(x)$ can be done PSPACE (see Theorem 19). It is easy to see that the problem is PSPACE-hard. Recall the $\mathbb{F}$ program P^T from Section 6.2 that simulates a Turing machine T on the input I. To check whether T accepts the input, we can pose a brave query asking whether $St_{accept}(t)$ is in the minimal model of P^T for some term t, i.e., $I \in \mathcal{L}$ iff $P^T \models_b \exists.St_{accept}(x)$. As already argued, P^T is of polynomial in size T and I.

Theorem 21. *For $\mathbb{F}$ programs, brave entailment of unary existential queries is* PSPACE*-complete. The same holds for binary existential queries (see Proposition 1).*

For $\mathbb{FD}$ programs, PSPACE-completeness of brave existential queries is not straightforward, since they may have several minimal models and hence the task can not be simply reduced to consistency testing as for $\mathbb{F}$. Using constraints leads to $\mathbb{FDC}$, where consistency testing is already EXPTIME-complete.

The strategy is to use the non-deterministic procedure *checkCondition* from Section 6.2 for consistency testing in $\mathbb{FC}$. To this end, we observe that the semantic characterization of stable models of $\mathbb{FDNC}$ allows us conclude the following.

Theorem 22. *Let P be a $\mathbb{FD}$ program. The following two are equivalent:*

(i) $P \models_b \exists x.A(x)$.

(ii) There exists a minimal model G of P^{G}, a constant c of P, a set of knots L founded w.r.t. P and $\{\mathsf{st}(G, c)\}$ such that L contains some knot K with $A(\mathbf{x}) \in K$.

Proof. If (i) holds, then, due to Theorem 1, we can easily define G and L such that the conditions in (ii) are satisfied. On the other hand, if (ii) is satisfied, then consistency of P and Theorem 1 implies that a minimal model of P such that $A(t) \in P$ for some term t is constructible. □

Let $q = \exists x.A(x)$ be a query and P be an $\mathbb{FD}$ program. The theorem above suggests a method to decide $P \models_b q$. The crucial point is to have a procedure to decide whether for a given state U over P, there exists a set of knots L t founded w.r.t. P and $\{U\}$ containing some knot K such that $A(\mathbf{x}) \in K$.

Let $cond_2$ be a Boolean function that maps each world state U to *true* if $A(\mathbf{x}) \in U$, and to $false$ otherwise.

Proposition 16. *Let U be a state, and P be an $\mathbb{FD}$ program. The following two are equivalent.*

(i) There exists a set of knots L founded w.r.t. P and $\{U\}$, ant $A(\mathbf{x}) \in K$, for some $K \in L$.

(ii) There exists a run of the procedure $checkCondition(P, U, cond_2)$ that returns true.

Proof. (i) $\Rightarrow$ (ii): this holds since the size of L is bounded by max.

(ii) $\Rightarrow$ (i): consider the sequence of knots that was constructed during the run of the procedure that returned *true*. Since P has no constraints, this sequence can be always augmented to a founded set by computing the successors knots that are missing. $\square$

Similarly as it was argued for consistency check in $\mathbb{FC}$, *checkCondition* runs in PSPACE in the size of the input. Note that traversing the world states of constants occurring in minimal models of P^{G} can obviously be done in PSPACE. Therefore, we conclude that the condition (ii) in Theorem 22 can be decided in PSPACE with a PSPACE oracle, which amounts to PSPACE. Keeping in mind that deciding $P \models_b q$ is PSPACE-hard (see Lemma 1), we conclude:

Theorem 23. *For $\mathbb{FD}$ programs, brave entailment of a unary existential query is* PSPACE*-complete. The same holds for binary existential queries (see Lemma 1).*

In contrast to existential queries, brave and cautious reasoning with ground queries is easier in $\mathbb{F}$ and $\mathbb{FD}$ than in $\mathbb{FC}$ and $\mathbb{FN}$. The methods are based on constructing only relevant parts of stable models to answer a given query. Since $\mathbb{F}$ and $\mathbb{FD}$ do not allow for constraints, we do not need to care about the global consistency of interpretations. By the relevant part of a model, we essentially mean a sequence of knots that is constructed following the path encoded in the term t of a ground query $A(t)$. The following proposition elaborates on that.

Proposition 17. *Let P be a $\mathbb{FD}$ program, $A(t)$ a ground atom, and c the only constant occurring in t. Moreover, let $l = \langle s_1, \ldots, s_n \rangle$ be the list of subterms of t ordered by increasing depth of terms, i.e., $s_1 = c$ and $s_n = t$. Then the following two are equivalent.*

1. *$P \models_b A(t)$ if and only if ($\star$) there exists some stable model G of P^{G} and a sequence $\langle K_1, \ldots, K_n \rangle$ of stable knots with $\mathsf{root}(K_i) = s_i$, $1 \leq i \leq n$, such that:*

   (a) $\mathsf{st}(G, s_1) = \mathsf{st}(K_1, s_1)$,

   (b) $s_{i+1} \in \mathsf{succ}(K_i)$ and $\mathsf{st}(K_i, s_{i+1}) = \mathsf{st}(K_{i+1}, s_{i+1})$, where $1 \leq i \leq n$, and

   (c) $A(s_n) \in K_n$.

2. *$P \not\models_c A(t)$ if and only if ($\star\star$) there exists some model G of P^{G} and a sequence $\langle K_1, \ldots, K_n \rangle$ of knots with $\mathsf{root}(K_i) = s_i$, $1 \leq i \leq n$, such that:*

   (a) $\mathsf{st}(G, s_1) = \mathsf{st}(K_1, s_1)$,

   (b) $s_{i+1} \in \mathsf{succ}(K_i)$ and $\mathsf{st}(K_i, s_{i+1}) = \mathsf{st}(K_{i+1}, s_{i+1})$, where $1 \leq i \leq n$,

   (c) K_i is a model of $P(\mathsf{st}(K_i, s_i))$, where $1 \leq i \leq n$, and

   (d) $A(s_n) \notin K_n$.

Proof. For the only-if direction of the first claim, assume we have a stable model I of P such that $A(t) \in I$. Due to Theorem 1, we can simply define $G := I^c$ and $K_i := \mathcal{HB}_{s_i} \cap I$, where $1 \leq i \leq n$.

For the if direction, since $\mathbb{FD}$ programs are always consistent, due to Theorem 1, we can easily construct a stable model containing $A(s_n)$; simply start with $G \cup K_1 \cup \ldots, \cup K_n$ and extend the interpretation with the necessary stable knots. Due to consistency of P, such an extension is always possible.

For the "only if" direction of the second claim, the arguments is as for the first one. If I is a stable model of P such that $A(t) \notin I$, then, due to Theorem 1, we can easily define necessary G and the sequence of knots. Again, take $G := I^c$ and : $K_i = \mathcal{HB}_{s_i} \cap I$, where $1 \leq i \leq n$.

For the other direction, let I be the unique stable model of P. Let $K_i' := \mathcal{HB}_{s_i} \cap I$, where $1 \leq i \leq n$. Due to Theorem 1, we have $I^c \subseteq G$ and $K_i' \subseteq K_i$, where $1 \leq i \leq n$. Hence, $A(s_n) \notin I$. □

Proposition 17 allows use to derive complexity results for $\mathbb{F}$ and $\mathbb{FD}$.

Suppose P is an $\mathbb{F}$ program and $A(t)$ a ground query. Since P is a Horn program, the local programs for P have least models computable in polynomial time. Moreover, the least model of P^{G} is also computable in polynomial time. Hence, $P \models_b A(t)$ can be decided according to Proposition 17 by constructing in polynomial time the least model of P^{G} and the unique sequence of knots. Hence, $P \models_b A(t)$ is in P. On the other hand, since P has the least model, $P \models_b A(t)$ iff $P \models_c A(t)$. Hence, $P \models_c A(t)$ is also in P.

Now suppose that P is an $\mathbb{FD}$ program and $A(t)$ a ground query. It is easy to see that the condition $(\star)$ for the program P can be verified in Σ_2^P. Indeed, guess an interpretation I for P^{G} and a suitable candidate sequence of knots over the signature of P; this results in a structure of polynomial size. For any such guess, one can check in polynomial time with an NP oracle whether I is minimal and each of the knots satisfies the conditions in $(\star)$.

To decide $P \not\models_c A(t)$, it suffices to verify the condition $(\star\star)$ in Proposition 17. Since the condition does not require minimality of models, it can be decided in NP. Indeed, we need to guess an interpretation for P^{G} and a candidate sequence of knots over the signature of P. After the guess is made it is possible to decide in polynomial time if the structure satisfies $(\star\star)$. Hence, $P \not\models_c A(t)$ is in NP, while $P \models_c A(t)$ is in CO-NP.

It is not difficult to see that the given upper bounds are tight, since they correspond to complexity of brave and cautious reasoning in the propositional case. Simply consider fragments $\mathbb{F}_p$ of $\mathbb{F}$ and $\mathbb{FD}_p$ of $\mathbb{FD}$ that allow only for rules of type (R1), unary facts, and only one constant. Indeed, any propositional Horn (resp. propositional positive disjunctive program) can be rewritten in LOGSPACE into an $\mathbb{F}_p$ (resp. $\mathbb{FD}_p$) program while preserving the set of minimal models (up to renaming of atoms). This implies that brave/cautious reasoning in propositional Horn and positive disjunctive logics programs are LOGSPACE-reducible to brave/cautious entailment of ground unary queries in $\mathbb{F}$ and $\mathbb{FD}$ programs respectively. Since brave entailment for propositional disjunctive programs is Σ_2^P-complete and cautious entailment CO-NP-complete, while both tasks are P-complete for Horn programs [18], we obtain completeness results for our formalisms.

Theorem 24. *For $\mathbb{FD}$ programs, brave and cautious entailment of unary ground queries is complete Σ_2^P and* CO-NP*, respectively. Both problems are* P*-complete for $\mathbb{F}$ programs. These results extend to binary queries (see Lemma 1).*

7 Applications and Extensions

In Section 5, we have already encountered an application of $\mathbb{FDNC}$ programs to Description Logics. In this section, we consider first a further application of $\mathbb{FDNC}$ programs in the area of reasoning about actions and planning; recall that non-monotonic logic programs under answer set semantics have been widely used in this area. In particular, we apply $\mathbb{FDNC}$ programs to planning under incomplete knowledge and non-deterministic action effects, based on the expressive action language $\mathcal{K}$ [17]. We then consider a decidable extension of $\mathbb{FDNC}$ that supports predicate and function symbols of higher arities, which allows for more succinct and convenient knowledge representation in practice, which we discuss also on a planning scenario.

7.1 Reasoning about Actions and Planning

Transition-based action formalisms are based on languages for describing legal transitions between states of the world which happen due to the execution of actions by some agent. A classical problem is that of *plan existence*, which consists of finding a sequence of actions that leads the agent from an initial to some desired goal state of the world. Apart from this, many problems have been considered, including *plan verification* (i.e., whether a given candidate plan is good to reach a goal state) and *temporal projection* (i.e., reasoning about the hypothetical future if a sequence of action would be taken); as for the concerns of this paper, we refer to [6] for background and a study of these problems based on logic programs under answer set semantics.

As for temporal projection in Example 2, view *grow*, cell_1, cell_2, and *die* as actions and *Young*, *Warm*, *Cold*, and *Mature* as fluents. As seen, if the sequence of actions *grow* and cell_1 would happen, the fluent *Young* would be possibly true, as $\mathit{Young}(\mathit{cell}_1(\mathit{grow}(b)))$ is bravely entailed by the program. On the other hand, *Young* is not necessarily true after this action sequence. Indeed, using similarly as in Proposition 15 an auxiliary fact $C_0(b)$. and rules $R(x, \mathit{grow}(x)) \leftarrow C_0(x)$; $C_1(y) \leftarrow C_0(x), R(x, y)$; $R(x, \mathit{cell}_1(x)) \leftarrow C_1(x)$; and $\leftarrow R(x, y), \mathit{not}\ \mathit{Change}(x, y)$, we can eliminate those stable models of P^{ex} which do not correspond to the occurrence of this sequence; the resulting program P' does not cautiously entail $\mathit{Young}(\mathit{cell}_1(\mathit{grow}(b)))$, as it has a stable model which does not contain this atom. In this scenario, planning seems not to make sense (as bacteria can't really take actions), and we thus consider a different one.

For modeling planning domains, several dedicated action languages have been proposed that are rooted in knowledge representation formalisms, including $\mathcal{A}$ [24] (which was extensively studied in [6]), $\mathcal{C}$ [25], and $\mathcal{K}$ [17]. The latter, which we consider in the sequel, is based on the principles of logic programming under the stable model semantics. In contrast to the other languages, $\mathcal{K}$ allows to describe transitions between knowledge states, which are incompletely described states of the world. The availability of non-monotonic negation in $\mathcal{K}$ makes the formalism suitable for common-sense and heuristic reasoning in planning applications.

In $\mathcal{K}$, a *planning domain PD* is a set of rules that describes the initial state I and legal transitions. At the core, it distinguishes two kinds predicates: *fluents* and *actions*.[4] A *state* is

[4]We consider here merely a simplified version of $\mathcal{K}$ that contains the salient elements; missing features like static predicates, typing and others can be added easily on top. Furthermore, we assume that actions are not executed in parallel (parallel execution may be encoded using designated action symbols), that at each stage some action has to be taken to move on (thus passage of time would have to be modeled explicitly by an action), and that taking an executable action always results in a follow up state. Technically, such planning domains are *proper* and more general than *plain* ones in the sense of [17].

given by a set of ground fluent literals which are known to hold at a particular stage. A *goal G* is a set of ground fluent literals, each of which can also be default negated.

An *optimistic (aka credulous) plan* for a given planning domain PD and a goal G is a sequence of action occurrences $\langle A_1, \ldots, A_n \rangle$, $n \geq 0$, that legally transforms the initial state I into some state that satisfies the goal G, i.e., for some sequence of state $S_0, \ldots, S_n$, we have (i) $S_0 = I$, (ii) each S_i, A_{i+1}, S_{i+1} is a legal transition, and (iii) S_n satisfies G.

In case of non-deterministic action effects or incomplete information about the initial state, executing an optimistic plan does not necessarily establish the goal. This is ensured by *secure plans, aka as conformant plans*, which are optimistic plans such that, regardless of such incompleteness and non-deterministic action effects, all actions can be executed and the goal is established after the last action.

The legal state transitions are defined in $\mathcal{K}$ in terms of stable model semantics. Roughly speaking, this is accomplished using a set of statements, similar to logic program rules, which describe the value of the fluents in the successor state S' depending on the previous state S, the action A that was taken, and the the value of other fluents in S'. Because of this similarity, planning problems in $\mathcal{K}$ can be naturally encoded into $\mathbb{FDNC}$ programs. Via such encodings, optimistic and secure plan existence can be characterized in terms of brave entailment of existential queries and cautious entailment of open queries, respectively.

More in detail, we consider here the propositional fragment of $\mathcal{K}$, i.e., predicates are nullary (predicates of higher arity will be addressed in the next subsection). A planning domain PD in $\mathcal{K}$ consists of *causation rules*, *executability conditions*, and *initial state constraints.* The causation rules of propositional $\mathcal{K}$ are of the form

$$\begin{array}{ll} \texttt{caused}\ d & \texttt{if}\ b_1, \ldots, b_n, \mathit{not}\ b_{n+1}, \ldots, \mathit{not}\ b_k \\ & \texttt{after}\ c_1, \ldots, c_m, \mathit{not}\ c_{m+1}, \ldots, \mathit{not}\ c_l \\ & \qquad a_1, \ldots, a_v, \mathit{not}\ a_{v+1}, \ldots, \mathit{not}\ a_w \end{array} \tag{2}$$

$k, l, w \geq 0$, where d and $b_1, \ldots, b_k, c_1, \ldots, c_l$ are fluent literals, and $a_1, \ldots, a_w$ are action atoms. Intuitively, the rule (2) describes the (incomplete) knowledge state after action execution, where the knowledge depends on the fluents that hold or do not hold in the previous and current state and the actions that were or were not executed.

The *executability conditions* in $\mathcal{K}$ are of the form

$$\begin{array}{ll} \texttt{executable}\ a\ \texttt{if} & b_1, \ldots, b_n, \mathit{not}\ b_{n+1}, \ldots, \mathit{not}\ b_k, \\ & a_1, \ldots, a_m, \mathit{not}\ a_{m+1}, \ldots, \mathit{not}\ a_l, \end{array} \tag{3}$$

where $a, a_1, \ldots, a_l$ are action atoms, and $b_1, \ldots, b_k$ are fluent literals, $k, l \geq 0$, Intuitively, they are the rules constraining the states for which a given action can be executed.

The initial state constraints in $\mathcal{K}$ are of the form

$$\texttt{initially caused}\ d \quad \texttt{if}\ b_1, \ldots, b_n, \mathit{not}\ b_{n+1}, \ldots, \mathit{not}\ b_k \tag{4}$$

where $d, b_1, \ldots, b_k$ are fluent literals, $k \geq 0$. These rules describe the initial knowledge. Unconditional initial knowledge is described by the rules with an empty `if` part.

We next sketch the elements of a possible encoding of the planning domain PD into an $\mathbb{FDNC}$ program. For this purpose, we tacitly enhance $\mathbb{FDNC}$ programs with "strong" negation $\neg p(\vec{x})$ [23], which is expressed in the core language as usual (view $\neg p$ as a fresh predicate symbol and add constraints $\leftarrow p(\vec{x}), \neg p(\vec{x})$); we assume this enhancement also for the predicate-version of $\mathcal{K}$ in Section 7.2.

- For each propositional fluent symbol d, we use a unary predicate symbol d in the encoding. The meaning of $d(x)$ is that d holds at stage x. For each propositional action a, we use a binary predicate symbol a in the encoding. Intuitively, $a(x, y)$ means that a is executed in stage x with the resulting stage y.
- We use a unary predicate symbol s, with $s(x)$ meaning that x is a stage (or a situation). For the encoding we add the fact $s(init) \leftarrow$ denoting that the constant $init$ is the initial stage. We also use a designated binary predicate symbol tr to denote the transition to the next stage. For this reason, we also add $s(y) \leftarrow tr(x, y)$.
- We adopt a function symbol f_a for each action a of the planning domain. Additionally, for each action a, we add the rule $a(x, f_a(x)) \leftarrow exec_a(x)$ and the rule $tr(x, y) \leftarrow a(x, y)$, where $exec_a$ is a designated predicate name. Intuitively, the first rule "implements" the action execution, i.e., if $exec_a$ holds at some stage x, then a is executed, which results in the follow up stage $f_a(x)$. The second rule makes tr capture all executed transitions.

We can now state the encoding of the three types of rules of the planning domain PD.

- The causation rule (2) is transformed into the following rule:

$$\begin{aligned} d(y) \leftarrow \quad & b_1(y), \ldots, b_n(y), not\ b_{n+1}(y), \ldots, not\ b_k(y), \\ & c_1(x), \ldots, c_m(x), not\ c_{m+1}(x), \ldots, not\ c_l(x), \\ & a_1(x, y), \ldots, a_v(x, y), not\ a_{v+1}(x, y), \ldots, not\ a_w(x, y), tr(x, y) \end{aligned}$$

- The executability condition (3) is transformed into the following rule:

$$\begin{aligned} exec_a(y) \leftarrow \quad & s(y), b_1(y), \ldots, b_n(y), not\ b_{n+1}(y), \ldots, not\ b_k(y), \\ & a_1(x, y), \ldots, a_v(x, y), not\ a_{v+1}(x, y), \ldots, not\ a_w(x, y). \end{aligned}$$

  Here, we assume for simplicity as in [16] that there are no positive cyclic interdependencies between actions.

- The initial state constraint (4) is transformed into the following rule:

$$d(init) \leftarrow \quad b_1(init), \ldots, b_n(init), not\ b_{n+1}(init), \ldots, not\ b_k(init),$$

The translation above allows to reformulate planning problems in PD as reasoning tasks for $\mathbb{FDNC}$ programs. A goal G in PD is an expression of the form

$$g_1, \ldots, g_n, not\ g_{n+1}, \ldots, not\ g_k \tag{5}$$

where each g_i is a fluent literal. For this, we add to the translation the following rule:

$$plan(x) \leftarrow g_1(x), \ldots, g_n(x), not\ g_{n+1}(x), \ldots, not\ g_k(x) \tag{6}$$

where $plan$ is a new predicate symbol. Let $P(PD, G)$ denote the resulting program.

To know whether an optimistic plan for G in PD exists, we can pose the brave query $\exists x.plan(x)$ to the program $P(PD, G)$. Similarly, the cautious open query $\lambda x.plan(x)$ can be posed for a secure plan. Due to the stable model semantics of both languages, it is not hard (yet technical) to show that a stable model of $P(PD, G)$ encodes a set of possible trajectories $S_0, A_1, S_1, A_2, \ldots$ in PD where S_0 is any initial knowledge state; the whole set $SM(P(PD, G))$ captures all the trajectories for PD.

Further, each term t such that $P(PD, G) \models_b plan(t)$ naturally encodes an optimistic plan for the problem, and each term t that is an answer for $\lambda x.plan(x)$ under cautious entailment encodes a secure plan. Thus, plan correctness and security verification problems can be readily solved by the standard inference tasks $P(PD, G) \models_b plan(t)$ and $P(PD, G) \models_c plan(t)$.

We note at this point that deciding the existence of some secure plan (of arbitrary length) to establish a given goal G in a given $\mathcal{K}$ action domain that conforms to the setting considered here is ExpSpace-complete (this is well-known for a generic related action formalism [27]; the hardness part can be shown by slightly adapting the NExpTime-hardness proof for the problem when a prescribed plan length is part of the input [17]).

Finally, also temporal projection with respect to an action sequence $\vec{a} = a_1, a_2, \ldots, a_k$, $k \geq 1$ can be easily expressed: whether a fluent d is possibly true after hypothetically taking $\vec{a}$ is expressed by the entailment $P(PD) \models_b d(t)$ where $t = f_{a_k}(f_{a_{k-1}} \cdots (f_{a_1}(init)))$ where $P(PD)$ is $P(PD, G)$ except the rules (5) and (6). Whether d is necessarily true when $\vec{a}$ would have happened can be expressed, using again a similar technique as in Proposition 15, as cautious entailment of $d(t)$ from $P(D)$ augmented with the auxiliary fact $C_0(init)$. and rules $R(x, f_{a_{i+1}}(x)) \leftarrow C_i(x)$, for $0 \leq i < k$, $C_{i+1}(y) \leftarrow C_i(x), R(x, y)$, for $0 \leq i < k - 1$, and $\leftarrow R(x, y), not\ tr(x, y)$, where all C_i and R are fresh predicates (this singles out the models in which $\vec{a}$ would be taken).

Further tasks like reasoning about the initial state or observation assimilation [6] can be similarly expressed.

Example 4. Table 3 presents an example encoding of a propositional planning domain in $\mathcal{K}$ into an $\mathbb{FDN}$ program P, which is an adaptation of the classical Yale-Shooting example [26]. Here we assume three fluents *See*, *Loaded*, *Hit*, and two actions *load* and *shoot*. In the initial situation, a hunter sees a target, but his gun is not loaded (row (1)). The fluents *See* and *Loaded* are inertial, i.e., their truth values do not change unless proved otherwise (rows (2) and (3)). The hunter can load the gun only if it is unloaded, and can shoot only if the gun is loaded (rows (4) and (5)). The gun becomes loaded after loading occurs (row(6)). Finally, the hunter hits the target, if he shoots while seeing the target (row (7)). The goal in the planning domain is *Hit*, and hence the rule $plan(x) \leftarrow Hit(x)$ is added to the encoding.

It is easy to see that $P \models_b \exists x.plan(x)$, i.e., there exists a plan where the hunter hits the target and is witnessed by the term $t = shoot(load(init))$. The inertia of *See* is crucial; dropping the statement in row (3) wouldn't let us assume that the hunter still sees the target after loading the gun.

The term t also encodes a secure plan for the domain, i.e., t witnesses the open query $\lambda x.plan(x)$. This becomes false when instead of sure knowledge that the gun is not loaded in the initial state, the status of the gun in the initial stage can vary freely. This situation is modeled by the two rules `caused` *Loaded* `if` *not* $\neg$*Loaded* and `caused` $\neg$*Loaded* `if` *not Loaded*. In this case, t is still an optimistic plan for the domain, but is not secure (as the first step might not be executable). On the other hand, if hypothetically t would happen, then *Hit* would be both possibly and necessarily true after it.

(1)	`initially caused` $See, \neg Loaded$	$\rightsquigarrow$	$See(init) \leftarrow;\ \neg Loaded(init) \leftarrow;$
(2)	`caused` $Loaded$ `if` $not\ \neg Loaded$ `after` $Loaded$	$\rightsquigarrow$	$Loaded(y) \leftarrow Loaded(x), tr(x,y), not\ \neg Loaded(y)$
(3)	`caused` See `if` $not\ \neg See$ `after` See	$\rightsquigarrow$	$See(y) \leftarrow See(x), tr(x,y), not\ \neg See(y)$
(4)	`executable` $load$ `if` $\neg Loaded$	$\rightsquigarrow$	$exec_{load}(x) \leftarrow \neg Loaded(x)$
(5)	`executable` $shoot$ `if` $Loaded$	$\rightsquigarrow$	$exec_{shoot}(x) \leftarrow Loaded(x)$
(6)	`caused` $Loaded$ `after` $load$	$\rightsquigarrow$	$Loaded(y) \leftarrow load(x,y), tr(x,y)$
(7)	`caused` Hit `after` $See, shoot$	$\rightsquigarrow$	$Hit(y) \leftarrow See(x,y), shoot(x), tr(x,y)$

Table 3: Example of Planning Domain Encoding

To provide a procedure for deciding plan existence in the planning domains of $\mathcal{K}$, the authors of [17] encode the domain into a disjunctive Datalog program and reformulate plan existence in terms of brave entailment. Since Datalog does not allow for function symbols, the encoding uses constants to instantiate the necessary successor stages. Obviously, only a finite number of constants can be used and hence, it has to be fixed in advance. For this reason the encoding is not general; only plans of certain length can be captured. Furthermore, such an encoding may also incur high space requirements.

The encoding into $\mathbb{FDNC}$ solves the problems of above. The availability of function symbols allows to easily generate an infinite time-line, and, hence, to avoid the usage of constants. Due to the properties of $\mathbb{FDNC}$, the encoding also allows to generate the successors states “on-demand” during the model construction; in this way, space might be saved.

7.2 Higher-arity $\mathbb{FDNC}$

In the previous section, we discussed how $\mathbb{FDNC}$ can be used to model propositional planning domains. However, a propositional setting is not always convenient to model complex planning problems. Parameterized actions and fluents are a means for more compact representation. It allows us to work with actions that, for instance, move an object x from a location l_1 to the location l_2. In this way, we avoid to introduce separate actions for each possible combination of x, l_1, and l_2 as needed in the propositional setting. (This is, e.g., widely used in [6].)

In this section, we extend the class $\mathbb{FDNC}$ of logic programs to allow for predicate and function symbols of higher arities. We assume two disjoint sets U and B of predicate names having arities at least 1 and 2, respectively. Given an atom $A(t_1,\ldots,t_n)$ with $A \in U$ or a term $f(t_1,\ldots,t_n)$, its *local positions* are $0,\ldots,n-1$ and its *global position* is n. Similarly, given an atom $A(t_1,\ldots,t_n)$ with $A \in B$, its *local positions* are $0,\ldots,n-2$ and its *global positions* are $n-1$ and n. An atom $A(\vec{t})$ with $A \in U$ (resp., $A \in B$) is *g-unary* (resp., *g-binary*).

Definition 7.1. A *higher arity* $\mathbb{FDNC}$ program is a finite disjunctive logic program whose rules are of the following forms:

$$\text{(R1)} \qquad \textstyle\bigvee_{i=1}^{n} A_i(\vec{t_i}, X) \leftarrow \bigwedge_{j=0}^{l} B_j^{\pm}(\vec{u_j}, X)$$

$$\text{(R2)} \qquad \textstyle\bigvee_{i=1}^{n} R_i(\vec{v_i}, X, Y) \leftarrow \bigwedge_{j=0}^{l} P_j^{\pm}(\vec{u_j}, X, Y)$$

$$\text{(R3)} \qquad \textstyle\bigvee_{i=1}^{n} R_i(\vec{v_i}, X, f_i(\vec{t_i}, X)) \leftarrow \bigwedge_{j=0}^{l} P_j^{\pm}(\vec{w_j}, X, g_j(\vec{u_j}, X))$$

$$
\begin{array}{lrcl}
\text{(R4)} & \bigvee_{i=1}^{n} A_i(\vec{t_i}, Y) & \leftarrow & R(\vec{w}, X, Y), \bigwedge_{j=0}^{l} B_j^{\pm}(\vec{u_j}, Z_j) \\
\text{(R5)} & \bigvee_{i=1}^{n} A_i(\vec{t_i}, f(\vec{v}, X)) & \leftarrow & R(\vec{w}, X, f(\vec{v}, X)), \bigwedge_{j=0}^{l} B_j^{\pm}(\vec{u_j}, W_j) \\
\text{(R6)} & \bigvee_{i=1}^{n} R_i(\vec{v_i}, X, f_i(\vec{t_i}, X)) & \leftarrow & \bigwedge_{j=0}^{l} B_j^{\pm}(\vec{u_j}, X) \\
\text{(R7)} & \bigvee_{i=1}^{n} A_i(\vec{t_i}, b) \vee \bigvee_{i=1}^{m} R_i(\vec{v_i}, c, d) & \leftarrow & \bigwedge_{i=0}^{k} B_i^{\pm}(\vec{t_i}, b'), \bigwedge_{i=0}^{l} P_i^{\pm}(\vec{t_i}, c', d'),
\end{array}
$$

where $m, n, l, k \geq 0$, and

- each $Z_i \in \{X, Y\}$, $W_i \in \{X, f(\vec{v}, X)\}$,
- each A_i and B_j is from U, and each R_i and P_j is from B,
- the tuples $\vec{v}$, $\vec{w}$, and all $\vec{v_i}, \vec{t_i}$, $\vec{u_j}$ are tuples of variables or constants.
- b, c, d, b', c', d' are constants,
- C_i and D_j is a name from $U \cup B$,
- X and Y do not occur in local positions of atoms and function symbols, and
- each rule r is *safe*, i.e., each of its variables occurs in $\mathsf{body}^{+}(r)$.

We additionally assume that each constant c of the program, does not occur both in the local position of an atom and in the global position of another atom.

The restrictions on the variable interaction allow us to transform higher-arity $\mathbb{FDNC}$ programs naturally into ordinary $\mathbb{FDNC}$ programs in a way such that the methods that were introduced for reasoning in the previous sections can be used. In the following, we present the transformation and a use case of a higher-arity $\mathbb{FDNC}$ program.

Definition 7.2. Let P be a higher-arity $\mathbb{FDNC}$ program. Let $ld(P)$ denote the set of constants occurring in the local positions of atoms in P. For a rule r, let $lv(r)$ denote the set of variables occurring in local positions of atoms in r. We say r' is a *parameter-ground* instance of r w.r.t. a set of constants S, if r' can be obtained by substituting each variable in $lv(r)$ with a constant in S. Let $gr(r, S)$ denote the set of all locally-ground instances of r w.r.t. S. The *parameter-grounding* of P is the program $pgr(P) = \{r' \in gr(r, ld(P)) \mid r \in P\}$. The $\mathbb{FDNC}$*-reduction of* P is the $\mathbb{FDNC}$ program $red(P)$ obtained from P by replacing each g-unary atom $A(t_1, \ldots, t_n)$ (resp., g-binary atom $R(t_1, \ldots, t_n)$) occurring in the rules of $pgr(P)$ with an atom $A_{t_1,\ldots,t_{n-1}}(t_n)$ (resp., $R_{t_1,\ldots,t_{n-2}}(t_{n-1}, t_n)$). Similarly, the $\mathbb{FDNC}$*-reduction of an interpretation* I *for* P is defined as $red(I) := \{A_{t_1,\ldots,t_{n-1}}(t_n) \mid A(t_1, \ldots, t_n) \in I,\ A \in U\} \cup \{R_{t_1,\ldots,t_{n-2}}(t_{n-1}, t_n) \mid R(t_1, \ldots, t_n) \in I,\ R \in B\}$.

The following result is then not difficult to establish.

Theorem 25. *For any higher-arity* $\mathbb{FDNC}$ *program* P,

$$SM(P) = \{I \mid red(I) \in SM(red(pgr(P)))\}.$$

Since $red(P)$ is finite, higher-arity $\mathbb{FDNC}$ programs inherit decidability from ordinary $\mathbb{FDNC}$ programs. However, their complexity is higher (by one exponential) in the general case. This is not surprising, since the parameter-grounding of P is exponential in the size of P.[5]

An exponential blow-up only occurs when arbitrarily many parameters are allowed in rules, i.e., if the number of variables that can occur in local position is unbounded. If the maximal number of variables in local positions is fixed, then the parameter-grounding is polynomial in the size of a higher-arity program, and our complexity results carry over for higher-arity $\mathbb{FDNC}$.

Below is an example of an application of higher-arity $\mathbb{FDNC}$ programs to compactly represent the *blocks world* problem (the example is an adaptation of the one in [17]).

Example 5. We assume that initially we have 3 blocks a, b, and c. In the initial state, a and b are on the table (*table*), while c is on top of a. This is formalized by the following facts:

$$\begin{array}{rr}
block(a,0)\leftarrow & on(a,table,0)\leftarrow \\
block(b,0)\leftarrow & on(b,table,0)\leftarrow \\
block(c,0)\leftarrow & on(c,a,0)\leftarrow \\
loc(table,0)\leftarrow &
\end{array}$$

We need to state the static knowledge about the objects, i.e., the properties of objects that do not change during the execution of actions. We thus state that blocks remain blocks, locations remain locations, and that occupation is determined by having a block on top:

$$\begin{array}{rl}
block(B,y) & \leftarrow block(B,x), change(x,y) \\
loc(L,y) & \leftarrow loc(L,x), change(x,y) \\
loc(B,x) & \leftarrow block(B,x) \\
occupied(B,x) & \leftarrow on(B1,B,x), block(B,x)
\end{array}$$

Next are the effects of action execution. We need to mark the locations that become occupied/unoccupied after moving a block from one location to another. On the other hand, we need to state that the rest of the configuration does not change:

$$\begin{array}{rl}
on(B,L,y) & \leftarrow block(B,x), loc(L,x), change(x, move(B,L,x)) \\
\neg on(B,L_1,move(B,L,x)) & \leftarrow block(B,x), loc(L,x), change(x, move(B,L,x)), on(B,L1,x), neq(L,L_1) \\
on(B,L,move(B,L,x)) & \leftarrow on(B,L,x), change(x, move(B,L,x)), not\ \neg on(B,L,move(B,L,x))
\end{array}$$

We use an inequality $neq(x,y)$ predicate over parameters, which is axiomatized by adding for each pair $c_1, c_2 \in ld(P)$ such that $c_1 \neq c_2$, the fact $neq(c_1,c_2) \leftarrow$ to the program.

Next is the executability of an action; only blocks can be moved, and they can only be placed in some location.

$$change(x, move(B,L,x)) \vee \neg change(x, move(B,L,x)) \leftarrow block(B,x), loc(L,x)$$

The disjunctive rule allows to freely execute the action. Since there might be several blocks that can be moved, the last rule does not force the execution of all applicable action simultaneously.

[5]The hardness results for ExpTime, co-NExpTime, and co-NExpTime$^{\mathrm{NP}}$ corresponding to P, co-NP, and Σ_2^P, respectively, follow from the complexity of ordinary function-free logic programs [12]; ExpSpace and 2-ExpSpace corresponding to PSpace and ExpSpace can be obtained by generalizing the given Turing machine encodings. As for 2-ExpTime corresponding to ExpTime, one can show ExpTime-hardness of reasoning in ordinary $\mathbb{FDC}$ and $\mathbb{FN}$ by adapting the given encoding of PSpace Turing machines into $\mathbb{FC}$ to *alternating* PSpace-Turing machines (which capture ExpTime). With parameters, this can be further lifted to alternating ExpSpace = 2-ExpTime.

The execution of an action can be prohibited by the constraints. In our setting, the block cannot be moved if either the destination is occupied or the block has a block on top of it:

$$\neg change(x, move(B, L, x)) \leftarrow occupied(B, x)$$
$$\neg change(x, move(B, L, x)) \leftarrow occupied(L, x)$$

We ask the question whether there exists a sequence of actions that transforms the initial configuration into the one where a is on the table, b is on a and c is on b. This is expressed by the following rule:

$$plan(x) \leftarrow on(c, b, x), on(b, a, x), on(a, table, x)$$

The existence of a plan for the encoded problem can now be decided by the brave query $\exists x.plan(x)$ to the higher-arity program that we constructed. It is easy to verify that there exists a stable model where the following term t satisfies the predicate *plan*:

$$t = move(c, b, move(b, a, move(c, table, 0)))$$

The term t encodes the plan of moving c to the *table*, b on top of a, and finally c on top of b. The same t is also an answer for the cautious open query $\lambda x.plan(x)$ to the program, and encodes a secure plan for the goal.

However, if the initial location of b were not known, i.e., $on(b, table, 0) \leftarrow$ is replaced by $on(b, table, 0) \vee on(b, c, 0) \leftarrow$, then the above plan is no longer secure, as the first step is not executable in the case where b is on top of c. Here, the answer

$$t = move(c, b, move(b, a, move(c, table, move(b, table, 0))))$$

to the cautious open query $\lambda x.plan(x)$ encodes a secure plan. □

We finally remark that higher-arity $\mathbb{FDNC}$ programs can be used to encode suitable fragments of the predicate version of the action language $\mathcal{K}$, but omit further of this issue.

8 Conclusion

8.1 Related Work

Several works have considered decidable logic programs with function symbols that are related to our work, including [11, 10, 8, 7, 51], as well as function-free programs that have similar semantic properties [30]. We discuss this now in more detail.

Datalog$_{\mathbf{nS}}$. A close relative of $\mathbb{FDNC}$ is $Datalog_{nS}$ [11, 10], which provides an extension of the Datalog language in deductive databases with function symbols in a way that is more liberal in spirit than in $\mathbb{FDNC}$ programs. The syntax of $Datalog_{nS}$ allows for rules in which atoms with complex terms affect atoms with less complex terms, which is not allowed in $\mathbb{FDNC}$ programs. On the other hand, $Datalog_{nS}$ features neither of disjunction, negation, and constraints, and thus has to be compared with $\mathbb{F}$; modulo minor differences, ordinary and higher-arity $\mathbb{F}$ programs are $Datalog_{nS}$ programs.

Chomicki and Imieliński presented in [11] a an algebraic approach to compile the least Herbrand models of $Datalog_{nS}$ programs (i.e., their single stable models) via homomorphisms into finite structures, on which query answering can be performed. Different representations of these structures, viz. a graph specification and an equational specification that uses a congruence relation, have been described and analyzed; other representation methods for restricted classes of programs in the literature were also discussed. The compilation technique in [11] does not extend to $\mathbb{FDNC}$ programs, which can have multiple (even infinitely many) stable models. The techniques of knots, which constitute building blocks of stable models, handles multiplicity of models by knot sharing, in a way such that every stable model can be readily assembled from knots.

Notably, ordinary and higher-arity $\mathbb{F}$ have lower complexity than $Datalog_{nS}$, at least regarding data complexity (which was considered in [11]). As reported there, cautious entailment of ground queries in $Datalog_{nS}$ is ExpTime-complete with respect to data complexity, i.e., w.r.t. the size of the set of facts in the program. On the other hand, cautious entailment of ground queries from $\mathbb{F}$ programs (which coincides with brave entailment) is feasible in polynomial time; the same holds for higher-arity $\mathbb{F}$ programs when the number of parameters in each rule is bounded by a constant, since then the parameter grounding $pgr(P)$ and the $\mathbb{FDNC}$-reduct of P have polynomial size; thus, the ground query can be answered in polynomial time when the rules are fixed. This continues to hold when facts added to P may also involve function symbols (in global positions only): complex terms in facts can be compiled away in polynomial time (e.g., by doing a partial instantiation and introducing fresh predicate and constant symbols for ground terms). Hence, w.r.t. data complexity, our $\mathbb{F}$-programs constitute a meaningful, tractable fragment of $Datalog_{nS}$. In [10], different evaluation strategies for query answering from $Datalog_{nS}$ programs have been considered; by their relationship to $\mathbb{F}$-programs, they can applied to the latter as well.

Finitely recursive and finitary programs. Our class $\mathbb{FN}$, which results from $\mathbb{FDNC}$ by disallowing constraints and disjunction, is a decidable subclass of the *Finitely Recursive Programs (FRPs)* [8, 7]. FRPs are normal logic programs P with function symbols that are restricted in a way such that in the grounding of P, each atom depends only on finitely many atoms. In this formalism, inconsistency checking is r.e.-complete and brave ground entailment is co-r.e.-complete in general [7]. For $\mathbb{FN}$ and our full class $\mathbb{FDNC}$, which implicitly obeys the condition of FRPs, these problems are ExpTime-complete. On the other hand, $\mathbb{FN}$ is not a subclass of the *Finitary Programs (FPs)* [8], which are those RFPs in whose grounding only finitely many atoms occur in odd cycles. For FPs, consistency checking is decidable, and brave and cautious entailment are decidable for ground queries but r.e.-complete for existential atomic queries. Note that for $\mathbb{FN}$, all these problems are decidable in exponential time. Finally, the explicit syntax of $\mathbb{FN}$, as well as of our other fragments of $\mathbb{FDNC}$, allows for effective recognition of such programs. FRPs and FPs, instead, suffer from undecidability of the conditions defining the classes, i.e., FRPs and FPs cannot be effectively recognized.

Omega-restricted logic programs. For logic programs with negation under stable model semantics, ω-restricted logic programs have been presented in [51]. These are normal logic programs that allow for function symbols of arbitrary arities and an unbounded number of variables, but have restricted syntax to ensure that they have the finite-model property, i.e., that finite answer sets always exist. The restriction is a generalization of classical stratification based on the existence of an acyclic ordering of the atom dependencies, which adds a special ω-stratum that holds all unstratifiable predicates of the logic program. In contrast, our $\mathbb{FDNC}$ programs as presented in this paper do not exclude cyclic dependencies, and they do not have the finite model property. Furthermore, $\mathbb{FDNC}$ programs have lower computational complexity. While consistency testing in general ω-restricted programs is 2-NEXPTIME-complete, the test can be done in EXPTIME for ordinary and in 2-EXPTIME for higher-arity $\mathbb{FDNC}$ programs.

Local extended conceptual logic programs. Another formalism related to $\mathbb{FDNC}$-programs are *Local Extended Conceptual Logic Programs (LECLPs)* [30] which evolved from [29]. Such programs are function-free but have answer sets over *open domains*, i.e., of the grounding of P with an arbitrary superset of the constants in P. LECLPs are syntactically restricted to ensure the forest-shape model property of answer sets. Deciding consistency of an LECLP P is feasible in 3-NEXPTIME, as one can ground P with double exponentially many constants in the size of P, and then use a standard ASP solver. For $\mathbb{FDNC}$, deciding the consistency is EXPTIME-complete and thus less complex.

Comparing the expressiveness of LECLPs and $\mathbb{FDNC}$ is intricate due the different settings. At least, both formalisms can encode certain description logics (e.g., $\mathcal{ALC}$). LECLPs may be more expressive than $\mathbb{FDNC}$ programs, since the expressive DL $\mathcal{ALCHOQ}$ is reducible to satisfiability in LECLPs. On the other hand, LECLPs undermine the general intuition behind minimal model semantics of logic programs. So-called *free rules* of the form $p(x) \vee not\ p(x) \leftarrow$; allow to unfoundedly add atoms for p in an answer set. $\mathbb{FDNC}$, instead, has no free rules, and each atom in a stable model of P must be justified from the facts of P.

Reductions of Description Logics to ASP Reductions of Description Logics to ASP have been considered e.g. in [1, 6, 50, 32, 31, 30]. Alsaç and Baral [1, 6] gave a reduction of $\mathcal{ALCQI}$ to normal function-free logic programs (i.e., Datalog with stable negation), which was geared towards the Herbrand domain of a knowledge base; by adding rules to generate inductively terms with a function symbol, they extended it to infinite domains. Their reduction is, in a sense, less constructive than the one given here and others, where function symbols are used to handle existential quantifiers by skolemization. Swift [50] reported a reduction of deciding satisfiability of $\mathcal{ALCQI}$ concepts to Datalog with stable negation, which exploits the finite model property of this problem. Heymans et al. [31, 30] reduced $\mathcal{SHIQ}$ (which subsumes $\mathcal{ALCQI}$) to their Conceptual Logic Programs and extensions; however, they used answer sets over open domains rather than the standard Herbrand domain.

Most relevant for the present paper is Hustadt et al.'s [32]. They reduced reasoning in $\mathcal{SHIQ}$ to the evaluation of a positive disjunctive Datalog program. The program is generated in three steps. First, the knowledge base is translated into first-order logic in a standard way. After that, resolution and superposition techniques are applied to saturate a clausal form of the transformation. Finally, function symbols are removed using new constant symbols.

The reduction of $\mathcal{ALC}$ to $\mathbb{FDC}$ in Section 5.2 is, in essence, similar to others and a close relative of the one in [32]. The main differences to the latter are with respect to step 2, where our method uses knots for compilation, and that our method aims at model building while the one in [32] is geared towards query answering. Notably, the disjunctive Datalog program constructed in [32] is generally exponential in the size of $\mathcal{K}$ (but is evaluable in CO-NP), while the $\mathbb{FDC}$ program is polynomial (but may need exponential time for evaluation).

Furthermore, the reduction contributes in two respects. First, the knowledge base $\mathcal{K}$ is rewritten on the DL syntax side into a normal form (which can be done very efficiently), rather than on the first-order logic side after the mapping. Second, a transformation into $\mathbb{FDC}$ as a language opens the possibility to use any dedicated evaluation algorithm for such programs, beyond a specific method (like the one in this paper).

Reasoning about Actions and Planning. As already discussed in the previous sections, the use of non-monotonic logic programs under answer set semantics as a tool for expressing and solving problems in reasoning about actions has been considered in many papers, including [14, 36, 6, 17, 52, 48, 49, 42]. The work presented here adds to these other works by providing an underpinning of the computational properties of non-monotonic logic programs with functions symbols that naturally emerge in this context, and, importantly, capture indefinitely long action sequences. They help in assessing the complexity of particular problems and may be useful to show that tractability can be achieved in some cases. Furthermore, our results provide algorithms to solve problems expressed in the language.

Using higher arity $\mathbb{FDNC}$, we can represent the Yale-Shooting scenario in Example 4 alternatively using a generic predicate $\mathit{holds}(f, x)$ to express truth of the fluent f in a situation x, where f is reified using a constant symbol, in the style of [6]; e.g., $\mathit{holds}(\mathit{Loaded}, \mathit{init})$ corresponds then to $\mathit{Loaded}(\mathit{init})$. Further predicates, e.g. $\mathit{abnormal}(f, x)$, can be used to express other aspects of fluents. While the syntax of $\mathbb{FDNC}$ does not allow reification of fluents with parameters, e.g. $on(A, B)$ to $\mathit{holds}(on(A, B), x)$, which is also used [6], this can be easily accommodated with tailored predicates, e.g. $\mathit{holds}_{on}(A, B,\ x)$; on the other hand, an extension of the syntax of $\mathbb{FDNC}$ programs that allows such terms in local positions is easily accomplished, and does not affect the worst case complexity.

8.2 Summary and Further Issues

In line with efforts to pave the way for effective Answer Set Programming engines with function symbols [8, 7], we presented $\mathbb{FDNC}$ programs as a decidable class of disjunctive logic programs with function symbols under stable model semantics.

$\mathbb{FDNC}$ and its subclass are a powerful tool for knowledge representation and reasoning for some applications involving infinite processes and objects, like evolving action domains. They are, by their intrinsic complexity, the proper fragment of logic programs to capture secure (alias conformant) planning in declarative action languages with a transition-based semantics like $\mathcal{K}$, $\mathcal{C}$, and similar languages, which is an ExpSpace-complete problem.

Notably, $\mathbb{FDNC}$ programs can have infinitely many and infinitely large stable models. To finitely represent those models, we introduced a technique that allows to reconstruct stable models as forests from so called *knots* from maximal founded set of knots. The finite representation technique allowed us to define an elegant decision procedure for brave reasoning in $\mathbb{FDNC}$, and may also be exploited for offline knowledge compilation to speed up online reasoning and model building, by precomputing and storing the knots of a program. From the precomputed set of knots, stable models can be built online comparatively fast.

Furthermore, we have characterized the complexity of reasoning in $\mathbb{FDNC}$ programs, which is lower than in the most popular of the recent approaches to enhance Answer Set Programming with functions symbols. $\mathbb{FDNC}$ and its subclasses provide *effective syntax* for expressing problems in PSpace, ExpTime, and ExpSpace using logic programs with function symbols.

The are several avenues for future research. One is to generalize the syntax of $\mathbb{FDNC}$, while keeping decidability and benign semantic properties. An interesting such extension is to allow rules of the form $R(y, x) \leftarrow R(x, y)$. The ability to define inverse relations, which are common in expressive DLs and related modal logics, would allow for the bi-directional flow of information between elements in the domain; the resulting programs are, as seem by a reduction to monadic second-order logic over trees *SkS*, still decidable. While the forest-shaped model property would remain, such relations are problematic due to the stable model semantics. In presence of inverse relations, testing the minimality of interpretation for a program becomes more involved. Intuitively, the justification of atoms in an interpretation can no longer be verified by only considering the structurally less complex atoms. To deal with these issues, we plan to generalize the knot-based technique for the finite representation of stable models.

$\mathbb{FDNC}$ and, in particular, the finite representation of stable models is a promising basis of developing algorithms for answering more complex queries than those considered in this paper. Since $\mathbb{FDNC}$ easily captures some basic DLs, the algorithms developed for $\mathbb{FDNC}$ may be applicable in other domains also. In general, query answering algorithms need to construct a set of models in order answer the query. The algorithms using the maximal founded set of knots as an input, would be relieved from computationally expensive model building since the relevant part of the model can be built using knots without the need to ensure the consistency.

Finally, implementation of $\mathbb{FDNC}$-programs is a subject of future work. Since the stable knots are defined as stable models of local programs (which are finite propositional disjunctive logic programs), the implementation will certainly include exporting parts of reasoning to one of the highly optimized answer set solvers currently available. In particular, recent extensions of the DLV system like DLVHEX, which implements hex programs [20] that feature external function calls (by which limited skolemization could be simulated), may be attractive for this.

Acknowledgments

We are grateful to the reviewers of the preliminary LPAR 2007 paper and our colleagues for useful comments on this paper.

A Proofs and Constructions

A.1 Auxiliary Lemma

Proof of Lemma 1. The statement (i) follows directly from the fact that in the basic fragment $\mathbb{F}$ we can state unary and binary facts. Indeed, P is consistent iff $P \cup \{Q(c) \leftarrow\} \models_b \exists x.Q(x)$, where Q and c are fresh symbols not occurring in P. Hence, whenever a fragment allows for unary facts, the consistency problem in that fragment can be reduced in logarithmic space to brave entailment of existential unary queries in the same fragment. The same can be shown for binary existential queries, and also for ground queries.

It is easy to see that the statement (ii) holds for existential queries. Indeed, for an arbitrary logic program P, the following hold:

1) $P \models_b \exists x, y.R(x, y)$ iff $P \cup \{Q(x) \leftarrow R(x, y)\} \models_b \exists x.Q(x)$, and

2) $P \models_b \exists x.A(x)$ iff $P \cup \{W(x, f(x)) \leftarrow A(x)\} \models_b \exists x.W(x, y)$,

where Q, W, and f are fresh symbols not occurring in P. This defines a logarithmic space reduction from brave entailment of binary existential queries to unary ones, and vice versa. Since even in the basic $\mathbb{F}$ fragment the syntax allows to add the necessary rule, the claim follows.

As in the case above, by utilizing additional rules, brave entailment of binary ground queries can be reduced in logarithmic space to brave entailment of unary ground queries, and vice versa. Hence, the statement (ii) also holds for ground queries. We state the properties that allow for reduction. Let q be a binary ground atom, and let P be an an $\mathbb{FDNC}$ program. Due to the forest-shape model property, if q is not of the form (a) $R(c,d)$ or (b) $R(t,f(t))$, where c,d are constants, then $P \not\models_b q$. Therefore, without loss of generality, we can assume that binary queries over $\mathbb{FDNC}$ programs are of the form (a) or (b). The reduction then follows from the following properties:

a) $P \models_b R(c,d)$ iff $P \cup \{R'(c,d) \leftarrow; R''(x,y) \leftarrow R(x,y), R'(x,y); Q(y) \leftarrow R''(x,y)\} \models_b Q(d)$,

b) $P \models_b R(t,f(t))$ iff $P \cup \{Q(y) \leftarrow R(x,y)\} \models_b Q(f(t))$,

c) $P \models_b A(v)$ iff $P \cup \{R'(x,f(x)) \leftarrow A(x)\} \models_b R'(v,f(v))$,

where Q, R', R'', and f are fresh symbols not occurring in P.

For the statement (iii), it is easy to see that cautious entailment of unary open queries can be reduced in linear time to cautious entailment of binary open queries. Indeed, $P \models_c \lambda x.A(x)$ with the answer $x = t$ iff $P \cup \{R(x,f(x)) \leftarrow A(x)\} \models_c \lambda x,y.R(x,y)$ with the answer $x = t$, $y = f(t)$, where R and f are fresh symbols not occurring in P. For the reduction in the other direction, consider a $\mathbb{FDNC}$ program P and a query $\lambda x,y.R(x,y)$. We define the program P' obtained from P by adding

(a) for each pair c,d of constants of P, the rules

- $R'_{c,d}(c,d) \leftarrow$,
- $R_{c,d}(x,y) \leftarrow R'_{c,d}(x,y), R(x,y)$, and
- $A_{c,d}(y) \leftarrow R_{c,d}(x,y)$,

where $R'_{c,d}$, $R_{c,d}$ and $A_{c,d}$ are fresh symbols, and

(b) for each function symbol f of P, the rule $A_f(f(y)) \leftarrow R(x,f(x))$, where A_f is a fresh symbol.

It is easy to verify that $P \models_c \lambda x,y.R(x,y)$ iff at least one of the following holds:

1. for some pair c,d of constants of P, $P \models_c \lambda x.A_{c,d}(x)$, or

2. for some function symbol f of P, $P \models_c \lambda x.A_f(x)$.

where each of the predicate symbols in the heads is a fresh symbol. By this construction, cautious entailment of a binary open query can be decided by polynomially many cautious entailment problems of unary open queries that are constructible in polynomial time. Hence statement (iii) holds. □

Phase	Axiom		Result
Ph.1	$D \oplus \hat{C} \sqsubseteq E$	$\rightsquigarrow$	$D \oplus A \sqsubseteq E$, $\hat{C} \sqsubseteq A$
	$D \sqsubseteq E \oplus \hat{C}$	$\rightsquigarrow$	$D \sqsubseteq E \oplus A$, $A \sqsubseteq \hat{C}$
	$\mathbb{Q}R.\hat{C} \sqsubseteq E$	$\rightsquigarrow$	$\hat{C} \sqsubseteq A$, $\mathbb{Q}R.A \sqsubseteq E$
	$D \sqsubseteq \mathbb{Q}R.\hat{C}$	$\rightsquigarrow$	$D \sqsubseteq \mathbb{Q}R.A$, $A \sqsubseteq \hat{C}$
Ph.2	$\hat{C} \sqsubseteq \hat{D}$	$\rightsquigarrow$	$\hat{C} \sqsubseteq A$, $A \sqsubseteq \hat{D}$
	$C \sqcup D \sqsubseteq B$	$\rightsquigarrow$	$C \sqsubseteq B$,$D \sqsubseteq B$
	$B \sqsubseteq C \sqcap D$	$\rightsquigarrow$	$B \sqsubseteq C, B \sqsubseteq D$
Ph.3	$\mathbb{Q}R.B \sqsubseteq D$	$\rightsquigarrow$	$\top \sqsubseteq A \sqcup D$, $A \sqsubseteq \mathbb{Q}^{-}R.A'$, $A' \sqcap B \sqsubseteq \bot$
Ph.4	$C \sqsubseteq D \sqcup \neg E$	$\rightsquigarrow$	$C \sqcap E \sqsubseteq D$
	$C \sqcap \neg D \sqsubseteq E$	$\rightsquigarrow$	$C \sqsubseteq D \sqcup E$
	$\bot \sqcap D \sqsubseteq E$	$\rightsquigarrow$	$\emptyset$
	$D \sqsubseteq E \sqcup \top$	$\rightsquigarrow$	$\emptyset$
	$\top \sqcap D \sqsubseteq E$	$\rightsquigarrow$	$D \sqsubseteq E$
	$D \sqsubseteq E \sqcup \bot$	$\rightsquigarrow$	$D \sqsubseteq E$

where $\oplus \in \{\sqcap, \sqcup\}$, $\mathbb{Q} \in \{\forall, \exists\}$, concepts $\hat{C}$, $\hat{D}$ are not literal concepts, A, A' are fresh concepts, B is atomic, the rest are arbitrary.

Table 4: Rules for Rewriting into Normal Form

A.2 Normalization of $\mathcal{ALC}$ KBs

We show how to transform in linear time an arbitrary $\mathcal{ALC}$ KB $\mathcal{K}_1$ into a KB $\mathcal{K}_2$ such that $\mathcal{K}_2$ is in normal form, is safe, and $\mathcal{K}_1$ is satisfiable iff $\mathcal{K}_2$ is satisfiable (i.e., $\mathcal{K}_1$ and $\mathcal{K}_2$ are equi-satisfiable). For technical reasons, we assume that $\mathcal{ALC}$ KBs contain only concepts that are in *negation normal form*, i.e., negation may occur only in front of atomic concepts. It is well known that an arbitrary $\mathcal{ALC}$ concept can be transformed in linear time into an equivalent concept in negation normal form. We start with the transformation into normal form and then move to safety of KBs.

Given an arbitrary $\mathcal{ALC}$ KB $\mathcal{K}$, an equi-satisfiable KB $\mathcal{K}'$ in normal form can be obtained by exhaustive rewriting of axioms in $\mathcal{K}$ using the rules in Table 4. The rewriting is performed in 4 phases. It is easy to verify that the transformation is terminating, preserves the consistency, and after the exhaustive rewriting in the final Phase 4 yields a KB in normal form.

We analyze the computational complexity of rewriting in each of the phases. Following the standard assumption in Description Logics, we assume that each of the atomic concepts in $\mathbf{C}$ is of constant size, i.e., the size of the representation of atomic concepts does not depend on a particular knowledge base. Since the number of axioms in $\mathcal{K}$ is linear in $|\mathcal{K}|$, w.l.o.g. we assume that $\mathcal{K}$ contains only one axiom α.

It is easy to see that in Phase 1 the number of rewritings is bounded by $c + q$, where c and q respectively denote the number of binary connectives, and quantifiers ("$\forall$" or "$\exists$") occurring in

$\mathcal{K}$. Since each application of a rule removes an axiom and adds two axioms, the number of axioms resulting by rewriting α is bounded by $c+q$. Since the application of a rewrite rule to an axiom yields two axioms whose combined size increases by some fixed constant not depending on the size of the KB (due to the assumption on the size of atomic concepts), the rewriting in Phase 1 is feasible in linear time in the size of the initial KB.

Phase 2 is feasible in linear time in the size of the knowledge base obtained in Phase 1. Indeed, only linearly many rule applications can occur and each of the rewriting causes a constant overhead in the representation of new axioms.

Phase 3 that deals with the elimination of quantifier in the antecedent of an axiom is clearly linear in the size of the KB obtained in Phase 3.

In Phase 4 the number of rewrite steps is bounded by the number of negation symbols and occurrences of $\top$ and $\bot$ in the knowledge base resulting from Phase 3, i.e., it is clearly linear.

Since each phase requires at most linear time in the size of the input, we conclude that normalizing a KB $\mathcal{K}$ is feasible in linear in the size of $\mathcal{K}$.

We now show that each $\mathcal{ALC}$ KB in normal form can be transformed in linear time into a safe KB in normal form while preserving the consistency. For a given KB $\mathcal{K}$ we can construct the safe knowledge base $\mathcal{K}'$ by modifying $\mathcal{K}$ in the following way:

– for each individual name i occurring in $\mathcal{K}$, adding the assertion $Dom(i)$ to $\mathcal{K}$,

– for each role R of $\mathcal{K}$, adding $Dom \sqsubseteq \forall R.Dom$ to $\mathcal{K}$, and

– replacing each axiom $\top \sqsubseteq D \in \mathcal{K}$ of type (T3), by $Dom \sqsubseteq D$,

where Dom is a fresh concept name not occurring in $\mathcal{K}$. Indeed, $\mathcal{K}'$ is safe and in normal form by construction. It is easy to verify that $\mathcal{K}$ is consistent iff $\mathcal{K}'$ is consistent. Indeed, if $\mathcal{I}$ is a first-order interpretation that is the model of $\Theta(\mathcal{K})$, then we can extend $\mathcal{I}$ to be a model of $\Theta(\mathcal{K}')$ by extending $\mathcal{I}$ to interpret Dom as the whole domain of $\mathcal{I}$. For the other direction, suppose $\mathcal{K}'$ is consistent. Since $\mathcal{ALC}$ has the forest-shaped model property $\boxed{\text{REF}}$, due to the construction, there exists a model $\mathcal{I}$ of $\Theta(\mathcal{K}')$ where every domain element satisfies Dom. Then, trivially, $\mathcal{I}$ is a model of $\Theta(\mathcal{K})$. The construction of $\mathcal{K}'$ is clearly linear in the size of $\mathcal{K}$.

A.3 Brave Entailment in $\mathbb{FDNC}$

Proof of Theorem 8. Suppose (A) holds, i.e., there exists some $I \in SM(P)$ such that $A(t) \in I$, for some term t. By Theorem 1, $I^c \in SM(P^{\mathsf{G}})$. Let $G := I^c$. By Proposition 3, $\mathbb{K}(I)$ is a set of knots that is founded w.r.t. P and $S(G)$. Due to the definition of $\mathbb{K}_P$ and Proposition 5, we have that $\mathbb{K}_P$ is compatible w.r.t. $S(G)$. It remains to show that (ii) in (B) holds. Consider $\mathcal{E}_{\mathbb{K}(I)}$. Due to the fact that $A(t) \in I$ and the construction of $\mathcal{E}_{\mathbb{K}(I)}$, for some constant c there exists $K \in \mathbb{K}(I)$ such that $\mathsf{st}(G,c) \approx \mathsf{st}(K,\mathbf{x})$ and $\langle K, A\rangle \in \mathcal{E}_{\mathbb{K}(I)}$. Due to the definition of $\mathbb{K}_P$, $\mathbb{K}(I) \subseteq \mathbb{K}_P$. Therefore $K \in \mathbb{K}_P$. Moreover, it trivially holds that $\langle K, A\rangle \in \mathcal{E}_{\mathbb{K}(I)}$ implies $\langle K, A\rangle \in \mathcal{E}_{\mathbb{K}_P}$. Therefore, (ii) holds.

Suppose (B) holds. The facts that $G \in SM(P^{\mathsf{G}})$, and that $\mathbb{K}_P$ is compatible w.r.t. $S(G)$ imply that $\mathcal{F}(G,\mathbb{K}_P) \neq \emptyset$ and each $I \in \mathcal{F}(G,\mathbb{K}_P)$ is a stable model of P (see Theorem 2). The condition (ii) and the construction of forest-shaped interpretations ensure that some $I \in \mathcal{F}(G,\mathbb{K}_P)$ contains an atom $A(t)$, where t is some term. □

Proof of Theorem 10. Similar to the proof of Theorem 8, and thus omitted. □

A.4 Open Queries

Proof of Proposition 9. For the "only if" direction, suppose a term t is such that $P \models_c A(t)$. Suppose c is the constant of t, and $t = f_n(\ldots f_1(c)\ldots)$. Let $t_0, \ldots, t_n$ be the list of subterms of t ordered w.r.t. increasing term depth, i.e., $t_0 = c$ and $t_i = f_i(\ldots f_1(c)\ldots)$, where $i \in \{1, \ldots, n\}$.

Define the sequence $s = [\frac{L_0}{c}, \frac{L_1}{f_1}, \ldots, \frac{L_n}{f_n}]$, where $L_i = \{K_{\downarrow \mathbf{x}} \mid I \in SM(P),\ K = I \cap \mathcal{HB}_{t_i}\}$, for $i \in \{0, \ldots, n\}$. We verify that s is convergent.

Since P is consistent, each L_i is a nonempty subset of $\mathbb{K}_P$. Since $P \models_c A(t)$, the sequence s trivially satisfies the conditions (1) and (3) in Definition 5.7. Suppose (2) is not satisfied. Then there exists some $j \in \{1, \ldots, n\}$, some $K \in L_{j-1}$ and some $K' \in \mathbb{K}_P$ such that $\mathsf{st}(K, f_j(\mathbf{x})) \approx \mathsf{st}(K', \mathbf{x})$ and $K' \notin L_j$. Take the smallest index j for which the statement above holds. Then there exists a sequence $K_0, \ldots, K_{j-1}, K_j$ of knots in $\mathbb{K}_P$ such that the sequence $N = [\frac{K_0}{t_0}, \ldots, \frac{K_{j-1}}{t_{j-1}}, \frac{K_j}{t_j}]$ has the following properties:

- $K_i \in L_i$ for each $i \in \{0, \ldots, j-1\}$, while $K_j \notin L_j$;
- $\mathsf{st}(K_i, f_{i+1}(\mathbf{x})) \approx \mathsf{st}(K_{i+1}, \mathbf{x})$ for each $i \in \{0, \ldots, j-1\}$.

Let $S = \mathsf{st}(K_0, \mathbf{x})$. Due to the definition of trees, we know that there exists a tree T induced by $\mathbb{K}_P$ starting at S such that $N \in T$. Consider the stable model $I \in SM(P)$ where $\mathsf{st}(c, I) \approx S$. Such I must exists due to the way we defined L_0. By the semantic characterization (see Theorem 3), I can be represented as $I = I^c \cup (T^{c_1})_{\downarrow} \cup \ldots \cup (T^{c_n})_{\downarrow}$, where $\{c_1, \ldots, c_n\}$ is the set of all constants of P, and each T^{c_i} is a tree induced by $\mathbb{K}_P$ starting at $\mathsf{st}(c_i, I^c)$. Suppose $c_1 = c$. Simply define $I' := I^c \cup (T)_{\downarrow} \cup (T^{c_2})_{\downarrow} \cup \ldots \cup (T^{c_n})_{\downarrow}$. By Theorem 2, we have that I' is also a stable model of P. We arrive at a contradiction to the assumption that $K_j \notin L_j$. Indeed, $K_j = (I' \cap \mathcal{HB}_{t_j})_{\downarrow \mathbf{x}}$, and, due to the definition of s, $K_j \in L_j$.

For the other direction we show that the failure of (A) implies the failure of (B). Suppose for each term t, $P \not\models_c A(t)$. Furthermore, assume there exists a converging sequence $s = [\frac{L_0}{c}, \frac{L_1}{f_1}, \ldots, \frac{L_n}{f_n}]$ for A, where $L_0 = \mathsf{seeds}(c, P)$. First, we reconstruct the term encoded in the sequence. Let $t_0 = c$, while t_n is defined inductively as $t_i := f_i(t_{i-1})$, where $1 \leq i \leq n$. Consider the term t_n. By assumption, there exists a model I of P such that $A(t_n) \notin I$. There are two possibilities.

a) $t_i \hat{\in} I$, for each $i \in \{0, \ldots, n\}$. Due to the definition of $\mathbb{K}_P$ and the fact that $\mathbb{K}(I)$ is founded, we have that each $K_i := (\mathcal{HB}_{t_i} \cap I)_{\downarrow \mathbf{x}}$ is in $\mathbb{K}_P$, where $0 \leq i \leq n$. By assumption, we have $K_0 \in L_0$. The condition (2) in Definition 5.7 implies that $K_n \in L_n$. Since $A(\mathbf{x}) \notin K_n$, we have that s is not a converging sequence for A due to violation of (3) in Definition 5.7.

b) For some i, where $0 < i \leq n$, we have $t_i \hat{\notin} I$. Note that $t_0 \hat{\in} I$ since it is a constant. Take the smallest m, where $0 \leq m < n$, such that $t_{m+1} \hat{\notin} I$. As it was argued, each $K_i := (\mathcal{HB}_{t_i} \cap I)_{\downarrow \mathbf{x}}$ is in $\mathbb{K}_P$, where $0 \leq i \leq m$. By assumption, we have $K_0 \in L_0$. The condition (2) in Definition 5.7 implies that $K_m \in L_m$. Since $K_m := (\mathcal{HB}_{t_m} \cap I)_{\downarrow \mathbf{x}}$ and $t_{m+1} \hat{\notin} I$, we have that $f_m(\mathbf{x}) \notin \mathsf{succ}(K_m)$. We have that s is not a converging sequence for A due to violation of (1) in Definition 5.7.

In both cases s is a converging sequence for A, which contradicts the assumption. □

Proof of Theorem 13 (ExpSpace-*Hardness*). Consider a language L over an alphabet Σ which is in ExpSpace. Then there is a Turing machine $M = (S, \Sigma, \delta, s_0)$ as in Definition 6.3 that decides membership of a given word I in L on a tape whose length is bounded by an exponential in the size of I. We construct a $\mathbb{FD}$ program $P(M, I)$ of size polynomial in M and I such that acceptance of I by M is equivalent to the existence of an answer for an open query $\exists x.A(x)$ under cautious entailment.

For convenience, we assume here that I is not the empty word. Suppose the number of cells (the space) used by M on the input I is bounded by $m := 2^{as}$, where as is polynomial in the size of I. The reduction relies on keeping two addresses of the cells in the work tape, each of which is represented using $as = \log_2 m$ bits. The first address is the position of the read/write (r/w) head, which is encoded by the unary predicate symbols $rwpos^b_0$, …, $rwpos^b_{as}$, $b \in \{0,1\}$. For each bit of the address, we dedicate two symbols and will ensure that exactly one of them holds for each term. In our encoding, terms will represent stages reached in the computation of the machine on some path. Similarly, the second address is the one of the *observed cell*, which is encoded by the unary predicate symbol $opos^b_0$, …, $opos^b_{as}$, $b \in \{0,1\}$. Intuitively, the observed cell is the single cell of the machine for which the correct state transition will be ensured by the program. By non-deterministically generating all cells for observation in parallel, and exploiting the properties of cautious entailment of open queries we will ensure that accepting computations of M (represented by terms) can be singled out.

We sketch the construction of the program $P(M, I)$ in steps. We need rules for checking the equality of the r/w head address and the address of the observed cell. To this end, for a bit b, let $\bar{b} = 1 - b$ denote the complement of b. For the comparison of separate bits in the two addresses, we add the following rule

$$equ_i(x) \leftarrow opos^b_i(x), rwpos^b_i(x) \qquad \text{for all } i \in \{0, \ldots, as\} \text{ and } b \in \{0,1\}. \tag{7}$$

The equality of two addresses at some point of computation is then expressed easily by the rule

$$rwoequ(x) \leftarrow equ_0(x), \ldots, equ_{as}(x). \tag{8}$$

The inequality is also easily expressed by the rules

$$nonequ(x) \leftarrow opos^b_i(x), rwpos^{\bar{b}}_i(x) \tag{9}$$

for all $i \in \{0, \ldots, as\}$ and $b \in \{0,1\}$.

We move to the representation of the initial configuration of the machine, which we do from the perspective of an observed cell. To this end, we add, for $0 \leq i \leq as$, the facts

$$rwpos^0_i(st) \ \leftarrow \ , \tag{10}$$

$$state_{s_0}(st) \ \leftarrow \ , \tag{11}$$

$$opos^1_i(st) \vee opos^0_i(st) \ \leftarrow \ . \tag{12}$$

Intuitively, (10) sets the position of the r/w head to the left most cell and (11) set the machine into the start state, while (12) non-deterministically chooses an observed cell of the tape. To represent the content of each observed cells in the initial configuration, we proceed as follows.

For each symbol $\alpha \in \Sigma$, we use a designated unary predicate symbol $symbol_\alpha$. Let $n \geq 0$ be the position of the last symbol of I written on the tape, i.e., $I = I_0 I_1 \cdots I_n$ is on positions 0,…,n. For each position $i \leq n$ with binary representation $i = b_0 \cdots b_{as} = i$ and $\alpha = I_i$, we add the rule

$$symbol_\alpha(st) \leftarrow opos^{b_0}_0(st), \ldots, opos^{b_{as}}_{as}(st). \tag{13}$$

For all other positions, the symbols are blank. Assuming that $n = b^*_0 \cdots b^*_{as}$ in binary, we express this with rules

$$symbol_b(st) \leftarrow opos_1^{b^*_1}(x), \ldots, opos_{j-1}^{b^*_{j-1}}(x), opos_j^1(x), \tag{14}$$

for all $j \in \{0 \ldots, as\}$ such that $b^*_j = 0$.

This describes the initial configuration; note that it is captured by the whole set of models for the program described so far. Although each model captures only the content of one (the observed) cell, the contents of the whole work tape is entirely captured as the addresses of the observed cells cover the whole work space of M.

To encode the transitions, it is handy to view δ as a table. For each tuple $t = \langle s, \alpha, s', \alpha', D\rangle$ such that $\delta(s, \alpha) = \langle s', \alpha', D\rangle$, we use a function symbol $\bar{t}$ and define the following rules:

$$\begin{aligned} next(x, \bar{t}(x)) &\leftarrow rwoequ(x), state_s(x), symbol_\alpha(x), && (15)\\ next(x, \bar{t}(x)) &\leftarrow nonequ(x), state_s(x), && (16)\\ state_{s'}(\bar{t}(x)) &\leftarrow next(x, \bar{t}(x)), && (17)\\ symbol_{\alpha'}(\bar{t}(x)) &\leftarrow rwoequ(x), next(x, \bar{t}(x)), && (18)\\ move_D(\bar{t}(x)) &\leftarrow next(x, \bar{t}(x)). && (19)\end{aligned}$$

The rules above are explained as follows. If the r/w head is at the position of the observed cell, and the symbol and the state are correct for the transition, the transition is made (15). If the r/w head is not at the position of the observed cell, the transition is made blindly (16). The single case where the transition is not made is if the r/w head is at the position of the observed cell, but either the symbol or the state is not the right one. The rule (17) sets the new state, while (18) sets the new symbol of the observed cell. The rule (19) triggers the movement of the r/w head. The effect of $move_D$ is explained next. Moving the r/w head boils down to adding or subtracting one bit from the address. To this end, we use unary predicates $shift^b_0$,…, $shift^b_{as}$, $b \in \{0, 1\}$, to simulate the values of the carry bit. When the r/w head position changes, the last bit should be inverted. This is stated by the rules

$$\begin{aligned} shift^1_{as}(x) &\leftarrow move_{+1}(x), && (20)\\ shift^1_{as}(x) &\leftarrow move_{-1}(x), && (21)\\ shift^0_{as}(x) &\leftarrow move_0(x). && (22)\end{aligned}$$

The position of r/w head after shifting is then defined by the following rules for each $j \in \{0, \ldots, as\}$, $j' \in \{1, \ldots, as\}$, and $b \in \{0, 1\}$:

$$\begin{aligned} rwpos^{\bar{b}}_j(y) &\leftarrow shift^1_j(y), rwpos^b_j(x), next(x, y), && (23)\\ rwpos^b_j(y) &\leftarrow shift^0_j(y), rwpos^b_j(x), next(x, y), && (24)\\ shift^b_{j'-1}(y) &\leftarrow move_{+1}(y), shift^1_{j'}(y), rwpos^b_{j'}(x), next(x, y), && (25)\\ shift^{\bar{b}}_{j'-1}(y) &\leftarrow move_{-1}(y), shift^1_{j'}(y), rwpos^b_{j'}(x), next(x, y), && (26)\\ shift^0_j(x) &\leftarrow move_0(x). && (27)\end{aligned}$$

Furthermore, we have to state that the address of the observed cell does not change, i.e., is fixed for a model. This expressed by the rules

$$opos^b_i(y) \leftarrow opos^b_i(x), next(x, y), \tag{28}$$

for each $i \in \{0, \ldots, as\}$ and $b \in \{0, 1\}$. Finally, we ensure that the symbol written in the observed cell does not change if it is not affected by the transition. This is expressed by the following inertia rule for each $\alpha \in \Sigma$:

$$symbol_\alpha(y) \leftarrow nonequ(x), symbol_\alpha(x), next(x, y). \tag{29}$$

This completes the description of the program $P(M, I)$. It is not hard to see that $P(M, I)$ has exactly $m = 2^{as}$ minimal models (and thus stable models, as in $P(M, I)$ no negation occurs) that are induced by different choices of the position of the observed cell. Let $R^0, \ldots, R^{m-1}$ be these models ordered with respect to the position of the observed cell, i.e., R^0 is the one for first position 0 while R^{m-1} is the one for the last position $m - 1$.

Without loss of generality, we view a run of M on an input I as a sequence $t_1, \ldots, t_n$ of transitions, and assume that it is always non-empty. The run is accepting, if after performing t_n, the machine enters the accepting state *accept*. We establish the following lemmas.

Lemma 2. *If the machine M accepts the input I on the run $t_1, \ldots, t_n$, $n \geq 1$, then $P(M, I) \models_c state_h(u)$, where $u = \bar{t}_n(\ldots \bar{t}_1(st) \ldots)$.*

Proof. Suppose that $I^0 = Ib \cdots b$ is the word describing the initial tape contents, and that after executing the transitions $t_1, \ldots, t_i$, (i) I^i is the word given by the tape contents, (ii) s^i is the state of the machine, and (iii) pos^i is the position of the r/w head.

We show that for each R^w, $w \in \{0, \ldots, m - 1\}$, we have $state_{accept}(u) \in R^w$. To this end, we show that in R^w the content of the observed cell w, the state, and the r/w head position are correctly reflected through the computation. More formally, let $u_0 := st$, and $u_i := \bar{t}_i(u_{i-1})$, where $0 < i \leq n$. Then we argue that, for each $j \in \{0, \ldots, n\}$, (i) $symbol_\alpha(u_j) \in R^w$ whenever $\alpha = I^j_w$, i.e., α is written in cell w, (ii) $state_{s_j}(u_j) \in R^w$, and (iii) pos^j is, encoded, in binary, by the atoms $rwpos^b_i(u_j) \in R^w$, $0 \leq i \leq as$. Note that this will prove the lemma, since $s^n = accept$.

We proceed by induction on $j \geq 0$. The base case $j = 0$ is clear by the encoding of the initial word (rules (13) and (14)), the initial r/w head position (facts (10)) and the initial state (fact (11)).

For the inductive case, assume the claim holds for $0 \leq j < n$ and consider $j + 1$. By the induction hypothesis, $symbol_\alpha(u_i) \in R^w$, $state_{s_j}(u_j) \in R^w$, and pos^j is described by the atoms $rwpos^b_i(u_j) \in R^w$. There are now, by the rules (7) – (9) two disjoint cases: either $nonequ(u_j) \in R^w$ or $rwoequ(u_j) \in R^w$. In the former case, $next(u_j, t_{j+1}(u_j)) \in R^w$ by the rule (16); by the rule (29), we then have $symbol_\alpha(u_{j+1}) \in R^w$. In the latter case, $next(u_j, t_{j+1}(u_j)) \in R^w$ by the rule (15); by the rule (18), we then have $symbol_{\alpha'}(u_{j+1}) \in R^w$. In both cases, R^w contains $symbol_\alpha(u_{j+1})$ where $I^{i+1}_w = \alpha$. Hence (i) holds for $j + 1$.

As for (ii), as we have $next(u_j, t_{j+1}(u_j)) \in R^w$, by the rule (17) we have $state_{s_{i+1}}(u_{i+1}) \in R^w$, and thus (ii) holds for $j + 1$. Finally, the rules (19) and (20) – (27) effect that atoms $pos^b_i(u_{j+1})$ which correctly represent pos^{j+1} are derived. Hence, (iii) holds for $j + 1$. □

Lemma 3. *If $P(M, I) \models_c \lambda x.state_{accept}(x)$, then there exists an accepting run of M.*

Proof. Suppose $P(M, I) \models_c state_{accept}(u)$. By assumption, the initial state is not *accept* and thus $u = \bar{t}_n(\ldots \bar{t}_1(st) \ldots)$, where $n \geq 1$. Let $u_0 := st$, and $u_i := \bar{t}_i(u_{i-1})$, where $0 < i \leq n$. Then, in each model R^w, we must clearly must have $next(u_{i-1}, t_i(u_{i-1}))$ for each $0 < i \leq n$ (otherwise, $state_{accept}(u_n)$ would not be contained in R^w).

For each $i \in \{0, \ldots, n\}$, define (i) the word $I^i = \alpha_0 \cdots \alpha_{m-1}$ where α_j is such that $symbol_{\alpha_j}(u_i) \in R^j$, $0 \leq j < m$, (ii) s^i as the state s such that $state_s(u_i) \in R^w$, and (iii) pos^i as the integer which, in binary, is encoded by the facts $rwpos_i^{b_i}(u_j) \in R^w$, i.e., $pos^i = b_0 \cdots b_{as}$, where $w \in \{0, \ldots, m-1\}$ is arbitrary.

We claim that each I^i, s^i, and pos^i is well-defined and is the tape contents, state, and r/w head position, respectively, after the partial run $t_1, \ldots, t_i$ of M on the input I. Since $s^n = accept$, this will prove the lemma.

The proof is by induction on $i \geq 0$. For the base case $i = 0$, by construction, I^0 clearly is the initial tape contents, $s^0 = s_0$, and $pos^0 = 0$ by the facts and rules (10) – (14). Suppose the claim holds for $0 \leq i < n$ and consider $i + 1$. Assume $t_{i+1} = \langle s, \alpha, s'\alpha', D\rangle$. Since we have $next(u_i, t_{i+1}(u_i))$ in each R^w, we must have $state_{s'}(u_{i+1})$ in R^w by rule (17); since no other fact $state_{s''}(u_{i+1})$ can be in R^w, s^{i+1} is well-defined. Furthermore, we must have $state_s(u_i)$ in R^w and either (a) $rwoequ(u_i) \in R^w$ or (b) $nonequ(u_i) \in R^w$; by the induction hypothesis and the rules (7) – (9), (a) is the case if $pos^i = w$ and (b) if $pos^i \neq w$. In case (a), we must have $symbol_\alpha(u_i) \in R^w$ and $symbol_{\alpha'}(u_{i+1}) \in R^w$ by rule (18), and in case (b) $symbol_\alpha(u_{i+1}) \in R^w$ by rule (29), where $symbol_\alpha(u_i) \in R^w$. Since no other facts $symbol_{\alpha''}(u_{i+1})$ can be in R^w, I^{i+1} is well-defined. Finally, we must have $move_D(u_{i+1})$ in R^w by rule (19); by the induction hypothesis and the rules (20) – (27), we have facts $rwpos_j^{b_j}(u_{i+1})$ in R^w, $0 \leq j \leq as$, such that $b_0, \ldots, b_{as}$ represents $pos^i + D = pos^{i+1}$ in binary.

Summing up, I^{i+1}, s^{+1}, and pos^{i+1} are all well-defined and encode tape contents, state, and r/w head position, respectively, after the partial run $t_1, \ldots, t_{i+1}$ of M on the input I, which concludes the induction step. □

As $P(M, I)$ and $\lambda x.state_{accept}(x)$ are constructible in polynomial time from M and I, from Lemmas 2 and 3 the claimed EXPSPACE-hardness result follows for $\mathbb{FD}$, $\mathbb{FDN}$, and $\mathbb{FDNC}$; by replacing the disjunctive guessing rules (12) with unstratified rules $opos_i^1(st) \leftarrow not\ opos_i^0(st)$; $opos_i^0(st) \leftarrow not\ opos_i^1(st)$, we obtain the result for $\mathbb{FN}$ and $\mathbb{FNC}$. □

A.5 Reasoning in $\mathbb{FN}$

Proof of Proposition 11. Let I be a stable model of $F(P)$. The properties (1), (2) and (3) hold by the construction of $F(P)$, i.e., due to the fact that I is a stable model that satisfies (F1)-(F4) rules of $F(P)$. Suppose I does not satisfy (4), i.e., for some term $t \hat{\in} I$ and some unary predicate A of P, either (a) $\{A(t), \bar{A}(t)\} \cap I = \emptyset$, or (b) $\{A(t), \bar{A}(t)\} \subseteq I$. In case (a), we have $\{A(t) \leftarrow Dom(t); \bar{A}(t) \leftarrow Dom(t)\} \subseteq F(P)^I$. Since $Dom(t) \in I$, I is not a model of $F(P)^I$. Contradiction. Assume the case (b). There are two possibilities: (i) either $A^c(t) \in I$, or (ii) $A^c(t) \notin I$. Assume (i). Since the single rule (the reduct of the rule of type (F7)), where $A^c(t)$ occurs in the head, is not in $F(P)^I$, we have that I is not a minimal model of $F(P)^I$. Contradiction. Assume (ii). Then the rule $A^c(t) \leftarrow A(t), \bar{A}(t)$ is in $F(P)^I$, and the body is in I by the assumption (b). It follows that I is not a model of $F(P)$. Contradiction. To sum up, both (a) and (b) lead to the contradiction to the assumption that I is a stable model of $F(P)$. Analogously to the argument for (4), one can show that the property (5) holds.

For the other direction, assume an interpretation I of $F(P)$ for which the given properties hold. It is easy to see that such an interpretation satisfies each of the rules in $F(P)^I$. We verify that I a minimal model of $F(P)^I$. By the construction of $F(P)$, each minimal model of $F(P)$ has to satisfy (1), (2) and (3). Therefore, if I is not a minimal model of $F(P)^I$, there should exists a model $H \subset I$ of $F(P)^I$ for which (4) or (5) does not hold. We arrive to a contradiction. Due to the rules of type (F5) or (F6) in $F(P)$, H cannot be a model of $F(P)^I$. □

Proof of Proposition 12. Suppose I is a minimal model of P. We know that I is forest-shaped. Let J be a Herbrand interpretation for $TR(P)$ defined as the smallest set of atoms satisfying the following conditions:

a) $I \subseteq J$,

b) $S(c, d) \in J$, for each pair c, d of constants of P,

c) if $t \hat{\in} J$, then $Dom(t) \in J$,

d) if $t \hat{\in} J$ and f is a function symbol of P, then $S(t, f(t)) \in J$,

e) if $t \hat{\in} J$ and $A(t) \notin J$, then $\bar{A}(t) \in J$, and

f) if $S(s, t) \in I$ and $R(s, t) \notin I$, then $\bar{R}(s, t) \in J$,

where A and R are predicate symbols of P. We show that J is a stable model of $TR(P)$. Assume that it is not the case. There are two possibilities.

- J is not a model of $TR(P)^J$. Since $F(P) \subseteq TR(P)$, we have $F(P)^J \subseteq TR(P)^J$. From the construction of J it follows that J is a model of $F(P)^J$. Then there has to exists some ground rule $C(\vec{v}_1) \leftarrow Q_1(\vec{v}_1), \ldots, Q_m(\vec{v}_m), \bar{W}_1(\vec{t_1}), \ldots, \bar{W}_n(\vec{t_n})$ in $TR(P)^J$ such that

  ($\star$) $\{Q_1(\vec{v}_1), \ldots, Q_m(\vec{v}_m), \bar{W}_1(\vec{t_1}), \ldots, \bar{W}_n(\vec{t_n}\} \subseteq J$ and $C(\vec{v}_1) \notin J$.

  Due to construction, P contains the rule $W_1(\vec{t_1}) \vee \ldots \vee W_n(\vec{t_n}) \leftarrow Q_1(\vec{v}_1), \ldots, Q_m(\vec{v}_m)$. Since I is a model of P, then either (a) $\{Q_1(\vec{v}_1), \ldots, Q_m(\vec{v}_m)\} \nsubseteq I$ or (b) $\{W_1(\vec{t_1}) \vee \ldots \vee W_n(\vec{t_n})\} \cap I \neq \emptyset$. In case (a), by the definition of J, ($\star$) does not hold. In case (b), for some $W_i(\vec{t_i})$ of the rule, $\bar{W}_i(\vec{t_i}) \notin J$ and, hence, ($\star$) does not hold.

- J is a model but is not a minimal model of $TR(P)^J$. Since $F(P) \subseteq TR(P)$, we have $F(P)^J \subseteq TR(P)^J$. Then we also have that J is a model of $F(P)^J$, but is not minimal. We arrive at a contradiction, since J is a minimal model of $F(P)^J$ due to the construction of J and Proposition 11.

For the other direction, let I be a stable model of $TR(P)$. Let J be the interpretation obtained by restricting I to the predicates of P. Suppose J is not a model of P. Then $\mathsf{Ground}(P)$ contains a rule $W_1(\vec{t_1}) \vee \ldots \vee W_n(\vec{t_n}) \leftarrow Q_1(\vec{v}_1), \ldots, Q_m(\vec{v}_m)$, where $n, m > 0$, such that $\{Q_1(\vec{v}_1), \ldots, Q_m(\vec{v}_m)\} \subseteq I$ and $\{W_1(\vec{t_1}), \ldots, W_n(\vec{t_n})\} \cap I = \emptyset$. By construction, $\mathsf{Ground}(TR(P))$ contains $r = C(\vec{t_1}) \leftarrow Q_1(\vec{v}_1), \ldots, Q_m(\vec{v}_m), \bar{W}_1(\vec{t_1}), \ldots, \bar{W}_n(\vec{t_n}), not\ C(\vec{t_1})$. Furthermore, due to the assumption that $I \in SM(TR(P))$ and Proposition 11, we have $\{Q_1(\vec{v}_1), \ldots, Q_m(\vec{v}_m), \bar{W}_1(\vec{t_1}), \ldots, \bar{W}_n(\vec{t_n}\} \subseteq I$. It holds that either $C(\vec{v}_1) \notin J$, or $C(\vec{v}_1) \in I$. The former case implies that I is not a model of $TR(P)$. In the later case we get that I is not a minimal model of $TR(P)$ since by definition r is the single rule where $C(\vec{t_1})$ occurs in the head. However, the rule $r \notin TR(P)$ by definition of the GL reduct. We arrived at a contradiction that I be a stable model of $TR(P)$. At this point we know that J is a model of P. J might not be minimal, but that is enough to prove that P is consistent. □

A.6 Higher-arity $\mathbb{FDNC}$

Proof of Theorem 25. We analyze the impact of restricted variable interaction in higher-arity $\mathbb{FDNC}$. As easily verified, the atoms that can be motivated in the stable models are of the particular form. Let P be a higher-arity $\mathbb{FDNC}$ program, and let $\mathit{pterms}(P)$ be the set of *proper* terms defined as the smallest set such that

a) if $c \in \mathcal{HU}^P$ is a constant and $c \notin \mathit{ld}(P)$, then $c \in \mathit{pterms}(P)$;

b) if $t \in \mathcal{HU}^P$ is a complex term such that (1) in its local positions there are only constants from $\mathit{ld}(P)$, and (2) in its global positions are the terms from $\mathit{pterms}(P)$, then $t \in \mathit{pterms}(t)$.

Note that $\mathit{pterms}(P)$ is closed under subterms. Let $\mathit{patoms}(P)$ be the set of all *proper* atoms for P, which are the atoms in $\mathcal{HB}^P$ that have constants from $\mathit{ld}(P)$ in the local positions and terms from $\mathit{pterms}(P)$ in the global positions.

Due to the syntax of higher-arity $\mathbb{FDNC}$, given any Herbrand interpretation I of P, a rule r in the reduct P^I contains a non-proper atom iff it contains a non-proper atom in the body. Hence, every minimal model $J \subseteq I$ of P^I must satisfy $J \subseteq \mathit{patoms}(P)$. Thus if P' is the program obtained from $\mathsf{Ground}(P)$ by deleting each rule that contains an atom $A \notin \mathit{patoms}(P)$, then P^I and P'^I have the same minimal models. This implies that $SM(P) = SM(P')$. Moreover, only proper atoms can be motivated in stable models of P, i.e., $I \subseteq \mathit{patoms}(P)$ holds for each $I \in SM(P)$.

Trivially, $SM(P') = \{I \mid \mathit{red}(I) \in SM(\mathit{red}(P'))\}$. On the other hand, we can easily verify that $\mathit{red}(P') = \mathsf{Ground}(\mathit{red}(\mathit{pgr}(P)))$. Since $SM(\mathsf{Ground}(\mathit{red}(\mathit{pgr}(P))) = SM(\mathit{red}(\mathit{pgr}(P))))$, it follows that $SM(P) = \{I \mid \mathit{red}(I) \in SM(\mathit{red}(\mathit{pgr}(P)))\}$ as claimed.

□

References

[1] G. Alsaç and C. Baral. Reasoning in description logics using declarative logic programming. Technical report, Department of Computer Science and Engineering, Arizona State University, 2001.

[2] H. Andreka and I. Nemeti. The generalised completeness of Horn predicate logic as programming language. *Acta Cybernetica*, 4(1):3–10, 1978.

[3] Asparagus homepage. `http://asparagus.cs.uni-potsdam.de/`, Since 2005.

[4] F. Baader, S. Brandt, and C. Lutz. Pushing the EL envelope. In L. P. Kaelbling and A. Saffiotti, editors, *IJCAI*, pages 364–369. Professional Book Center, 2005.

[5] F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation and Applications.* Cambridge University Press, 2003.

[6] C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving.* Cambridge University Press, 2002.

[7] S. Baselice, P. A. Bonatti, and G. Criscuolo. On finitely recursive programs. In *ICLP*, 2007. Accepted for publication.

[8] P. A. Bonatti. Reasoning with infinite stable models. *Artif. Intell.*, 156(1):75–111, 2004.

[9] M. Cadoli and F. Donini. A survey on knowledge compilation. *AI Communications*, 10(3-4):137–150, 1997.

[10] J. Chomicki. Depth-bounded bottom-up evaluation of logic programs. *Journal of Logic Programming*, 25(1):1–31, 1995.

[11] J. Chomicki and T. Imielinski. Finite representation of infinite query answers. *ACM Trans. Database Syst.*, 18(2):181–223, 1993.

[12] E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov. Complexity and expressive power of logic programming. *ACM Computing Surveys*, 33(3):374–425, 2001.

[13] A. Darwiche and P. Marquis. A knowledge compilation map. *Journal of Artificial Intelligence Research*, 17:229–264, 2002.

[14] Y. Dimopoulos, B. Nebel, and J. Koehler. Encoding planning problems in nonmonotonic logic programs. In *Proceedings of the European Conference on Planning 1997 (ECP-97)*, pages 169–181. Springer Verlag, 1997.

[15] J. Dix, U. Furbach, and A. Nerode, editors. *Logic Programming and Nonmonotonic Reasoning, 4th International Conference, LPNMR'97, Dagstuhl Castle, Germany, July 28-31, 1997, Proceedings*, volume 1265 of *Lecture Notes in Computer Science*. Springer, 1997.

[16] T. Eiter, W. Faber, N. Leone, G. Pfeifer, and A. Polleres. A logic programming approach to knowledge-state planning, II: The $\mathtt{dlv}^{\mathcal{K}}$ system. *Artificial Intelligence*, 144(1-2):157–211, 2003.

[17] T. Eiter, W. Faber, N. Leone, G. Pfeifer, and A. Polleres. A logic programming approach to knowledge-state planning: Semantics and complexity. *ACM Transactions on Computational Logic*, 5(2):206–263, Apr. 2004.

[18] T. Eiter and G. Gottlob. On the computational cost of disjunctive logic programming: Propositional Case. *Annals of Mathematics and Artificial Intelligence*, 15(3/4):289–323, 1995.

[19] T. Eiter and G. Gottlob. Expressiveness of stable model semantics for disjunctive logic programs with functions. *J. Log. Program.*, 33(2):167–178, 1997.

[20] T. Eiter, G. Ianni, R. Schindlauer, and H. Tompits. A uniform integration of higher-order reasoning and external evaluations in answer set programming. In L. P. Kaelbling and A. Saffiotti, editors, *Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI-05)*, pages 90–96. Professional Book Center, 2005.

[21] T. Eiter, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. A deductive system for non-monotonic reasoning. In Dix et al. [15], pages 364–375.

[22] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. *clasp* : A conflict-driven answer set solver. In C. Baral, G. Brewka, and J. S. Schlipf, editors, *LPNMR*, volume 4483 of *Lecture Notes in Computer Science*, pages 260–265. Springer, 2007.

[23] M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Comput.*, 9(3/4):365–386, 1991.

[24] M. Gelfond and V. Lifschitz. Representing actions in extended logic programming. In *JICSLP*, pages 559–573, 1992.

[25] E. Giunchiglia and V. Lifschitz. An action language based on causal explanation: Preliminary report. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI '98)*, pages 623–630, 1998.

[26] S. Hanks and D. V. McDermott. Nonmonotonic logic and temporal projection. *Artif. Intell.*, 33(3):379–412, 1987.

[27] P. Haslum and P. Jonsson. Some results on the complexity of planning with incomplete information. In S. Biundo and M. Fox, editors, *ECP*, volume 1809 of *Lecture Notes in Computer Science*, pages 308–318. Springer, 1999.

[28] J. Herbrand. *Logical Writings.* Harvard University Press, 1971. Edited by Warren D. Goldfarb.

[29] S. Heymans. *Decidable Open Answer Set Programming.* PhD thesis, Theoretical Computer Science Lab (TINF), Department of Computer Science, Vrije Universiteit Brussel, Pleinlaan 2, B1050 Brussel, Belgium, February 2006.

[30] S. Heymans, D. V. Nieuwenborgh, and D. Vermeir. Nonmonotonic ontological and rule-based reasoning with extended conceptual logic programs. In *ESWC*, pages 392–407, 2005.

[31] S. Heymans and D. Vermeir. Integrating semantic web reasoning and answer set programming. In M. de Vos and A. Provetti, editors, *Proceedings ASP-2003 — Answer Set Programming: Advances in Theory and Implementation*, volume 78 of *CEUR Workshop Proceedings*, pages 194–208. CEUR-WS.org, 2003.

[32] U. Hustadt, B. Motik, and U. Sattler. Reducing SHIQ-description logic to disjunctive datalog programs. In *Proceedings KR-2004*, pages 152–162. AAAI Press, 2004.

[33] U. Hustadt, R. A. Schmidt, and L. Georgieva. A survey of decidable first-order fragments and description logics. *JoRMiCS*, 1:251–276, 2004.

[34] A. Itai and J. A. Makowsky. Unification as a complexity measure for logic programming. *Journal of Logic Programming*, 4:105–117, 1987.

[35] H. J. Levesque, F. Pirri, and R. Reiter. Foundations for the situation calculus. *Electron. Trans. Artif. Intell.*, 2:159–178, 1998.

[36] V. Lifschitz. Answer Set Planning. In D. D. Schreye, editor, *Proceedings of the 16th International Conference on Logic Programming (ICLP'99)*, pages 23–37, Las Cruces, New Mexico, USA, Nov. 1999. The MIT Press.

[37] V. Lifschitz. Answer Set programming and plan generation. *Artificial Intelligence*, 138:39–54, 2002.

[38] V. W. Marek and J. B. Remmel. On the expressibility of stable logic programming. In *LPNMR*, pages 107–120, 2001.

[39] V. W. Marek and M. Truszczyński. Stable models and an alternative logic programming paradigm. In K. Apt, V. W. Marek, M. Truszczyński, and D. S. Warren, editors, *The Logic Programming Paradigm – A 25-Year Perspective*, pages 375–398. Springer, 1999.

[40] W. Marek, A. Nerode, and J. Remmel. How Complicated is the set of stable models of a recursive logic program? *Annals of Pure and Applied Logic*, 56:119–135, 1992.

[41] W. Marek, A. Nerode, and J. Remmel. The stable models of a predicate logic program. *Journal of Logic Programming*, 21(3):129–153, 1994.

[42] A. R. Morales, P. H. Tu, and T. C. Son. An extension to conformant planning using logic programming. In M. M. Veloso, editor, *IJCAI*, pages 1991–1996, 2007.

[43] B. Motik, I. Horrocks, and U. Sattler. Bridging the gap between OWL and relational databases. In *Proc. of WWW 2007*, pages 807–816, 2007.

[44] I. Niemelä. Logic programming with stable model semantics as constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3–4):241–273, 1999.

[45] I. Niemelä and P. Simons. Smodels - an implementation of the stable model and well-founded semantics for normal lp. In Dix et al. [15], pages 421–430.

[46] W. J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *J. Comput. Syst. Sci.*, 4(2):177–192, 1970.

[47] K. Schild. A correspondence theory for terminological logics: Preliminary report. In *IJCAI*, pages 466–471, 1991.

[48] T. C. Son, C. Baral, N. Tran, and S. A. McIlraith. Domain-dependent knowledge in answer set planning. *ACM Transactions on Computational Logic*, 7(4):613–657, 2006.

[49] T. C. Son, P. H. Tu, M. Gelfond, and A. R. Morales. Conformant planning for domains with constraints-a new approach. In M. M. Veloso and S. Kambhampati, editors, *AAAI*, pages 1211–1216. AAAI Press / The MIT Press, 2005.

[50] T. Swift. Deduction in ontologies via ASP. In *Proceedings LPNMR-2004*, volume 2923 of *LNCS/LNAI*, pages 275–288. Springer, 2004.

[51] T. Syrjänen. Omega-restricted logic programs. In T. Eiter, W. Faber, and M. Truszczynski, editors, *LPNMR*, volume 2173 of *Lecture Notes in Computer Science*, pages 267–279. Springer, 2001.

[52] P. H. Tu, T. C. Son, and C. Baral. Reasoning and planning with sensing actions, incomplete information, and static causal laws using answer set programming. *Journal of the Theory and Practice of Logic Programming*, 7(4):377–450, 2007.

[53] S. Woltran. Answer Set Programming: Model applications and proofs-of-concept. Technical Report WP5, Working Group on Answer Set Programming (WASP, IST-FET-2001-37004), July 2005. Available at `http://www.kr.tuwien.ac.at/projects/WASP/report.html`.