TECHNISCHE
UNIVERSITÄT
WIEN
Vienna University of Technology

DISSERTATION

Nonmonotonic Logic Programs with Function Symbols

ausgeführt zum Zwecke der Erlangung des akademischen Grades eines Doktors der technischen Wissenschaften unter der Leitung von

O. Univ. Prof. Dipl.-Ing. Dr. Thomas Eiter
Institut für Informationssysteme 184/3
Abteilung für Wissenbasierte Systeme

eingereicht an der Technischen Universität Wien
Fakultät für Informatik

von

Mantas Šimkus
Matrikelnummer 0527230
Lerchenfelder Straße 39/37
1070 Wien

Wien, am 18. Mai 2010

....
Mantas Šimkus

To my grandparents,
Algirdas and Marija,
Juozas and Sofija

Abstract

Rule-based formalisms play a dominant role in Computer Science, and in Artificial Intelligence in particular. We focus on *Answer Set Programming (ASP)*, which is a rule-based declarative problem solving paradigm that emerged from Logic Programming and Nonmonotonic Reasoning. In a broad sense, ASP couples logic programming with an additional construct of *default negation*, for which ASP is recognized as particularly well-suited for modeling and solving problems that involve common-sense reasoning. The *answer set semantics* assigns to each program a set of its intended models, or solutions, that are called *stable models* (or, alternatively, *answer sets*). Using suitable encodings, ASP has been fruitfully exploited to provide solutions to problems in many application domains, including planning, diagnosis, belief revision, configuration, data integration, security engineering, text classification and others.

Current decidable ASP frameworks and their implementations are based mainly on *function-free* languages. In these languages, the stable models of a program are always finite relational structures, while infinite structures are disallowed. It is widely acknowledged that the function-free setting leads to expressiveness drawbacks and can be inconvenient for knowledge representation. In a nutshell, it prohibits modeling infinite processes, indefinite time, recursive data structures, and, generally, problems where it is necessary to create new objects in the spirit of first-order existential quantification. Function symbols, in turn, are a very convenient means for generating infinite domains and objects, and allow a more natural representation of problems in the above domains. Unfortunately, an unrestricted use of function symbols makes ASP highly undecidable.

The need to represent problems with unbounded domains raises the challenge to single out fragments of ASP with function symbols that have sufficient expressiveness and still retain the decidability of the standard reasoning tasks. The main contributions of this thesis are two (families of) such fragments, which we call $\mathbb{FDNC}$ and $\mathbb{BD}$. The two languages are powerful formalisms for rule-based modeling of applications with potentially infinite processes or objects, accommodating common-sense reasoning through nonmonotonic negation. They can be applied in solving planning problems, modeling and reasoning about recursive data structures—especially, tree-shaped data, like HTML or XML documents—and to represent ontologies in some description logics. We develop novel algorithms for important ASP reasoning problems, and give a detailed account of the computational complexity of reasoning in $\mathbb{FDNC}$ and $\mathbb{BD}$ programs, and

also in a range of syntactic fragments of these languages.

Kurzfassung

Regelbasierte Formalismen spielen in der Informatik und vor allem in der Künstlichen Intelligenz eine wichtige Rolle. Wir konzentrieren uns hier auf Antwortmengen-Programmierung (ASP), ein Paradigma für regelbasierte deklarative Problemlösung, welches aus dem Gebiet der Logischen Programmierung und Nichtmonotonen Schlussfolgerung hervorgeht. Im weiteren Sinne verbindet ASP die logische Programmierung mit dem zusätzlichen Konstrukt der Default Negation, wodurch ASP besonders gut geeignet ist, Probleme zu modellieren und zu lösen, welche „Common-sense Schließen“ beinhalten. Die Antwortmengen-Semantik ordnet jedem Programm eine Menge von intendierten Modellen zu, die sogenannten stabilen Modelle (auch Antwortmengen genannt). Mit geeigneten Kodierungen wird ASP ausgiebig genutzt, um Lösungen für Probleme in vielen Anwendungsdomänen zu finden, darunter Planen, Diagnose, Meinungsberichtigung, Konfiguration, Datenintegration, Sicherheitstechnik, Text Klassifikation, und vieles mehr.

Derzeit bekannte entscheidbare Klassen von ASP Programmen und deren Implementierungen basieren hauptsächlich auf funktionsfreien Sprachen. Aus diesem Grund sind stabile Modelle von Programmen solcher Klassen im Wesentlichen endliche relationale Strukturen; unendliche Strukturen sind nicht zulässig. Es gibt allgemeine Übereinstimmung, dass die funktionsfreien Sprachen eine eingeschränkte Ausdruckskraft und Nachteile bei der Wissensrepräsentation haben. Kurz gesagt unterbinden diese die Modellierung von unendlichen Prozessen, unbegrenzter Zeit, rekursiven Datenstrukturen, und von Problemen allgemein, welche die Erzeugung von neuen Objekten ähnlich wie bei der existentiellen Quantifikation in der Logik erster Stufe benötigen. Funktionssymbole bieten auf der anderen Seite die Möglichkeit zur Erzeugung von unendlichen Domänen und Objekten, und sie erlauben eine natürlichere Repräsentation von Problemen in den zuvor genannten Bereichen. Leider macht eine uneingeschränkte Verwendung von Funktionssymbolen ASP hohem Maße unentscheidbar.

Die Notwendigkeit zur Repräsentation von Problemen mit unbegrenzten Domänen bringt die Herausforderung, Fragmente von ASP mit Funktionssymbolen zu finden, die ausreichende Ausdruckskraft besitzen und für die gleichzeitig die üblichen Schlussfolgerungsprobleme entscheidbar sind. Die Hauptbeiträge dieser Dissertation sind zwei entscheidbare Fragmente von ASP, die wir $\mathbb{FDNC}$ and $\mathbb{BD}$ nennen. Die zwei Sprachen sind ausdrucksstarke Formalismen für die regelbasierte Modellierung von Anwendungen mit möglicherweise unendlichen Prozessen oder Objekten, die auch Common-sense Schließen mit nichtmonotoner Negation unterstützen. Sie können zur Lösung von Planungsproblemen angewendet werden, zur Modellierung und zum Schlussfolgern über rekursive Datenstrukturen—im Speziellen baumförmige Datenstrukturen wie HTML oder XML Dokumente—, und sie können auch zur Repräsentation von Ontologien in einige Beschreibungslogiken verwendet werden. Wir entwickeln neue Algorithmen für wichtige ASP Schlussfolgerungsprobleme und geben eine detailierte Beschreibung der Berechnungskomplexität des Schließens in allgemeinen $\mathbb{FDNC}$ und $\mathbb{BD}$ Programmen, wie auch in einer Reihe von syntaktischen Fragmenten dieser Sprachen.

Acknowledgements

First of all, I would like to say thanks to my advisor, Prof. Thomas Eiter. It was a great luck and opportunity to work with such an excellent researcher and teacher. Thank you for the guidance, discussions, suggestions and support during the work on this thesis. Thank you also for the encouragement and friendliness.

I am grateful to the reviewers of the papers underlying this thesis for their useful comments and suggestions for improvement, as well as to Piero Bonatti, Georg Gottlob, Nicola Leone, Stijn Heymans and many other colleagues working in the field.

My thanks also go to all the colleagues in the KBS and the DBAI groups at the Institute of Information Systems. It was a pleasure to work in an environment where high-quality research is done with excitement. I'm delighted to have met so many great people here.

I would like to thank my parents Nijolė and Remigijus for all the love and care that they have given me. My thanks also to my brothers, Karolis and Kristijonas.

I am also grateful to Fiu and Jutta, who brought a lot of fun into my life in Vienna.

Finally, I would like to thank my wife Magdalena. Without her love and support, this thesis would not exist.

The work on this thesis was partially supported by the Austrian Science Fund (FWF) grant P20840 and the EC project OntoRule (IST-2009-231875).

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Rule-based formalisms play a dominant role in many fields of Computer Science. For instance, they are used in Databases as expressive query languages, and in Knowledge Representation (KR) as powerful means for declarative problem solving, mostly in the form of rule-based Logic Programming. In this thesis we focus on *Answer Set Programming (ASP)*, which is a declarative problem solving paradigm that emerged from Logic Programming and Nonmonotonic Reasoning [Bar02, Lif02, MT99, Nie99]. In a broad sense, ASP enriches logic programming with an additional construct called *default negation* (or *negation as failure*), which allows to infer a negative fact from the absence of a proof of the contrary. For instance, a navigating robot may employ such negation to infer that a path is not blocked based on the absence of evidence of a path-obstructing object. Default negation is widely appreciated for its suitability to model common-sense reasoning in the presence of incomplete information. It also provides a natural solution to some fundamental Artificial Intelligence problems, such as the *frame problem*, and is thus adequate for reasoning about dynamic domains. Formalisms that support default negation are inherently *nonmonotonic* because conclusions that are inferred using default negation may need to be withdrawn if new knowledge is added to the theory.

The nonmonotonic negation in ASP is interpreted under the *answer set semantics* [GL91], which assigns to each program a set of its intended models, called *stable models* (or, alternatively, *answer sets*). In contrast to logic programs in standard Prolog, an ASP program may have none, one or multiple stable models, which in turn can be generated by efficient ASP solvers like DLV [ELM$^+$97], Smodels [NS97], clasp [GKNS07] and others. This provides an effective way to deal with numerous AI problems that do not enjoy unique solutions: the problem is encoded as an ASP program, a solver is used to obtain the stable models of the program, and finally the solutions are extracted from the generated stable models. For instance, we can encode a planning domain as an ASP program in such a way that the stable models generated by a solver correspond to the possible sequences of actions that lead an agent to the desired goal.

ASP is recognized as particularly well-suited for modeling and solving problems that involve common-sense reasoning, and has been fruitfully exploited to provide solutions in a wide range of applications domains. They include planning, diagnosis, belief revision, configuration, data integration, security engineering, text classification and many others. As a consequence, ASP has evolved into a primary knowledge representation formalism. We refer the reader to [Wol05] for a more detailed discussion and an overview of applications, whose number has rapidly increased in the last years.

1.1 Motivation

The answer set semantics was originally defined in the setting of a general first-order language [GL91]. However, current decidable ASP frameworks and their implementations are mainly based on *function-free* languages, and are extensions of DATALOG with negation under the answer set semantics. In these languages, the stable models of a program are finite relational structures built over the constants occurring in the program. In some more sophisticated languages, the domain of constants of the program may be extended with additional (functional) terms (see e.g., [Syr01, GST07]). However, the stable models remain finite relational structures over the extended domain, and only finitely many stable models for a program exist. In fact, most solvers transform an input program into a propositional program in an initial *grounding step*. Thus, succinctness apart, these ASP languages can be seen as *propositional* and lack important features of first-order logic. In particular, they lack the possibility to assert the existence of new objects on demand, using (some form of) existential quantification. It is widely acknowledged that the resulting expressiveness limitations can be very inconvenient for knowledge representation (cf. [Bon04, CCIL08a]). In a nutshell, many important structures that require unbounded or even infinite domains cannot be naturally represented in ASP, including infinite processes, indefinite time, and recursive data structures.

Function symbols, in turn, are a very convenient means for generating infinite domains and objects, and extending ASP to support them has gained considerable attention in recent years (see, e.g., [Syr01, Bon04, BBC09, GST07, CCIL08a, LL09]). They provide a form of existential quantification that overcomes the aforementioned drawbacks, and allow a more natural representation of problems in several domains, like the ones we discuss next.

AI Planning

AI Planning is a prominent area of application of ASP. The excellent book [Bar02] (recommended for background) devotes a whole chapter to this subject, and the applications of ASP in planning have been explored in many works, including [SZ95, Lif02, DNK97, Lif99, EFL$^+$04, TSB07, SBTM06, STGM05, MTS07], making it a prominent subfield of research in logic-based knowledge representation and reasoning.

When representing a planning problem, it is imperative to have a suitable encoding of time. In one form or another, this is usually done by using a unary predicate $Init$ to indicate the initial time instant, and a binary predicate $Succ$ to relate successive time points. A timeline is then modeled as a relational structure

$$\mathit{Init}(t_0), \mathit{Succ}(t_0, t_1), \mathit{Succ}(t_1, t_2), \mathit{Succ}(t_2, t_3), \mathit{Succ}(t_3, t_4), \ldots,$$

where each t_i is an object corresponding to a point in time. The specification of the particular planning domain is then built on top of this structure. For instance, in case of ASP, we may use the rule

$$B(y) \leftarrow A(x), \mathit{Succ}(x, y)$$

to specify that the fluent B is true at a time point if A is true at the previous moment. For a more interesting example, consider the following *inertia* rule

$$A(y) \leftarrow A(x), \mathit{Succ}(x, y), \mathit{not}\ \neg A(y)$$

that uses default and *strong* negation to state that the fluent A remains true over time, unless it is proved to be false.

Since the universe of a function-free ASP program is a finite set of constants, only a bounded fragment of the above timeline can be represented in one program. Clearly, this rules out a general representation of planning domains. A common way to circumvent this is to instantiate an extended domain that may allow for a 'sufficiently long' timeline, the size of which must be estimated by the user and given as a parameter. A notable example of such a solution is found in the $\mathrm{DLV}^{\mathcal{K}}$ front-end of DLV [EFL+03] which implements the action language $\mathcal{K}$ [EFL+04]. However, the loss of generality and the overhead caused to the user by this partial solution are evident. Additionally, it may incur high space requirements and does not scale to large instances. This is an acknowledged limitation addressed in [GKK+08], where the authors consider a method for *incremental* evaluation of ASP programs.

In contrast, one constant together with a single function symbol are enough to generate an infinite timeline (or a finite one of an arbitrary length, as required by the problem), effectively overcoming the limitations above. In particular, the infinite timeline can be generated using the program P consisting of three rules:

$$\begin{aligned} \mathit{Init}(c) &\leftarrow\ , \\ \mathit{Succ}(c, f(c)) &\leftarrow\ , \\ \mathit{Succ}(y, f(y)) &\leftarrow\ \mathit{Succ}(x, y). \end{aligned}$$

The program P has a single stable model, which is exactly the following infinite relational structure over terms:

$$\mathit{Init}(c),\ \mathit{Succ}(c, f(c)),\ \mathit{Succ}(f(c), f(f(c))),\ \mathit{Succ}(f(f(c)), f(f(f(c))))\ldots$$

Here the constant c denotes the initial time point, and each term $f(t)$ denotes the time point that follows t.

Recursive Data Structures

In other contexts, it is not the power to express infinite structures that is important, but rather the ability to generate (possibly finite) structures whose size is not bounded by (a parameter depending on) the program. One notable example are *recursive data structures*, like lists and trees, that cannot be fully supported in ASP if only constants are available. This problem exposes traditional ASP as lacking important features of fully-fledged programming languages, and is perhaps the most notable reason motivating the research into extending ASP with function symbols (cf. discussions in [Bon04, CCIL08a]).

Function symbols allow to model recursive data structures in a natural way, and are in fact widely used for this purpose in standard Prolog. For instance, a full binary tree can be represented using the following (recursive) rules with function symbols:

$$\begin{aligned}
\mathit{Node}(c) &\leftarrow\ , \\
\mathit{Inner}(x) &\leftarrow \mathit{Node}(x), \mathit{not}\ \mathit{Leaf}(x), \\
\mathit{Leaf}(x) &\leftarrow \mathit{Node}(x), \mathit{not}\ \mathit{Inner}(x), \\
\mathit{Node}(f_1(x)) &\leftarrow \mathit{Node}(x), \mathit{Inner}(x), \\
\mathit{Node}(f_2(x)) &\leftarrow \mathit{Node}(x), \mathit{Inner}(x).
\end{aligned}$$

In the above example, the constant c is dedicated to be the root node, and via the rules with negation each node in the tree is classified to be either a leaf node or an inner node. Since we are interested in a full binary tree, each inner node must have two child nodes, which are created by the last two rules with function symbols. The generated trees can be processed using additional rules.

The particular capability to model trees would allow to deploy ASP on the Web. In this increasingly important context, rules with default negation may be useful for commonsense reasoning about semistructured data, like XML and HTML documents (see, e.g., [GK04b] where the authors apply a rule-based language to extract data from the Web).

Combining Description Logics and Rule-based Languages

Extensions of ASP with function symbols have potential impact in the development of *hybrid languages* that integrate *Description Logics (DLs)* and rule-based languages. The motivation for these formalisms comes mainly from the Semantic Web, where *ontologies* expressed in DLs are intended to describe and structure complex Web resources, making them readily available for automated agents [MvH04, PSHH04]. In turn, query languages based on rules with default negation are seen as expressive means for an automated agent to access these ontologies in a declarative way.

The integration of DLs and rule-based languages has received considerable attention in the last decade (see, e.g., [ADG$^+$05, PFT$^+$04, Ros06, EIP$^+$06, Eit07] for surveys and references), and many research efforts have been aimed at identifying languages that support *tight integration* into expressive formalisms that can simultaneously describe ontologies in DLs and support declarative rule-based access to them. However, the fundamentally different syntactic and semantic assumptions underlying the two families make it hard to identify decidable languages that are expressive enough for such purposes (see, e.g., [dBEPT06, EIP$^+$06, Ros06] for discussion).

A crucial difference is that, as we have mentioned, function-free languages like DATALOG have only finite structures as intended models. Indeed, DATALOG was designed and intended for reasoning over finite databases, under the assumption that only the objects explicitly mentioned in the database exist. In contrast, DL-based ontologies are usually theories in restricted fragments of first-order logic that support existential quantification, and are thus able to refer to or imply the existence of objects that are not explicitly named in the ontology. Existential quantification plays an important role in ontologies. It is in fact supported even by the simplest fragments of the standard Web Ontology Languages (such as the EL and QL profiles of OWL 2 [CHM$^+$08]), and there are many important DLs (such as the ones underlying the Lite and DL profiles of OWL) in which ontologies may have infinite and only infinite models that can not be captured by the models of a function-free program.

Enriching ASP with the ability to introduce fresh objects—as we do in this thesis—is a way to at least partially overcome the aforementioned difference and tightly integrate the two paradigms into an expressive hybrid knowledge representation language. This has already been attempted, for example, in [CGK08, CGL09] where the authors extend plain DATALOG with (restricted) existential quantification to obtain the language DATALOG$\pm$ that can accommodate constraints in some *light-weight* DLs that do not support disjunction. In the languages we propose in this thesis—which allow for function symbols rather than existential quantification—the presence of disjunction makes it possible to express DL ontologies in more expressive DLs. Furthermore, the presence of default negation gives the possibility of enhancing traditional reasoning with some form of non-monotonic inference.

1.2 Challenges and State of the Art

Using function symbols, we can easily generate infinite domains and, as we have already illustrated, represent some relevant problems from many different areas. Unfortunately, an unrestricted use of function symbols makes ASP highly undecidable. In fact, already inference from Horn logic programs becomes undecidable [AN78], and equipped with negation under the answer set semantics, it is at the second level of the analytical hierarchy (deciding the existence of a stable model is Σ_1^1-complete, cf. [MNR94, MR03, MNR92]). Intuitively, function symbols make the Herbrand universe infinite, and a program can have infinitely many possibly infinite stable models. The huge complexity then stems from the answer set semantics, which requires each answer set to satisfy a minimality property that quantifies over interpretations. In the presence of function symbols, we must quantify over possibly infinitely many infinite structures.

Many researches were discouraged by the above negative result, and adding function symbols to ASP was deemed as infeasible and barred from main-stream research for almost two decades. However, the need to represent problems with unbounded domains and to deploy the common-sense reasoning and declarative problem solving power of ASP has lead to reconsidering this position. Thus in recent years significant attention was devoted to the identification of meaningful fragments of ASP with function symbols that have sufficient expressiveness for certain applications, but still retain the decidability of the standard reasoning tasks, or at least ensure lower complexity of reasoning.

Several works have addressed this issue, including [CI93, Cho95, Syr01, Bon04, CCIL08a, BBC09], and some restricted classes of programs with function symbols have been identified. For example, *finitary programs* and *finitely recursive logic programs* were introduced in [Bon04, BBC09], while *finitely ground programs* were introduced in [CCIL08a]. In all three cases, the fragment is defined in terms of *syntactic restrictions on the grounding* of a logic program, rather than on the program itself. While this allows for identifying large and expressive decidable (or semidecidable) fragments, it also has a major disadvantage that seriously limits the applicability of the results: as the grounding of a program with function symbols is infinite, deciding if a given program belongs to the fragment (the *recognition* problem) is in fact undecidable. In contrast, the syntactic conditions that underly the fragments identified in [Syr01] and [CI93, Cho95] can be effectively checked, but the fragments are significantly less expressive. The programs in [Syr01] are defined by imposing certain *stratification* conditions, and as a result they can only have finite stable models of bounded size. Hence, they do not allow to reasoning about unbounded structures. The programs in [CI93, Cho95] allow to generate infinite structures but they do not support default negation, and are, in fact, a fragment of Horn logic programs. We refer the reader to Chapter 5 for more details and discussions of related work.

Despite the active research in the field, there were no available fragments of ASP with functions symbols that would

(i) support reasoning over infinite structures,

(ii) allow for a flexible use of default negation,

(iii) allow for an efficient recognition of programs,

(iv) be decidable, and

(v) have computational complexity that is adequate for relevant KR problems.

The identification of such fragments, along with the development of reasoning algorithms for them and the characterization of their computational complexity, is the main subject of this thesis. It is not trivial, and many challenges have to be overcome. They include the following:

- The minimality condition, that an interpretation must satisfy to be a stable model, quantifies over infinitely—actually, uncountably—many smaller structures. Testing minimality is particularly challenging in the presence of disjunction. In fact, we are not aware of any decidable fragments of ASP with function symbols that would support disjunction and allow for reasoning over infinite structures. If disjunction is disallowed (i.e., in normal programs), we can in principle resort to the so-called *fixed-point computation of an immediate consequence operator* to test minimality. However, this property is lost in the presence of disjunction, and we must devise genuine methods to ensure minimality without explicitly quantifying over uncountably many interpretations.

- Infinite stable models cannot be explicitly represented for reasoning purposes. Hence, to achieve correct and terminating algorithms for the relevant reasoning problems—existence of a stable model and truth of (different kinds of) queries in one or all the stable models of a given program—it is necessary to develop methods for reasoning about stable models without explicitly building them.

- Finally, we aim not only at establishing decidability of the formalisms we study, but to go further and obtain optimal complexity bounds. Hence naive terminating algorithms are not enough, and we must develop carefully crafted procedures whose requirement of resources (time and space) does not exceed those that arise from the provable hardness of the problem.

1.3 Contributions

In this thesis, we aim at identifying decidable fragments of ASP with function symbols that are effectively recognizable and support full negation as failure over infinite structures. *Our main contribution is to propose two such fragments, which we call* $\mathbb{FDNC}$ *and* $\mathbb{BD}$ *programs.* We consider a wide range of reasoning problems that are relevant in ASP, provide novel algorithms for solving them in both formalisms, and characterize their precise complexity. We analyze the effect of disallowing different constructors and of imposing additional syntactic restrictions on the rules of $\mathbb{FDNC}$ and $\mathbb{BD}$ programs, identifying fragments with better computational properties. Moreover, we discuss some possible applications of these rich families of logic programs, which include solving planning problems, modeling recursive data structures, and encoding ontologies in some description logics.

The fragments we obtain are defined by merely constraining the syntax of the rules, similarly as in [CI93, Cho95], and inspired by other important areas of knowledge representation, and, in particular, by Modal Logics and Description Logics [BCM$^+$03]. In these areas, decidability, algorithms and complexity results have been shown for various fragments of first-order logic, many of which do not have finite models. Most of these fragments allow for only a limited number of variables (often two), and impose some form of *guardedness*, which can be roughly understood as a syntactically restricted form of quantification that only allows to talk about relations between objects that are close to each other in a structure, and results in regular models that are conveniently similar to trees. While guardedness is of course a limitation, it is often claimed to be a robust cause for decidability [Var96, Grä99]. Furthermore, there is wide evidence suggesting that guardedness is not overly restrictive for many knowledge representation problems, and it is implicitly or explicitly present in many of the popular languages.

This kind of limited quantification is well understood in the context of classical first-order logic, but not in the context of ASP. Many reasoning techniques have been employed to show decidability and complexity results for description logics and related fragments of first order logic, including tableaux algorithms, automata theoretic techniques, and resolution. They all exploit in some way the guarded quantification and other syntactic restrictions, and allow to reason about infinite structures. However, transferring these techniques to the ASP setting is not easy. From the technical point of view, the nonmonotonic features of the language pose some significant challenges. The minimality condition mentioned above, that quantifies over all the possibly infinitely many interpretations of a program, goes beyond the expressive power of first-order logic and needs special methods. Adapting traditional ASP methods to these setting does not seem easier, in particular because most of them rely on explicit model constructions that are not feasible in the presence of infinite structures. Hence, we must develop novel reasoning techniques that allow for effective decision procedures, and carefully tailor them in such a way that the resulting algorithms are worst-case optimal.

The contributions of this thesis can be summarized as follows:

1. $\mathbb{FDNC}$ **Programs.** We introduce the class $\mathbb{FDNC}$ of logic programs, which allow for function symbols ($\mathbb{F}$), disjunction ($\mathbb{D}$), nonmonotonic negation ($\mathbb{N}$) under the answer set semantics [GL91], and constraints ($\mathbb{C}$). In order to provide decidable reasoning, $\mathbb{FDNC}$ programs are syntactically restricted to ensure that they have the *forest-shaped model* property, i.e., each stable model of an $\mathbb{FDNC}$ program can be viewed as a set of tree-shaped structures. In the basic $\mathbb{FDNC}$ programs predicates are unary and binary, and function symbols are unary. An extensions of $\mathbb{FDNC}$ with predicates of higher arities is also considered, and it is translated into the original $\mathbb{FDNC}$: higher-arity $\mathbb{FDNC}$ programs can be viewed as succinctly represented plain $\mathbb{FDNC}$ programs. The syntactic restrictions limit the use of functions symbols, and are similar to those in [CI93], although slightly more restrictive. However, they enable us to develop special techniques for handling $\mathbb{FDNC}$ programs, which are needed in order to cope with negation, disjunction, and constraints, which were not considered in [CI93].

$\mathbb{FDNC}$ is an expressive language that allows, e.g., to encode action domain descriptions in transition-based action formalisms supporting incomplete states and nondeterministic action effects, like $\mathcal{C}$ [GL98], $\mathcal{K}$ [EFL$^+$04], or fragments of the situation calculus (see e.g. [LPR98] for background). The availability of function symbols allows to naturally handle arbitrarily long action sequences. $\mathbb{FDNC}$ also facilitates a polynomial and modular encoding of knowledge bases in the expressive description logic $\mathcal{ALC}$ (cf. [BCM$^+$03]) to logic programs under answer set semantics. This reveals $\mathbb{FDNC}$ as a nonmonotonic rule language that supports features of expressive ontology languages, which is important for the integration of rules and ontologies.

2. $\mathbb{BD}$ Programs. *Bidirectional programs* (or, $\mathbb{BD}$ *programs*) are, in essence, an extension of $\mathbb{FDNC}$. The differences can be explained as follows. The syntactic restrictions in $\mathbb{FDNC}$ programs ensure that an atom A can only be inferred from atoms that are structurally not more complex than A. $\mathbb{BD}$ programs are defined by relaxing this condition. For instance, the rule $A(x) \leftarrow B(f(x))$ that allows to infer $A(t)$ from $B(f(t))$ is not allowed in $\mathbb{FDNC}$, but is a legal rule in $\mathbb{BD}$ programs.

Relaxing this restriction on the direction atom inference has both positive and negative consequences. On the positive side, it significantly increases the expressiveness of the language, as is particularly evident in the context of planning or, more generally, temporal reasoning. The syntax of $\mathbb{BD}$ programs allows to naturally express statements about both the *future* and the *past*, while using $\mathbb{FDNC}$ programs one can reason about the future *or* the past, but not both at the same time. This richer syntax allows, e.g., to change the historic values of a fluent in a planning context, based on a current observation. On the negative side, it has a high computational cost and increases dramatically the complexity of reasoning.

We additionally note that $\mathbb{BD}$ programs allow to simulate expressive description logics with the so-called *inverse roles*, and also allow for powerful manipulation of tree-shaped data, like HTML or XML documents. Using a minor rewriting, $\mathbb{BD}$ programs can be viewed as an extension of DATALOG$_{nS}$ [CI93] with disjunction and negation under the answer set semantics. Another important feature of $\mathbb{BD}$ programs is the ability to impose finiteness of stable models. By adding additional rules to a $\mathbb{BD}$ program, one can filter out infinite stable models. This not possible in $\mathbb{FDNC}$ but is desirable as it allows, e.g., to filter out infinite action sequences in encodings of planning.

3. Algorithms. We develop novel algorithms for the relevant ASP reasoning tasks over $\mathbb{FDNC}$ and $\mathbb{BD}$ programs. This includes checking the existence of a stable model of a program, as well as various kinds of queries under different modes of entailment:

- cautious/brave entailment of ground atomic queries, i.e., checking if a ground atom $Q(\vec{t})$ is true in each/some stable model of a program P,
- cautious/brave entailment of existentially quantified atomic queries, i.e., checking if an existentially quantified formula $\exists\vec{x}.Q(\vec{x})$ evaluates to true in each/some stable model of a program P, and
- cautious entailment of open queries of the form $\lambda\vec{x}.Q(\vec{x})$, which consists of finding a ground atom $Q(\vec{t})$ such that the program under consideration cautiously entails $Q(\vec{t})$.

For instance, brave entailment of existential queries is especially handy when encoding planning domains. When encoding legal action sequences as stable models of a program, one can use such a query to identify a sequence of actions that leads to the planning goal. Open queries are also useful in planning and can be used to check the existence of a so-called *secure* plan for the planning domain (the application of open queries in this context is discussed in Section 3.5).

4. Complexity results and identification of fragments. We analyze the complexity of the presented algorithms, and prove matching lower bounds to obtain completeness results. We also analyze the different sources of complexity, in order to identify syntactic fragments of these languages with better computational properties. As we show, for full $\mathbb{FDNC}$ the majority of reasoning tasks are EXPTIME-complete, but under suitable restrictions reasoning is feasible in polynomial space and, in even more restricted settings, in polynomial time. Reasoning in $\mathbb{BD}$ programs is in general harder by an exponential than in $\mathbb{FDNC}$, but we also identify sublanguages that allow for reasoning in polynomial space, in the second level of the polynomial hierarchy, and in nondeterministic polynomial time.

As a result, we give a precise account of the complexity of reasoning in full $\mathbb{FDNC}$ and $\mathbb{BD}$ programs, and in the identified sublanguages (see Table 6.1 for a summary), significantly advancing the state of the art in understanding the computational complexity of answer set programming with function symbols.

5. Techniques for reasoning and finite representation of stable models. $\mathbb{FDNC}$ and $\mathbb{BD}$ programs can have infinite and infinitely many stable models, which therefore cannot be explicitly represented for reasoning purposes. Hence we consider two reasoning techniques that allow us to finitely represent (sufficient information about) the stable models of a program, in such a way that we can effectively solve the desired reasoning problems.

- For $\mathbb{FDNC}$ programs, we provide a method to finitely represent all the stable models of a given program using *knots*. Informally, *knots* are labeled trees of depth at most one that can be seen as ‘patterns’ that may occur in stable models, and which can be assembled into full stable models. Finite sets of knots that satisfy some effectively verifiable conditions can be used to reconstruct all stable models of a program, and can be used for solving all the considered reasoning tasks.

The knot technique—which is related to the mosaic technique known from modal logics [Ném86]—is interesting on its own. As we show, it can be applied for various querying tasks over infinite structures. It has many positive features, like being well-suited for offline *knowledge compilation* [CD97, DM02] to speed up on-line reasoning, and has already been successfully transferred to other knowledge representation formalisms. In particular, it has been applied for answering conjunctive queries in various description logics [EGOŠ08, OŠE08a, OŠE08b, ELOŠ09].

- Due to the higher computational complexity and increased expressiveness of $\mathbb{BD}$ programs, the knot technique does not scale to this case, and a knot-based model representation does not seem feasible. Hence for reasoning in $\mathbb{BD}$ programs we use techniques based on *automata on infinite trees*, which can also be viewed a method for finite representation of stable models. Such techniques have often been employed to reason in modal and program logics, and in other formalisms lacking finite models. However, applying them to languages with default negation and to problems that require testing minimality is challenging and requires novel approaches like the ones followed in this thesis.

1.4 Organization of this Thesis

The remainder of this document is organized as follows:

- In Chapter 2 we introduce the preliminary notions that we use in this thesis. We introduce the syntax and semantics of logic programs under the answer set semantics, and the ASP reasoning problems that we study. We present some important notions of computational complexity, and recall some results on the computational complexity of ASP. We also introduce automata on infinite trees.

- Chapter 3 presents $\mathbb{FDNC}$ programs. We define the syntax of full $\mathbb{FDNC}$ and its fragments, as well as its extension to arbitrary arities. We introduce and discuss the knot technique, which we use to provide algorithms for the main reasoning problems. Finally, we provide complexity results for the different reasoning tasks in the introduced variations of the $\mathbb{FDNC}$ language.

- Chapter 4 presents $\mathbb{BD}$ programs as class of programs designed to circumvent some of the limitations of $\mathbb{FDNC}$. We develop automata-based algorithms for disjunctive and normal $\mathbb{BD}$ programs, and by providing lower bounds show that the algorithms are worst-case optimal. We then consider *function-safety* as a restriction for $\mathbb{BD}$ programs to reduce the complexity of reasoning. For the restricted fragments we develop algorithms and characterize the complexity of reasoning.

- Chapter 5 discusses the related work. We compare $\mathbb{FDNC}$ and $\mathbb{BD}$ programs to other fragments of ASP with function symbols, and also discuss other related methods and techniques.

- In Chapter 6 we summarize and discuss the main results of this thesis. In particular, Table 6.1 summarizes the results concerning the complexity of reasoning in ASP, possibly with function symbols. We also mention a few directions for future research.

We note that Chapter 3 is based on [ŠE07, EŠ10], while Chapter 4 is a significantly extended version of [EŠ09]. The knot technique was also described in [EOŠ08], while a general discussion of the application of knots for query answering in description logics was given in [ELOŠ09]. As already noted, knot-based approaches were taken to obtain worst-case optimal complexity results for query answering in description logics in [EGOŠ08, OŠE08a, OŠE08b, ELOŠ09].

Chapter 2

Preliminaries

We introduce here the notions that will be used throughout this thesis. We start by presenting the syntax and the semantics of ASP programs. We discuss the standard reasoning tasks in ASP, like consistency testing and answering different kinds of queries. We also recall some basic notions in Complexity Theory.

2.1 Answer Set Programming

Recall that the basic idea in ASP is to use logic programs with default negation as a language for declaratively describing and solving problems. Logic programs are built from *rules*, which are a special representation of clauses in first-order logic. We refer to [Fit96] for an excellent book on first-order logic, and to [Min88] for a more extensive introduction to Logic Programming. The *default negation* connective in ASP programs has its roots in Nonmonotonic Reasoning, and is designed to deal with problems that arise in this field; e.g., modeling common sense reasoning, defeasible inference, and preferences and priority. We refer to [Bre91] for the basics of Nonmonotonic Reasoning. Default negation in ASP programs is formally treated using the *answer set semantics* which was presented in [GL91]. For a more extensive introduction to ASP the reader may refer to [Bar02] or [EIK09].

2.1.1 Syntax

We assume the following fixed, countably infinite, pairwise disjoint sets of symbols:

(i) the set $\mathbf{CS}$ of *constant symbols* (denoted $a, b, c, d, \ldots$);

(ii) the set $\mathbf{FS}$ of *function symbols* (denoted $f, g, h, \ldots$);

(iii) the set $\mathbf{PS}$ of *predicate symbols* (denoted $R, Q, E, \ldots$);

(iv) the set $\mathbf{VS}$ of *variables* (denoted $x, y, z, \ldots$).

Each function symbol $f \in \mathbf{FS}$ and each predicate symbol $R \in \mathbf{PS}$ has an associated non-negative integer, its *arity*. Then the set $\mathbf{T}$ of *terms* is inductively defined as follows:

(i) each constant $c \in \mathbf{CS}$ and each variable $x \in \mathbf{VS}$ is a term;

(ii) if $f \in \mathbf{FS}$ is function symbol with arity n and $\langle t_1, \ldots, t_n \rangle$ is a tuple of terms, then $f(t_1, \ldots, t_n)$ is also a term.

Logic programs consist of *rules*, which in turn are built from *literals*.

Definition 2.1 (Atom, literal)**.** *An* atom *is an expression of the form $R(\vec{t})$, where R is a predicate symbol with arity n and $\vec{t}$ is an n-tuple of terms. 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.*

Definition 2.2 (Rule)**.** *A* disjunctive rule *(or simply,* rule*) is an expression r of form*

$$A_1 \vee \ldots \vee A_n \leftarrow B_1, \ldots, B_k, not\ B_{k+1}, \ldots, not\ B_m \tag{2.1}$$

where $n + m > 0$, and $A_1, \ldots, A_n, B_1, \ldots, B_m$ are atoms. The atoms $A_1, \ldots, A_n$ are called the head atoms *of r, while $B_1, \ldots, B_m$ are the* body atoms *of r. We define* $\mathsf{head}(r) = \{A_1, \ldots, A_n\}$, $\mathsf{body}^+(r) = \{B_1, \ldots, B_k\}$ *and* $\mathsf{body}^-(r) = \{B_{k+1}, \ldots, B_m\}$*. In case* $\mathsf{body}^-(r) = \emptyset$*, we call r a* positive *rule, and we let* $\mathsf{body}(r) = \mathsf{body}^+(r)$*. If r has an empty body ($m = 0$), then r is a (possibly disjunctive)* fact*. If r has an empty head ($n = 0$), then r is a* constraint*.*

Definition 2.3 (Program)**.** *A* logic program *(or* program*) is a set of rules* (2.1) *above. A program P is* positive*, if all rules in P are positive. A program P is* normal*, if each rule in P has exactly one atom in the head. If a program P is positive and normal, then P is a* Horn *program. If all predicate symbols of a program are of arity* 0*, then the program is* propositional.

If a program P has no occurrence of a function symbol, then P is a DATALOG$^{\neg,\vee}$ *program. Additionally, P is:*
- *a* DATALOG$^{\neg}$ *program if P is normal,*
- *a* DATALOG$^{\vee}$ *program if P is positive,*
- *a* DATALOG *program if P is positive and normal.*

2.1.2 Semantics

The semantics of a program P is given in terms of *Herbrand interpretations*, which we define next.

Definition 2.4 (Herbrand universe, base and interpretation)**.** *The* Herbrand universe *of P, denoted $\mathcal{HU}^P$, is the set of all terms that can be built from constants and function symbols occurring in P. The* Herbrand base *of P, denoted $\mathcal{HB}^P$, is the set of all atoms that can be built from predicate symbols of P and terms in $\mathcal{HU}^P$. An* (Herbrand) interpretation *for P is an arbitrary subset of $\mathcal{HB}^P$.*

As usual, to define the semantics of a program, we consider its *grounding*.

Definition 2.5 (Grounding)**.** *A term, atom, rule, or program that does not contain any variables is called* ground. *Let P be a program. A rule r' is called a* ground instance of a rule *$r \in P$, if r' is a ground rule obtained from r by uniformly replacing each variable in r with a term in $\mathcal{HU}^P$. The* grounding *of P, denoted* $\mathsf{Ground}(P)$*, is the set of all ground instances of all rules in P.*

The satisfaction of rules and programs is defined as follows.

Definition 2.6 (Rule satisfaction, (minimal) model)**.** *An (Herbrand) interpretation I* satisfies *a ground rule r, denoted $I \models r$, if* $\mathsf{body}^+(r) \subseteq I$ *and* $\mathsf{body}^-(r) \cap I = \emptyset$ *implies* $I \cap \mathsf{head}(r) \neq \emptyset$.

An interpretation I is a model *of a program P, denoted $I \models P$, if I satisfies each rule* $r \in \mathsf{Ground}(P)$.

A model I of P is called a minimal model *of P if there exists no $J \subset I$ that is a model of P. The set of minimal models of P is denoted by $MM(P)$.*

Minimal models provide a natural semantics for positive programs. In the presence of default negation, we consider *stable models*. Intuitively, an interpretation I is a stable model of a program P if (i) I does not violate any rule in P, and (ii) I satisfies the *stability* condition, i.e., if we assume that the truth values of the negated literals in P are given by I itself, then I is exactly the set of atoms that are justified by the rules in P. To formally define stable models we use the *Gelfond-Lifschitz reduct P^I*, which is obtained from a P by incorporating the truth values of the negated literals as given by I.

Definition 2.7 (Stable model (or, answer set))**.** *Given an Herbrand interpretation I for a program P, the* Gelfond-Lifschitz reduct *of P [GL91], 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 P is denoted by $SM(P)$. A program P is* consistent*, if $SM(P) \neq \emptyset$.*

We remind here that Horn programs are always consistent and have a unique minimal model (*the least model*). Indeed, if P is a Horn program, then $\mathcal{HB}^P$ is a model of P. Uniqueness of a minimal model follows from the fact $I_1 \cap I_2$ is a model of P for any pair I_1, I_2 of models of P. We use $LM(P)$ to denote the least model of a Horn program P. If P' is a set of constraints, then $P \cup P'$ is a *Horn program with constraints*. It is immediate to see that if $P \cup P'$ admits a model, then it has the least model, again denoted $LM(P \cup P')$.

Example 2.8. *One of the classical examples in Computer Science is that of* graph 3-colorability. *The input is an undirected graph $G = (V, E)$, and the problem is to check whether each node in V can be assigned exactly one color–green, red or blue–in such a way that adjacent nodes have different colors. We view vertices in V as constants, and build the program P consisting of the fact $Edge(c, d) \leftarrow$ for each edge $(c, d) \in E$, and the following rules:*

$$\begin{aligned} Green(x) &\leftarrow not\ Blue(x), not\ Red(x) && (2.2)\\ Blue(x) &\leftarrow not\ Green(x), not\ Red(x) && (2.3)\\ Red(x) &\leftarrow not\ Blue(x), not\ Green(x) && (2.4)\\ &\leftarrow Green(x), Edge(x, y), Green(x) && (2.5)\\ &\leftarrow Blue(x), Edge(x, y), Blue(x) && (2.6)\\ &\leftarrow Red(x), Edge(x, y), Red(x) && (2.7) \end{aligned}$$

Each stable model $I \in SM(P)$ corresponds to a possible coloring of G and vice versa. For instance, if $G = \langle\{a, b, c\}, \{(a, b), (b, c)(c, a)\}\rangle$, then G has 6 legal coloring and P has 6 stable models. One of them is

$$I = \{Edge(a, b), Edge(b, c), Edge(c, a), Green(a), Blue(b), Red(c)\}.$$

2.1.3 Reasoning Tasks

Besides checking program consistency, in this thesis we consider various queries over programs: brave and cautious entailment of ground and existential queries, and cautious entailment of open queries. We define them next.

Definition 2.9 (queries, brave/cautious entailment)**.** *A* ground (atomic) query *is any ground atom. An* (n-ary) existential (atomic) query *is an expression $\exists\vec{x}.Q(\vec{x})$, where $\vec{x}$ is an n-tuple of variables and Q is an n-ary predicate symbol. An* open query *is an expression $\lambda\vec{x}.Q(\vec{x})$, where $\vec{x}$ is an n-tuple of variables and Q is an n-ary predicate symbol.*

For a program P, we define the following:

- P bravely entails a ground query A, *in symbols $P \models_b A$, if there exists a stable model $I \in SM(P)$ such that $A \in I$.*
- P bravely entails an n-ary existential query $\exists\vec{x}.Q(\vec{x})$, *in symbols, $P \models_b \exists\vec{x}.Q(\vec{x})$, if there exists a stable model $I \in SM(P)$ and an n-tuple $\vec{t}$ of ground terms such that $Q(\vec{t}) \in I$; the tuple $\vec{t}$ is called an* answer *for the query.*
- P cautiously entails a ground query A, *in symbols $P \models_c A$, if $A \in I$ for all $I \in SM(P)$.*

- P cautiously entails an n-ary existential query $\exists\vec{x}.Q(\vec{x})$, *in symbols,* $P \models_c \exists\vec{x}.Q(\vec{x})$*, if for each stable model* $I \in SM(P)$*, there exists an* n*-tuple* $\vec{t}$ *of ground terms such that* $Q(\vec{t}) \in I$.

- P cautiously entails an n-ary open query $\lambda\vec{x}.Q(\vec{x})$ *(in symbols,* $P \models_c \lambda\vec{x}.Q(\vec{x})$*) if there exists an* n*-tuple* $\vec{t}$ *of ground terms such that* $P \models_c Q(\vec{t})$.

Note that the cautious entailment of open and existential queries is different: $\lambda\vec{x}$ 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* (or, *secure*) *plans*, i.e., sequences of actions whose execution leads to the goal, regardless of possibly incomplete knowledge about the initial state and/or nondeterministic action effects (we discuss this in Section 3.5).

The reasoning problems we consider are summarized as follows:

- **Program consistency:** Given a program P, decide whether P has some stable model.
- **Brave entailment of ground queries:** Given a program P and a ground query A, decide whether $P \models_b A$.
- **Brave entailment of existential queries:** Given a program P and an existential query $\exists\vec{x}.Q(\vec{x})$, decide whether $P \models_b \exists\vec{x}.Q(\vec{x})$.
- **Cautious entailment of ground queries:** Given a program P and a ground query A, decide whether $P \models_c A$.
- **Cautious entailment of existential queries:** Given a program P and an existential query $\exists\vec{x}.Q(\vec{x})$, decide whether $P \models_c \exists\vec{x}.Q(\vec{x})$.
- **Cautious entailment of open queries:** Given a program P and an open query $\lambda\vec{x}.Q(\vec{x})$, decide whether $P \models_c \lambda\vec{x}.Q(\vec{x})$.

Example 2.10. *Consider the program* P *consisting of the following rules:*

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

P *has two stable models* $I_1 = \{D(a), B(f(a)), C(f(a))\}$ *and* $I_2 = \{D(a), A(a), C(a)\}$. *This is because* I_1 *and* I_2 *are the minimal models of* P^{I_1} *and* P^{I_2}*, respectively, where*

$$\begin{array}{rll} P^{I_1} & = & \{D(a) \leftarrow, \\ & & B(f^{i+1}(a)) \leftarrow D(f^i(a)), \\ & & A(f^{i+1}(a)) \leftarrow D(f^{i+1}(a)) \\ & & C(f^i(a)) \leftarrow A(f^i(a)), \\ & & C(f^i(a)) \leftarrow B(f^i(a)) \mid i \geq 0\} \end{array} \qquad \begin{array}{rll} P^{I_2} & = & \{D(a) \leftarrow, \\ & & B(f^{i+2}(a)) \leftarrow D(f^{i+1}(a)), \\ & & A(f^i(a)) \leftarrow D(f^i(a)) \\ & & C(f^i(x)) \leftarrow A(f^i(x)), \\ & & C(f^i(x)) \leftarrow B(f^i(x)) \mid i \geq 0\} \end{array}$$

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

2.2 Computational Complexity

In this section we introduce some basic notions in Complexity Theory that will be relevant throughout the thesis. The material here is based on the existing literature; in particular, we follow the presentation in [DEGV01] and in the excellent book [Pap94], to which we refer the reader for a more extensive exposition.

2.2.1 Turing Machines

We start by defining *Turing machines*, which form the basic model of computation in Complexity Theory. This has a good reason: by the widely accepted Church-Turing thesis, *any* decidable problem can be solved by a Turing machine.

Deterministic Turing Machines

First we define the simplest model of Turing machines.

Definition 2.11 (DTM). *A* Deterministic Turing Machine (DTM) *is given by a tuple*

$$M = (Q, \Sigma, q_0, \delta),$$

where Q is a set of states*, Σ is an* alphabet*, $q_0 \in Q$ is the* initial state*, and*

$$\delta : Q \times \Sigma \to Q \times \Sigma \times \{+1, 0, -1\}$$

is the transition function *(or* program*). Furthermore, Q contains the* accepting state *q_{accept} and the* rejecting state *q_{reject}, while Σ contains* the blank symbol $\sqcup$.

Intuitively, a DTM $M = (Q, \Sigma, q_0, \delta)$ works as follows. An *input* to M is simply a string I of symbols that is written on a *tape*, which consists of *cells* each storing one symbol. M has a *read/write (R/W) head* that can move along the tape, reading and modifying the contents of the cell it is currently on. At each time instant, M is in some state $q \in Q$, the tape contains some string $c_1 \cdots c_n$, and the head is positioned at some cell $p \leq n$. Such an instantaneous description is called a *configuration*, and is described by the triple $(q, c_1 \cdots c_{p-1}, c_p \cdots c_n)$. A run of M starts in the initial state q_0 and with the head over the first symbol of I. Then it executes the program δ. In particular, if the current configuration is (q, w, u) and the first symbol of u is d, and $\delta(q, d) = (q', d', D)$, it overwrites d with d', changes its state to q' and based on D moves the R/W head: -1

means one step to the left, $+1$ means one step to the right, while 0 means staying in the current position. If M eventually reaches the state q_{accept}, then, intuitively, the answer to the "question" I is "yes". If it reaches q_{reject}, the answer is "no".

We next state these intuitions more formally:

Definition 2.12 (Configuration, yields)**.** *Assume a DTM $M = (Q, \Sigma, q_0, \delta)$. A* configuration *for M is a tuple (q, w, u), where $q \in Q$ and $w, u \in \Sigma^+$. Assume a configuration $C = (q, w{\cdot}c, d{\cdot}u)$, where $w, u \in \Sigma^*$ and $c, d \in \Sigma$, and suppose $\delta(q, d) = (q', d', D)$. Then C* yields *the following configuration C':*

(i) if $D = 0$, then $C' = (q', w{\cdot}c, d'{\cdot}u)$;

(ii) if $D = +1$, then $C' = (q', w{\cdot}c{\cdot}d', u')$, where $u' = u$ if $u \neq \epsilon$, and $u' = \text{␣}$ otherwise;

(iii) if $D = -1$, then $C' = (q', w', c{\cdot}d'{\cdot}u)$, where $w' = w$ if $w \neq \epsilon$, and $w' = \text{␣}$ otherwise;

We can now formally define the computation of a DTM given an input word.

Definition 2.13 (Computation, accepting/rejecting a word)**.** *Let $M = (Q, \Sigma, q_0, \delta)$ be a DTM and $w \in (\Sigma \setminus \{\text{␣}\})^*$ be word. The* computation *of M* on *w is the (possibly infinite) sequence $C_0, C_1, C_2 \ldots$ of configurations of M such that:*

(i) $C_0 = (q_0, \text{␣}, w)$ (recall that q_0 is the initial state);

(ii) for each $i > 0$, C_{i-1} yields C_i;

(iii) for any $i > 0$, if $C_i = (q', w', u')$ is such that $q' \in \{q_{accept}, q_{reject}\}$, then C_i is the last element in the sequence.

We say M accepts *(resp.,* rejects*) w if the computation of M on w is finite and the state in the last configuration is q_{accept} (resp., q_{reject}).*

Turing machines recognize languages, i.e., sets of finite words over some alphabet.

Definition 2.14 (Language, accepting/deciding a language)**.** *Assume an alphabet Σ with $\text{␣} \notin \Sigma$. A* language *over Σ is any collection $L \subseteq \Sigma^*$. We say a DTM $M = (Q, \Sigma \cup \{\text{␣}\}, q_0, \delta)$* accepts *$L$ if M accepts every word in L. If L is a language accepted by some M, then L is called* recursively enumerable*. Note that given a word $w \notin L$ the machine M may reject w or may run forever.*

We say a DTM M decides *L if M accepts every word $w \in L$ and rejects every word $w \notin L$. If for a language L there exists a machine M such that M decides L, then L is called* recursive *(or* decidable*).*

Note that any recursive language is also recursively enumerable.

Decision problems can be viewed as languages: using a suitable encoding, finite mathematical objects, like finite graphs, tables, lists, and others, can be represented as words. Thus the problem of deciding whether an object has a desired property P (e.g., a graph is 3-colorable) is equivalent to deciding whether the word representation of the object is in the language consisting of all words that encode objects with the property P.

Alternating Turing Machines

In this thesis we also use *Alternating Turing Machines*, which were introduced in [CKS81] as a generalization of DTMs. This model of computation is not more powerful in terms of computability, i.e., alternating machines accept exactly the recursively enumerable languages, but it will be useful for some algorithms and complexity characterizations.

Definition 2.15 (ATM). *An* Alternating Turing Machine (ATM) *is given by a tuple*

$$M = (Q_\exists, Q_\forall, \Sigma, q_0, \delta),$$

where $Q_\exists$ is a set of existential *states, $Q_\forall$ is a set of* universal *states, Σ is an* alphabet *containing the blank symbol ␣, $q_0 \in Q_\exists \cup Q_\forall$ is the* initial *state, and*

$$\delta \subseteq Q \times \Sigma \times Q \times \Sigma \times \{+1, 0, -1\}$$

is a transition relation, *where $Q = Q_\exists \cup Q_\forall$. $Q_\exists$ contains the* accepting state q_{accept} *and the* rejecting state q_{reject}. *It is assumed that the transition is not defined for accepting and the rejecting state. In case $Q_\forall = \emptyset$, M is a* Nondeterministic Turing Machine (NTM).

We now generalize the notions of acceptance to ATMs. Recall that in DTMs the configuration C' that follows a configuration C is uniquely determined by C and the transition function. Instead, depending on the type of a state q and a symbol d under the R/W head of an ATM, the successive configurations depend on the set $S = \{(q', d', D) \mid (q, d, q', d', D) \in \delta\}$. In case q is an existential state, the machine proceeds to a configuration that results by nondeterministically one triple $(q', d', D) \in S$. On the other hand, if q is universal, then the machine moves in parallel to all configurations that result from triples in S.

Definition 2.16 (Configuration, yields). *Assume an ATM $M = (Q_\exists, Q_\forall, \Sigma, q_0, \delta)$. A* configuration *for M is a tuple (q, w, u), where $q \in Q$ and $w, u \in \Sigma^+$.*

Assume a configuration $C = (q, w{\cdot}c, d{\cdot}u)$. If $(q, d, q', d', D) \in \delta$, then C yields *the following configuration C':*

(i) if $D = 0$, then $C' = (q', w{\cdot}c, d'{\cdot}u)$;

(ii) *if $D = +1$, then $C' = (q', w{\cdot}c{\cdot}d', u')$, where $u' = u$ if $u \neq \epsilon$, and $u' = ␣$ otherwise;*

(iii) *if $D = -1$, then $C' = (q', w', c{\cdot}d'{\cdot}u)$, where $w' = w$ if $w \neq \epsilon$, and $w' = ␣$ otherwise;*

Recall that computations of DTMs are sequences of configurations. In the case of ATMs, this generalizes to *trees* of configurations.

Definition 2.17 (Computation, accepting/rejecting a word)**.** *Let $w \in (\Sigma \setminus \{␣\})^*$ be a word. A* computation of M on w *is a (possibly infinite) tree $T = (V, E)$ where vertices in V are configurations of M and the child relation E is defined as follows:*

(i) *The root of T is $(q_0, ␣, w)$;*

(ii) *If $C = (q, u, u')$ is a node in V and $q \in Q_\exists$, then C has one child C', and C yields C'.*

(iii) *If $C = (q, u, u')$ is a node in V and $q \in Q_\forall$, then the set of children of C is $\{C' \mid C$ yields $C'\}$.*

(iv) *If $C = (q, u, u')$ is a node in V and $q \in \{q_{accept}, q_{reject}\}$, then C has no children.*

We say M accepts *w if there exists a computation of M on w where all leaves have q_{accept} as a state. We say M* rejects *w if M does not accept w and all computations of M on w are finite.*

2.2.2 Complexity Classes

Informally, a *complexity class* is a collection of problems that can be solved within a certain limit on resources, like *time* or *space*. We define these notions next.

Assume a function $f : \mathbb{N} \to \mathbb{N}$. Given a terminating DTM M, we say M *operates in time* $f(n)$ if for any input word w, M accepts or rejects w in at most $f(|w|)$ steps, i.e., the computation of M on w is at most $f(|w|)$ long. We say M *operates in space* $f(n)$ if the computation of M on w does not use more than $f(|w|)$ tape cells, i.e., for each configuration (q, u, u') in the computation, $|u| + |u'| \leq f(|w|)$.[1]

The above notions are easily extended to alternating machines. A terminating ATM M *operates in time* $f(n)$ if for any input w, all computation trees of M on w have depth at most $f(|w|)$, i.e., each branch has at most $f(|w|)$ configurations. We say M *operates in space* $f(n)$ if, for every input word w and every computation of M on w, each configuration does not use more than $f(|w|)$ tape cells.

[1]Observe that if $f(n)$ is sublinear function, then M cannot fully read its input. As usual in this case, we assume that the input is written on a *read-only* tape and the machine has an additional work tape where it is allowed to modify the content of $f(n)$ tape cells (see [Pap94] for a definition of *multiple* tape Turing machines).

$$\begin{aligned}
\mathrm{P} &= \bigcup_{k>0} \mathrm{DT{\scriptsize IME}}(n^k),\\
\mathrm{NP} &= \bigcup_{k>0} \mathrm{NT{\scriptsize IME}}(n^k),\\
\mathrm{AP} &= \bigcup_{k>0} \mathrm{AT{\scriptsize IME}}(n^k),\\
\mathrm{PS{\scriptsize PACE}} &= \bigcup_{k>0} \mathrm{DS{\scriptsize PACE}}(n^k),\\
\mathrm{NPS{\scriptsize PACE}} &= \bigcup_{k>0} \mathrm{NS{\scriptsize PACE}}(n^k),\\
\mathrm{APS{\scriptsize PACE}} &= \bigcup_{k>0} \mathrm{AS{\scriptsize PACE}}(n^k).
\end{aligned}$$

Figure 2.1: Defining main complexity classes

To introduce the complexity classes, we recall here the “big Oh” notation. Assume two functions $f : \mathbb{N} \to \mathbb{N}$ and $g : \mathbb{N} \to \mathbb{N}$. We write $f(n) = \mathcal{O}(g(n))$ if there exist integers $c, n_0 > 0$ such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.

We can now collect the languages that can be decided in bounded time or space, using the different Turing machine models above. Let DTIME$(f(n))$ (resp., NTIME$(f(n))$ and ATIME$(f(n))$) be the set of all languages L that can be decided by a DTM (resp., a NTM and an ATM) that operates in time $\mathcal{O}(f(n))$. Similarly, we use DSPACE$(f(n))$ (resp., NSPACE$(f(n))$ and ASPACE$(f(n))$) to denote the set of all languages L that can be decided by a DTM (resp., a NTM and an ATM) that operates in space $\mathcal{O}(f(n))$.

One of the most important complexity classes is P, which is the set of all languages that can be decided by a DTM in polynomial time. The classes NP and AP are defined analogously using NTMs and ATMs. The classes PSPACE, NPSPACE and APSPACE are defined by putting the bound on the space used by the machine. This is more formally defined in Figure 2.1.

We also need complexity classes to account for problems solvable in logarithmic space. We define the following:

$$\begin{aligned}
\mathrm{L} &= \mathrm{DS{\scriptsize PACE}}(\log n),\\
\mathrm{NL} &= \mathrm{NS{\scriptsize PACE}}(\log n),\\
\mathrm{AL} &= \mathrm{AS{\scriptsize PACE}}(\log n).
\end{aligned}$$

For problems solvable deterministically in exponential time, we define the class EXPTIME. More formally, EXPTIME $= \bigcup_{k>0} \mathrm{DT{\scriptsize IME}}(2^{n^k})$. Similarly, EXPSPACE =

$\bigcup_{k>0} \textsc{DSpace}(2^{n^k})$ is the set of languages that can be decided in exponential space using a DTM. We also define the classes for towers of exponents:

$$2\textsc{ExpTime} = \bigcup_{k>0} \textsc{DTime}(2^{2^{n^k}}), \quad 3\textsc{ExpTime} = \bigcup_{k>0} \textsc{DTime}(2^{2^{2^{n^k}}}), \ldots$$

$$2\textsc{ExpSpace} = \bigcup_{k>0} \textsc{DSpace}(2^{2^{n^k}}), \quad 3\textsc{ExpSpace} = \bigcup_{k>0} \textsc{DSpace}(2^{2^{2^{n^k}}}), \ldots$$

The exponential time and space classes for NTMs and ATMs are defined analogously, and are indicated by an additional "N" (e.g., NExpTime, 2NExpTime, 2NExpSpace) and "A" (e.g., AExpTime, 2AExpTime, 2AExpSpace).

Given a language L over Σ, $\overline{L}$ denotes its complement, i.e., $\overline{L} = \Sigma^* \setminus L$. If $\mathcal{C}$ is a complexity class, then co-$\mathcal{C} = \{\overline{L} \mid L \in \mathcal{C}\}$. The last notion that we need is that of the *polynomial hierarchy*, which is defined in terms of Turing machines with *oracles*. A Turing machine *with an oracle for a language* A (usually denoted M^A) is a standard Turing machine, but additionally there is a write-only tape on which the machine can write a *query* string, and it has three special states q_{query}, $q_{\in}$ and $q_{\notin}$ for querying the oracle. After the machine writes a query, it changes its state to q_{query}. In the successive configuration, the query tape is empty and the state of the machine is changed to $q_{\in}$ or $q_{\notin}$ depending on whether the query string is in the language A or not. Intuitively, the answer to the query is given by the oracle that decides A. For a complexity class $\mathcal{C}$, the class P$^{\mathcal{C}}$ is the set of all languages L for which there exists a language $A \in \mathcal{C}$ and a DTM M^A such that M^A decides L in polynomial time. The class NP$^{\mathcal{C}}$ is defined analogously. Then the polynomial hierarchy PH is defined as follows:

$$\begin{array}{rcl}
\Delta_0^p &=& \Sigma_0^p = \Pi_0^p = \mathrm{P}, \\
\Delta_{i+1}^p &=& \mathrm{P}^{\Sigma_i^p}, \\
\Sigma_{i+1}^p &=& \mathrm{NP}^{\Sigma_i^p}, \\
\Pi_{i+1}^p &=& \text{co-}\Sigma_{i+1}^p, \\
\mathrm{PH} &=& \bigcup_{k>0} \Sigma_k^p.
\end{array}$$

We finally recall some inclusions between the complexity classes introduced above:

$$\mathrm{L} \subseteq \mathrm{NL} \subseteq \mathrm{P} \overset{1}{=} \mathrm{AL} \subseteq \mathrm{NP} \subseteq \mathrm{PH} \subseteq \textsc{PSpace} \overset{2}{=} \textsc{NPSpace} \subseteq \textsc{ExpTime},$$

$$\textsc{ExpTime} \overset{1}{=} \textsc{APSpace} \subseteq \textsc{NExpTime} \subseteq \textsc{ExpSpace}.$$

The equality $\overset{1}{=}$ is due to [CKS81], while $\overset{2}{=}$ is a consequence of Savitch's Theorem [Sav70].

2.2.3 Reductions and Completeness

We provide here the standard notion of *completeness* for a complexity class. Assume a terminating DTM M. Given an input word w for M, by $f^M(w)$ we denote the word that is written on the tape in the last configuration of the computation of M on w. The notion of a *reduction* from a language L to another language L' allows to view L' as *at least as hard as* L.

Definition 2.18 (Reductions)**.** *We say a language L is reducible to L' if there exists a terminating DTM M such that, for all words w, $w \in L$ iff $f^M(w) \in L'$.*

If, in addition, M terminates in polynomial time, then we say that L is reducible to L' in polynomial time. If M operates in logarithmic space, then L is reducible to L' in logarithmic space.

Completeness for a complexity class is defined as follows:

Definition 2.19 (Hardness, completeness)**.** *Let $\mathcal{C}$ be a complexity class. We say that a language L is $\mathcal{C}$-*hard *if any language $L' \in \mathcal{C}$ is reducible to L. If $L \in \mathcal{C}$ and L is $\mathcal{C}$-hard, then L is $\mathcal{C}$-*complete.

As it is common in Complexity Theory, when proving that a language L is $\mathcal{C}$-hard for some complexity class $\mathcal{C}$, we require the following. If $\mathcal{C}$ contains NP or co-NP, then every language $L' \in \mathcal{C}$ must be reducible to L in polynomial time. If $\mathcal{C}$ is P, then every language $L' \in \mathcal{C}$ must be reducible to L in logarithmic space.

2.3 Complexity of Answer Set Programming

The complexity of logic programs under the answer set semantics is quite well understood in the propositional and in the DATALOG case. We refer the reader to [DEGV01] for a detailed exposition of complexity results in these settings; some of the results that we use in this thesis are summarized in Table 2.1.

Additionally, Table 2.1 has entries for the general case where function symbols are fully supported, and the *finitely recursive programs* [Bon04, BBC09] that will be used as a reference point in the following chapters. The complexity of reasoning in the presence of function symbols has been analyzed in [MNR94, MR03, MNR92]. For instance, in [MNR92] the authors show that the existence of a stable model and cautious inference in a logic program with function symbols are highly undecidable. In particular, the problems lie at the second level of the analytical hierarchy and are Σ^1_1-complete and Π^1_1-complete, respectively (see Table 2.1). As shown in [EG97], these results carry over to the disjunctive case also.

Finitely recursive programs form an expressive fragment of general normal programs with function symbols. The complexity of finitely recursive programs is lower

Languages	Consistency	$P \models_b A(\vec{t})$	$P \models_b \exists \vec{x}.A(\vec{x})$	$P \models_c A(\vec{t})$	$P \models_c \exists \vec{x}.A(\vec{x})$
propositional normal programs	NP	NP	NP	co-NP	co-NP
propositional disjunctive programs	Σ_2^P	Σ_2^P	Σ_2^P	Π_2^P	Π_2^P
propositional normal positive programs	trivial	P	P	P	P
propositional disjunctive positive programs	NP	Σ_2^P	Σ_2^P	co-NP	co-NP
DATALOG	trivial	EXPTIME	EXPTIME	EXPTIME	EXPTIME
DATALOG$^{\neg}$	NEXPTIME	NEXPTIME	NEXPTIME	co-NEXPTIME	co-NEXPTIME
DATALOG$^{\neg,\vee}$	NEXPTIME$^{\text{NP}}$	NEXPTIME$^{\text{NP}}$	NEXPTIME$^{\text{NP}}$	co-NEXPTIME$^{\text{NP}}$	co-NEXPTIME$^{\text{NP}}$
finitely recursive programs	co-R.E.	co-R.E.	co-R.E.	R.E.	R.E.
general case (normal and disjunctive)	Σ_1^1	Σ_1^1	Σ_1^1	Π_1^1	Π_1^1

Table 2.1: Complexity of Answer Set Programming (completeness results)

than in the general case, and is, in fact, in line with the complexity of inference in first-order logic. The fragment is defined by restricting atom *dependencies*. Assume a normal program P. We say $A \in \mathcal{HB}_P$ *directly depends* on $B \in \mathcal{HB}_P$ if there is rule $r \in \mathsf{Ground}(P)$ such that A is the head of r and B occurs in the body of r. The *depends* relations is the reflexive transitive closure of *depends directly*. The programs P is *finitely recursive* if each atom $A \in \mathcal{HB}_P$ depends on finitely many atoms in $\mathcal{HB}_P$. For example $P_1 = \{A(c) \leftarrow; A(f(x)) \leftarrow A(x)\}$ is finitely recursive, while $P_2 = \{A(c) \leftarrow; A(x) \leftarrow A(f(x))\}$ is not finitely recursive. This is because in the latter program, the atom $A(c)$ depends of $A(c), A(f(c)), A(f(f(c))), \ldots$, i.e., on infinitely many atoms. In contrast, in the former program each atom $A(t)$ depends only on atoms $A(t')$ where t' is a subterm of t of smaller depth, and there are only finitely many such atoms.

2.4 Automata over Infinite Trees

In the second part of the thesis we will employ *finite state automata over infinite trees* as a tool for obtaining worst-case optimal complexity results for $\mathbb{BD}$ programs. Automata over infinite trees are a generalization of standard finite state automata over finite words.

To deal with infinite words and trees, the former automata are equipped with *acceptance conditions* which prescribe which infinite words or trees the automaton accepts. For example, the simplest kind of acceptance condition is called *Büchi condition,* and given by a set of accepting states. An automaton accepts an infinite word if some accepting state is visited infinitely often. We use a more complex kind of condition called *parity condition,* that will be defined below. The research in this field was spurred by the seminal works in [Büc60, Rab69] on the decidability of monadic second-order logic over infinite words and trees. Nowadays automata over infinite trees are widely applied in computer aided verification, modal logics, description logics, program and fixed-point logics (cf. [EJ91, Var98, VW86, KSV02, BLMV08, CDG03]) or, simply, in logics that enjoy the tree-shaped model property [Tho90]. We refer the reader to [Tho90] for an excellent introduction to the topic.

We define here *2-way alternating tree automata* following closely the presentation in [Var98].

Definition 2.20. *(Infinite trees) An* infinite tree T *is any prefix-closed set of words over the positive integers (denoted by* $\mathbb{N}$*), i.e.,* $T \subseteq \mathbb{N}^*$ *such that* $x \cdot c \in T$*, where* $x \in \mathbb{N}^*$ *and* $c \in \mathbb{N}$*, implies* $x \in T$*.* T *is* full *if, additionally,* $x \cdot c' \in T$ *for all* $0 < c' < c$*. Each element* $x \in T$ *is a* node *of* T*, where* ϵ *(the empty word) is the root of* T*. The nodes* $x \cdot c \in T$*, where* $c \in \mathbb{N}$*, are the* successors *of* x*. By convention,* $x \cdot 0 = x$ *and* $(x \cdot i) \cdot (-1) = x$ *(note that* $\epsilon \cdot (-1)$ *is undefined).* T *is* k-ary *if it is full and each node in* T *has* k *successors.*

An infinite path *in* T *is a prefix-closed node set* $p \subseteq T$ *such that for every* $i \geq 0$ *there is a unique* $x \in p$ *such that* $|x| = i$*. A* labeled tree *over an alphabet* Σ *is a tuple* $(T, \mathcal{L})$*, where* $\mathcal{L} : T \to \Sigma$*, i.e., a tree where the nodes are labeled with symbols from* Σ*.*

For a set V of propositions, let $B(V)$ be the set of all Boolean formulas that can be built from $V \cup \{\mathbf{true}, \mathbf{false}\}$ using $\vee$ and $\wedge$ as connectives. We say that $I \subseteq V$ *satisfies* $\varphi \in B(V)$, if assigning $\mathbf{true}$ to each $p \in I$ and $\mathbf{false}$ to each $p \in V \backslash I$ makes φ true.

Definition 2.21. *(2ATAs) Let* $[k] = \{-1, 0, 1, \ldots, k\}$*. A* two-way alternating tree automaton (2ATA) over infinite k-ary trees *is a tuple*

$$A = \langle \Sigma, Q, \delta, q_0, F \rangle,$$

where Σ *is an input alphabet,* Q *is a finite set of states,* $\delta : Q \times \Sigma \to B([k] \times Q)$ *is a transition function,* $q_0 \in Q$ *is an initial state, and* F *specifies an acceptance condition. We consider here* parity *acceptance, which is given by a tuple* $F = (G_1, G_2, \ldots, G_m)$ *where* $G_1 \subseteq G_2 \subseteq \ldots \subseteq G_m$ *and* $G_m = Q$*.*

Informally, a *run* of a 2ATA A over a labeled tree $(T, \mathcal{L})$ is a tree T_r where each node $n \in T_r$ is labeled with $(x, q) \in T \times Q$. Here n describes a copy of A that is in state q and reads the node $x \in T$, and the labeling of its successor nodes must obey the transition function.

Definition 2.22. *(Runs) Formally, a run (T_r, r) is labeled tree over $\Sigma_r = T \times Q$, which satisfies the following:*

1. *$\epsilon \in T_r$ and $r(\epsilon) = (\epsilon, q_0)$.*
2. *For each $y \in T_r$, with $r(y) = (x, q)$ and $\delta(q, \mathcal{L}(x)) = \varphi$, there is a set*
$$S = \{(c_1, q_1), \ldots, (c_n, q_n)\} \subseteq [k] \times Q$$
such that (i) S satisfies φ, and (ii) for all $1 \leq i \leq n$, we have that $y \cdot i \in T_r$, $x \cdot c_i$ is defined, and $r(y \cdot i) = (x \cdot c_i, q_i)$.

A run (T_r, r) is accepting, if every infinite path $p \subseteq T_r$ satisfies F as follows. Let $inf(p)$ be the set of states $q \in Q$ that occur infinitely often in p. Then p satisfies F, if an even i exists for which $inf(p) \cap G_i \neq \emptyset$ and $inf(p) \cap G_{i-1} = \emptyset$. An automaton accepts a labeled tree, if there is a run that accepts it. By $L(A)$ we denote the set of trees that A accepts.

By restricting transitions of 2ATAs we can obtain other important kinds of automata. In case an automaton is over unary trees, then it is a *word* automaton. Assume $A = \langle \Sigma, Q, \delta, q_0, F \rangle$ is a 2ATA over k-ary trees. We say A is a *nondeterministic one-way tree automaton (1NTA)* if for each $q \in Q$ and $\sigma \in \Sigma$, $\delta(q, \sigma)$ is of the form:

$$\delta(q, \sigma) = \big((1, q_0^1) \wedge \ldots \wedge (k, q_0^k)\big) \vee \ldots \vee \big((1, q_n^1) \wedge \ldots \wedge (k, q_n^k)\big).$$

Intuitively, 1NTAs only move down the tree and with each guess the automaton proceeds with exactly one state for each child node. Importantly, 2ATAs can be translated in 1NTAs while preserving the recognized language.

Theorem 2.23 ([Var98]). *Let A be a two-way alternating tree automaton. Then there is a nondeterministic parity tree automaton A^n such that $L(A) = L(A^n)$. The number of states in A^n is exponential in the number of states in A, but the size of the acceptance condition of A^n is linear in the size of the acceptance condition of A.*

Given the above translation, we can use existing algorithms for testing nonemptiness of 1NTAs (e.g., in [EJ88, KV98]) to decide nonemptiness of 2ATAs. We recall that nonemptiness of a given 1NTA A over k-ary trees can be decided in time $\mathcal{O}(c) + (m \cdot n)^{\mathcal{O}(n \cdot k)}$, where c is the size of the alphabet, m is the number of states, and n is the index of the parity condition in A.

By combining the above two results we get the following:

Theorem 2.24. *Testing nonemptiness of a 2ATA A over k-ary trees is feasible in $\mathcal{O}(c) + n^{\mathcal{O}(n \cdot m \cdot k)}$, where c is the size of the alphabet, m is the number of states, and n is the index of the parity condition in A.*

Chapter 3

$\mathbb{FDNC}$ *Programs*

In this chapter, we introduce the class $\mathbb{FDNC}$ of disjunctive logic programs with function symbols and negation under the stable model semantics. In order to provide decidable reasoning services, $\mathbb{FDNC}$ programs are syntactically restricted to ensure that their stable models have the shape of a forest, i.e., a collection of tree-shaped structures. In the first stage we consider programs 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 3.1. To accommodate predicates of higher arity, an extension of $\mathbb{FDNC}$ to higher-arity predicates is conceived in Section 3.6.

We study several reasoning problems for $\mathbb{FDNC}$, including deciding the consistency of a program (i.e., existence of a stable model), brave and cautious entailment of ground and existential queries, as well as cautious entailment of open queries (see Section 2.1 for more details). We also consider the natural restrictions of $\mathbb{FDNC}$ programs 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 $\mathbb{F}$ in this family is a subclass of Horn programs that is (apart from minor deviations) a fragment of DATALOG_{nS} in [CI93].

For the considered reasoning tasks we develop algorithms and characterize their computational complexity over the whole program family from $\mathbb{F}$ 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 nonmonotonic 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, other complexity results range from polynomial time, over co-NP, Σ_2^P, PSPACE and EXPTIME, up to EXPSPACE. In particular, for $\mathbb{F}$ programs (which are a class of Horn programs), entailment of ground atoms can be decided in polynomial time; note that even in the absence of function symbols, this problem is NP-hard for (full) Horn programs with binary predicates. Table 3.1 on page 46 compactly summarizes our complexity results, which are discussed in detail in Section 3.2.

$\mathbb{FDNC}$ programs can have infinite and infinitely many 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 the forest-shaped stable models of a program from *knots*, which are generic labeled trees of depth at most 1. The knot technique allows us to define elegant decisions procedures for reasoning in $\mathbb{FDNC}$ and its fragments. It may also be exploited for offline *knowledge compilation* [CD97, DM02] to speed up online reasoning, by precomputing and storing a knot representation of a logic program P. Given such a representation, multiple queries over P can be answered comparatively efficiently (some problems are solvable in polynomial time), and also model building can be supported (which is of concern in ASP): with the knots as building blocks, any relevant part of any stable model of P can be gradually constructed (leading to an infinite process in general). In general, a knot representation of a logic program is exponential in the program size; this is the common 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 [DM02]).

Notably, 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 $\mathcal{ALC}$. Thus as a further result, we obtain a polynomial time mapping of this well-known description logic (cf. [BCM+03]) to logic programs under the answer set semantics. The mapping takes advantage of a normal form of $\mathcal{ALC}$ knowledge bases and is balanced in the sense that it maps to a class of logic program whose complexity is not higher than the one of $\mathcal{ALC}$ (see Chapter 5 for a discussion of other mappings). These results are interesting in their own right and may be exploited in other contexts, like integration of rules and ontologies.

Apart from simulating some description logics, the language of $\mathbb{FDNC}$ programs seems to be well suited for other knowledge representation problems. It can, for instance, be fruitfully exploited for reasoning about actions and planning. We recall that applicability of ASP in this area is well-known and has been explored in many works, including [DNK97, Lif99, Bar02, EFL+04, TSB07, SBTM06, STGM05, MTS07]. $\mathbb{FDNC}$ programs allow to encode action domain descriptions in such a way that arbitrarily long action sequences are handled naturally.

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 $S(x)$ for situations, and the constant $init$ for the initial situation. For the initializaiton, a fact $S(init) \leftarrow$ is added for the initial situation, and the initial state of the domain is described by facts of the form $F(init) \leftarrow$.

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. With a binary predicate Tr, we can use atoms $Tr(x, f_{A_i}(x))$ to express that a transition happened. A rule $A_1(x) \vee \cdots \vee A_n(x) \leftarrow S(x)$ may be used to select 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 using 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 nonmonotonic negation allows to conveniently state fluent inertia, i.e., the fluent value when taking an action remains the same *by default*. For a fluent F, this can be expressed using the two rules

$$\begin{array}{l} F(y) \leftarrow F(x), Tr(x, y), \textit{not } \bar{F}(y), \\ \bar{F}(y) \leftarrow \bar{F}(x), Tr(x, y), \textit{not } F(y), \end{array}$$

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 action domains from the literature, e.g., the Yale Shooting [HM87], Bomb in the Toilet, and others (cf. [EFL$^+$04]), and to solve reasoning and planning problems on them. In Section 3.5 we more concretely elaborate on an encoding of action domains in a fragment of the language $\mathcal{K}$ into $\mathbb{FDNC}$, and show in an example how query answering can be used to elegantly solve, among others, conformant planning problems in $\mathcal{K}$. The latter are EXPSPACE-complete in general, and show that $\mathbb{FDNC}$ programs offer the complexity tailored to these problems.

The remainder of this chapter is organized as follows. Section 3.1 introduces $\mathbb{FDNC}$ programs, and establishes their basic semantic properties. It also introduces the finite representation of stable models in terms of knots. Section 3.2 gives an overview and a discussion of the complexity results in this chapter, which are established in the subsequent Sections 3.3 and 3.4. In the course of this, also reasoning techniques and algorithms are developed. Section 3.5 discusses an application of $\mathbb{FDNC}$ to reasoning about actions. Section 3.6 considers an extension of $\mathbb{FDNC}$ to higher-arity programs. We conclude the chapter in Section 3.7.

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

$$
\begin{array}{lrcl}
\textit{(R1)} & A_1(x) \vee \ldots \vee A_k(x) & \leftarrow & B_0(x), B_1^{\pm}(x), \ldots, B_l^{\pm}(x) \\
\textit{(R2)} & R_1(x,y) \vee \ldots \vee R_k(x,y) & \leftarrow & P_0(x,y), P_1^{\pm}(x,y), \ldots, P_l^{\pm}(x,y) \\
\textit{(R3)} & R_1(x,f_1(x)) \vee \ldots \vee R_k(x,f_k(x)) & \leftarrow & P_0(x,g_0(x)), P_1^{\pm}(x,g_1(x)), \ldots, P_l^{\pm}(x,g_l(x)) \\
\textit{(R4)} & A_1(y) \vee \ldots \vee A_k(y) & \leftarrow & R_0(x,y), R_1^{\pm}(x,y), \ldots, R_l^{\pm}(x,y), \\
 & & & B_1^{\pm}(x), \ldots, B_m^{\pm}(x), C_1^{\pm}(y), \ldots, C_n^{\pm}(y) \\
\textit{(R5)} & A_1(f(x)) \vee \ldots \vee A_k(f(x)) & \leftarrow & R_0(x,f(x)), R_1^{\pm}(x,f(x)), \ldots, R_l^{\pm}(x,f(x)), \\
 & & & B_1^{\pm}(x), \ldots, B_m^{\pm}(x), C_1^{\pm}(f(x)), \ldots, C_n^{\pm}(f(x)) \\
\textit{(R6)} & R_1(x,f_1(x)) \vee \ldots \vee R_k(x,f_k(x)) & \leftarrow & B_0(x), B_1^{\pm}(x), \ldots, B_l^{\pm}(x) \\
\textit{(R7)} & C_1(\vec{c_1}) \vee \ldots \vee C_k(\vec{c_k}) & \leftarrow & D_1^{\pm}(\vec{b_1}), \ldots, D_l^{\pm}(\vec{b_l}),
\end{array}
$$

where $k, l, m, n \geq 0$, and each $\vec{c_i}$, $\vec{b_i}$ is a tuple of constants of arity ≤ 2. Moreover, at least one rule in the program is of type (R7). W.l.o.g., we assume that in one-variable (resp., two-variable) rules, the variable in unary atoms (resp., variable tuple in binary atoms) is always x (resp., $\langle x, y \rangle$).

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 restrictions draw their inspiration from classical first-order clauses with existential quantification restricted to positive literals, i.e., of implications $\forall\vec{x}\exists\vec{y}\alpha(\vec{x}) \rightarrow \beta(\vec{x},\vec{y})$ where $\alpha(\vec{x})$ is a conjunction and $\beta(\vec{x},\vec{y})$ is a disjunction of atoms with free variables $\vec{x}$ and $\vec{x},\vec{y}$, respectively. Of particular interest are clauses with predicate arities ≤ 2 where existential quantification is additionally restricted to one variable in binary literals. Skolemization eliminates each existentially quantified variable with a fresh unary function symbol; the Herbrand universe of a theory can then be represented by a labeled graph that has certain tree shape: terms $f(t)$ being children of a term t. The rules allow to describe unary predicates satisfied by terms (classification), and to define binary relationships between them. In particular, we can describe properties of a term t depending solely on t itself (by rules (R1)), and relations between t and another term t' depending on other existing relations (by rules (R2)). By rules (R4), we can talk about how the properties of a term t affect the properties of terms to which t is related. Rules (R6) are crucial as they allow to introduce new objects: we can state the existence of a child term $f(t)$ to which t is related. Such use of function symbols is convenient in applications, and ensures the forest-model property on which decidability and complexity proofs hinge. With rules (R3), we can describe further relations between t and a term $f(t)$ depending on other such relationships for terms $g(t)$, and with rules (R5) properties of $f(t)$, depending on relations between t and $f(t)$ and properties of t and other properties of $f(t)$. Finally, the rules (R7) allow us to express arbitrary properties of and binary relations between elementary objects (represented by constants).

We note that the rules (R5) are (non-ground) instances of (R4), and thus not strictly needed; in turn, rules (R4) can be eliminated using rules (R5) and (R7). Similarly, the rules (R2) could be equivalently replaced by rules (R3) and (R7). However, (R2) and (R5) are useful for modeling purposes and thus included.

The first body atom in the rules (R1)-(R6) ensures their safety, i.e., each variable occurs in a positive body atom. For (R1), (R3), (R5) and (R6) this could be relaxed (no positive body atom is prescribed). Such non-safe programs can be simulated by $\mathbb{FDNC}$ programs using a unary *domain predicate* and a binary *successor predicate* that holds for each term t and each pair $\langle t, f(t)\rangle$ of terms, respectively, in the Herbrand universe. Using fresh unary and binary predicates Dom and $Succ$, respectively, augment P with (i) $Dom(c) \leftarrow$ for each constant c of P, (ii) $Succ(x, f(x)) \leftarrow Dom(x)$ for each function symbol f of P, (iii) and the rule $Dom(y) \leftarrow Succ(x, y)$. Finally, add in the body of each original rule r the atom $Dom(x)$ if r is of form (R1), (R3), or (R6), and $Succ(x, f(x))$ if r is form (R5). As easily seen, the rewriting preserves stable models on the initial signature. By eliminating rules (R2) and (R4) beforehand, we could have a variant of $\mathbb{FDNC}$ without safety restrictions; the connected forest-shaped models of $\mathbb{FDNC}$ programs would change into a rudimentary form.

The structure of the rules in $\mathbb{FDNC}$ syntax, the availability of nonmonotonic 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.

Example 3.2. *As a running example, we use the $\mathbb{FDNC}$ program P in Figure 3.1. It represents the evolution of a cell, viz. growth, splitting into two cells, and death. (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)-(8) determine whether a cell is dead, young or mature. The rules (9)-(11) state the knowledge about the temperature. During growth (which takes some time), it might alter, while in the other changes (which happen quickly), it stays the same, which is expressed by inertia*

(1) $Change(x, grow(x)) \leftarrow Young(x),\, Warm(x)$
(2) $Change(x, cell_1(x)) \leftarrow Mature(x),\, Warm(x)$
(3) $Change(x, cell_2(x)) \leftarrow Mature(x),\, Warm(x)$
(4) $Change(x, die(x)) \leftarrow Cold(x), not\ Dead(x)$
(5) $Dead(die(x)) \leftarrow Change(x, die(x))$
(6) $Young(cell_1(x)) \leftarrow Change(x, cell_1(x))$
(7) $Young(cell_2(x)) \leftarrow Change(x, cell_2(x))$
(8) $Mature(grow(x)) \leftarrow Young(x), Change(x, grow(x))$
(9) $Warm(grow(x)) \vee Cold(grow(x)) \leftarrow Change(x, grow(x))$
(10) $Warm(y) \leftarrow Warm(x), Change(x, y), not\ Cold(y)$
(11) $Cold(y) \leftarrow Cold(x), Change(x, y), not\ Warm(y)$
(12) $\leftarrow Cold(x),\, Warm(x)$
(13) $Young(b) \leftarrow$
(14) $Warm(b) \leftarrow$

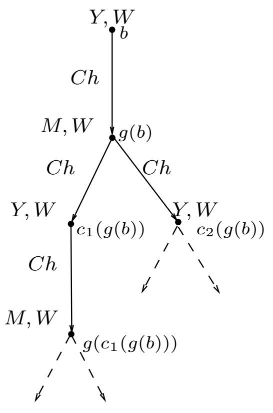


Figure 3.1: Example: Evolution of a Cell

rules (10) and (11). Finally, (13) and (14) are the initialization facts. (For brevity, we also shorten predicate symbols to $W(arm)$, $C(old)$, $Y(oung)$, $M(ature)$, $D(ead)$, 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 3.1 represents a development where the temperature does not change during the growth of b and its child. Another stable model is $I = \{\ Young(b)$, $Warm(b)$, $Change(b, grow(b))$, $Cold(grow(b))$, $Mature(grow(b))$, $Change(grow(b)$, $die(grow(b)))$, $Dead(die(grow(b)))$, $Cold(die(grow(b)))\ \}$ 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 3.2 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 P 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 P.

3.1.1 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 the nodes in the graph.

Definition 3.3. *An (Herbrand) 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 is of the form $R(c, d)$ or $R(t, f(t))$, where c, d are constants, and t is a ground 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, $R(t, f(t)) \in I$.

The "graph part" of I consists of the atoms $R(c, d)$, were c, d are constant symbols; intuitively, c and d are connected by an arc from c to d. The other binary atoms constitute a set of trees, as $f(t)$ has via $R(t, f(t))$ the term t as its uniquely determined ancestor, and the root of each such tree must be a constant symbol (i.e., a node of the graph part). The following proposition is important.

Proposition 3.4. *If H is an arbitrary interpretation of an $\mathbb{FDNC}$ program P and $J \in MM(P^H)$, then J is forest-shaped (in particular, every $J \in SM(P)$ is forest-shaped).*

Proof. The property follows directly from the structure of the rules and the minimality requirements. In particular, for (a) in Definition 3.3, note that the rules of P can have binary atoms only of the forms $R(c, d)$, $R(x, f(x))$ and $R(x, y)$. In the case of $R(x, y)$ atoms in the head (case of (R2) rules), the body atom arguments are $\langle x, y \rangle$, and hence such rules do not spoil the argument structure, i.e., they cannot introduce atoms of a shape different from the one in (a). For (b), note that the atoms of the form $A(f(t))$ can be derived only via the rules (R1), (R4) or (R5). Firing rules (R4) and (R5) requires an atom of the form $R(t, f(t))$. In the case of rules (R1), all body atoms have the same term as in the head and hence the derivation of $A(f(t))$ can be traced back to rules (R4) or (R5). Suppose H is an arbitrary interpretation for P. Assume some $J \in MM(P^H)$ contains an atom violating (a) or (b) in Definition 3.3. We can simply collect all the atoms violating (a) or (b) and remove them from J. Due to the observations above, 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 thesis 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 here can be inferred from the results in

[EG97]. The technique in [EG97] shows how the stable model semantics for 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 [MHS07] 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 3.1, we assume that P is an arbitrary $\mathbb{FDNC}$ program.

Notation 3.5. *For convenience, given a term t and a set of atoms I, we write $t \mathrel{\hat{\in}} I$, if there exists an atom in I having t as an argument.*

We next define states and atomic state sets associated with sets of atoms and programs.

Definition 3.6 (States $\mathsf{st}(I,t)$; atomic state sets $\mathsf{st}(I)$,$\mathsf{st}(P)$)**.** *A* state *of any ground term t is an arbitrary set U^t of unary atoms of form $A(t)$. For any set of atoms I and term $t \mathrel{\hat{\in}} I$, the* state of t in I *is $\mathsf{st}(I,t) = \{A(t) \mid A(t) \in I\}$. Furthermore, the* atomic state set *of I (resp., a program P) is $\mathsf{st}(I) = \{\mathsf{st}(I,c) \mid c \mathrel{\hat{\in}} I$ is a constant$\}$ (resp., $\mathsf{st}(P) = \bigcup_{I \in SM(P)} \mathsf{st}(I)$).*

Example 3.7 (Cont'd)**.** *For the above stable model I of P, we have*

$$\begin{aligned} \mathsf{st}(I,b) &= \{Young(b),\ Warm(b)\}, \\ \mathsf{st}(I,grow(b)) &= \{Cold(grow(b)),\ Mature(grow(b))\}, \text{ and} \\ \mathsf{st}(I,die(grow(b))) &= \{Dead(die(grow(b))),\ Cold(die(grow(b)))\}. \end{aligned}$$

Moreover, $\mathsf{st}(I) = \{\mathsf{st}(I,b)\}$, and as all stable models of P clearly agree on the function-free atoms, $\mathsf{st}(P) = \mathsf{st}(I)$.

We omit t from U^t if t is not of particular interest. For a one-variable rule r in $\mathbb{FDNC}$ syntax and a term t, let $r_{\downarrow t}$ denote the rule obtained by substituting the variable x in r with t. Similarly, for a two-variable r and terms s,t, let $r_{\downarrow s,t}$ denote the rule obtained by substituting x and y in r with s and t, respectively.

Definition 3.8 (Local Program $P(U^t)$)**.** *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 induce a stable model of $P(U)$. Stable models of local programs have a simple structural property, captured by the notion of *knots*.

Definition 3.9 (Knot)**.** *A* knot with 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) 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 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 3.2 shows an example of knots over the signature of the program P in Example 3.2.

It is easy to see that due to the structure of local programs, their stable models satisfy the conditions in Definition 3.9 and hence are knots. On the other hand, knots are also the structures that occur in the trees of the forest-shaped interpretations. To "extract" knots from such interpretations, the following is 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 over P.

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

Definition 3.10 (Stable Knot)**.** *Let* K *be a knot with 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 root term t and $U^t = \mathsf{st}(K,t)$ is stable w.r.t. P, and that t occurs in a forest-shaped interpretation I for P as a "leaf node", i.e., I has no atoms of form $R(t,f(t))$. If the

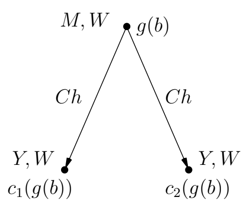
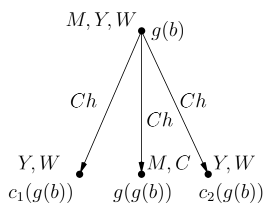
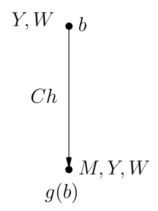
K_1	K_2	K_3
$M(g(b)), W(g(b)),$ $Ch(g(b), c_1(g(b))),$ $Y(c_1(g(b))),$ $W(c_1(g(b))),$ $Ch(g(b), c_2(g(b))),$ $Y(c_2(g(b))),$ $W(c_2(g(b)))$	$M(g(b)), Y(g(b)),$ $W(g(b))$ $Ch(g(b), c_1(g(b))),$ $Ch(g(b), c_2(g(b))),$ $Ch(g(b), g(g(b))),$ $Y(c_1(g(b))),$ $W(c_1(g(b))),$ $Y(c_2(g(b))),$ $W(c_2(g(b))),$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

states of t in I and K coincide, i.e., $\mathsf{st}(I, t) = U^t$, then intuitively K is a suitable set of atoms to give t the necessary successors in I.

Example 3.11 (Cont'd). *Consider the knots K_1, K_2 and K_3 in Figure 3.2. As easily seen, P has a stable model I in which K_1 occurs, i.e., $I \cap \mathcal{HB}_{g(b)} = K_1$; in fact, Figure 3.1 shows an example. In contrast, K_2 and K_3 do not occur in any stable model of P, as the rules of P do not force an element to satisfy both M and Y.*

The knot K_1 is stable: as easily checked, K_1 is a stable model of the local program $P(\{M(g(b)), W(g(b))\})$. While K_2 does not occur in any stable model of P, it is a stable model of $P(\{M(g(b)), Y(g(b)), W(g(b))\})$, and hence stable. Intuitively, K_2 is an eligible building block for a stable model of P only if $g(b)$ satisfies exactly W and both M and Y. The knot K_3 is not stable, since the stable models of $P(\{Y(b), W(b)\})$ 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 3.12 (Graph Program $\mathsf{gp}(P)$). *For a program P, by $\mathsf{gp}(P)$ we denote the set of all function-free rules $r \in \mathsf{Ground}(P)$.*

Example 3.13 (Cont'd). *In our running example, $\mathsf{gp}(P)$ consists of the two facts $Young(b) \leftarrow$ and $Warm(b) \leftarrow$.*

The following theorem characterizes the stable models of P. For an interpretation I, let $\mathit{ffa}(I)$ be the set of all function-free atoms in I.

Theorem 3.14. *If I is an interpretation for P, then the following are equivalent:*

(A) I is a stable model of P.

(B) I is a forest-shaped interpretation such that (i) $\mathit{ffa}(I)$ is a stable model of $\mathsf{gp}(P)$*, and (ii) for each term $t \mathrel{\hat{\in}} I$, $I \cap \mathcal{HB}_t$ is a knot that is stable w.r.t. P.*

Proof. (A) ⇒ (B). Assume $I \in \mathit{SM}(P)$. By Proposition 3.4, I is forest-shaped. First, we show that (i) holds, by exploiting the concept of *modularity* in disjunctive programs under the Answer Set semantics [EGM97] (this is closely related to *splitting sets* of [LT94] for normal programs). Let $Q = \mathsf{Ground}(P) \setminus \mathsf{gp}(P)$. Note that none of the head atoms in rules of Q occurs in rules of $\mathsf{gp}(P)$, and hence $\mathsf{gp}(P)$ is *independent* from Q. By Lemma 5.1 in [EGM97], since $I \in \mathit{SM}(\mathsf{Ground}(P))$ and $\mathsf{gp}(P)$ is independent from Q, $I \cap \mathcal{HB}^{\mathsf{gp}(P)}$ is a stable model of $\mathsf{gp}(P)$. Since $I \cap \mathcal{HB}^{\mathsf{gp}(P)} = \mathit{ffa}(I)$, the claim holds.

We similarly show that (ii) holds. Suppose $t \mathrel{\hat{\in}} I$ and $K = I \cap \mathcal{HB}_t$. As I is forest-shaped, K is a knot over the signature of P. Suppose K is not stable w.r.t. P, i.e., $K \notin \mathit{SM}(P(U))$, where $U = \mathsf{st}(K, t)$. There are two possibilities:

- $K \not\models P(U)^K$. Then some rule $r \in P(U)^K$ exists such that $\mathsf{body}(r) \subseteq K$ and $\mathsf{head}(r) \cap K = \emptyset$. As each fact $A(t) \leftarrow$ is in $P(U)$ iff $A(t) \in K$, r is not of this form. Thus $r \in P_t$, where P_t results from $P(U)$ by removing the facts. By the construction of local programs, $P_t \subseteq \mathsf{Ground}(P)$. As $K = I \cap \mathcal{HB}_t$, 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 $\mathsf{head}(r) \cap I = \emptyset$. Hence, $I \notin \mathit{SM}(P)$.

- $K \models P(U)^K$, but is not minimal, i.e., some $H \subset K$ fulfills $H \models P(U)^K$. Let $M = H \cup (I \setminus K)$. Obviously, $M \subset I$; we show that $M \models P^I$, which implies $I \notin \mathit{SM}(P)$. Suppose $M \not\models P^I$. Then some rule $r \in P^I$ exists such that $\mathsf{body}(r) \subseteq M$ and $\mathsf{head}(r) \cap M = \emptyset$. As $I \models P^I$ but $M \not\models P^I$, r has one of the following forms:

  (a) $A_1(t) \vee \ldots \vee A_k(t) \leftarrow B_0(t), \ldots, B_l(t)$,

  (b) $R_1(t, f_1(t)) \vee \ldots \vee R_k(t, f_k(t)) \leftarrow P_0(t, g_0(t)), \ldots, P_l(t, g_l(t))$,

  (c) $A_1(f(t)) \vee \ldots \vee A_k(f(t)) \leftarrow B_0(f(t)), \ldots, B_l(f(t))$,

  (d) $A_1(f(t)) \vee \ldots \vee A_k(f(t)) \leftarrow B_1(Z_1), \ldots, B_m(Z_m), R_0(t, f(t)), \ldots, R_l(t, f(t))$, or

  (e) $R_1(t, f_1(t)) \vee \ldots \vee R_k(t, f_k(t)) \leftarrow B_0(t), \ldots, B_l(t)$,

  where each $Z_i \in \{t, f(t)\}$, and $k, l, m \geq 0$. The rules above are derived by taking all rules of P^I that have only atoms with terms t or $f(t)$ in the head. Since M results from I by removing some atoms with the above property, r must have such atoms in the head.

Suppose the violated rule r is of the form (a). Then $K \setminus H$ contains an atom $A(t)$, for some unary predicate symbol A. It follows that $H \not\models P(U)^K$. This holds since $P(U)^K$ contains $A(t) \leftarrow$ by the definition of local programs.

Consequently, 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 $\mathsf{body}(r)$, we have $\mathsf{body}(r) \subseteq H$. Furthermore, due to $\mathsf{head}(r) \cap M = \emptyset$, we have $\mathsf{head}(r) \cap H = \emptyset$. This contradicts the assumption that $H \models P(U)^K$.

(B) $\Rightarrow$ (A). Suppose (B) holds, but $I \notin SM(P)$. Then, $I \notin MM(P^I)$ and again, there are two possibilities:

- $I \not\models P^I$. Then a rule $r \in P^I$ exists such that $\mathsf{body}(r) \subseteq I$ and $\mathsf{head}(r) \cap I = \emptyset$. Since $\mathit{ffa}(I) \in SM(\mathsf{gp}(P))$ and r belongs to the reduct of $\mathsf{gp}(P)$ w.r.t. $\mathit{ffa}(I)$, r cannot be function-free. Satisfaction of the other 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.

- $I \models P^I$, but is not minimal. Then, some $H \subset I$ exists such that $H \in MM(P^I)$. Due to forest-shaped model property, H is forest-shaped. If $\mathit{ffa}(H) \subset \mathit{ffa}(I)$ would hold, then $\mathit{ffa}(I) \notin SM(\mathsf{gp}(P))$ would hold. Therefore, $\mathit{ffa}(H) = \mathit{ffa}(I)$ must hold and some term t must exists satisfying the following two conditions.

  (a) It holds that:

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

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

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

  Intuitively, t is some smallest term (w.r.t. depth) 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. By choice of t, $K' = H \cap \mathcal{HB}_s$ is a knot such that $K' \subset K$ and $\mathsf{st}(K', s) = \mathsf{st}(K, s)$. As $H \models P^I$, it is easily verified that $K' \models P(\mathsf{st}(K, s))^K$; thus, K is not stable w.r.t. P, a 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. By choice of t and failure of (II), $K' = H \cap \mathcal{HB}_t$ is a knot such that $K' \subset K$ and $\mathsf{st}(K', t) = \mathsf{st}(K, t)$. Again, if $H \models P^I$, then $K' \models P(\mathsf{st}(K, t))^K$; hence K is not stable w.r.t. P, a contradiction.

In both cases we arrive at a contradiction to the assumption that $I \notin SM(P)$. □

3.1.2 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 3.14 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 3.15 (Knot Instance $K_{\downarrow u}$). *Given a term u and a knot K with root term t, the knot $K_{\downarrow u}$ results from K by replacing all occurrences of t in K with 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 3.16 (Scan $\mathbb{K}(I)$). *Let I be a forest-shaped interpretation for P. We define the set $\mathbb{K}(I)$ of $\mathbf{x}$-grounded knots as $\mathbb{K}(I) = \{(I \cap \mathcal{HB}_t)_{\downarrow \mathbf{x}} \mid t \hat{\in} I\}$.*

Example 3.17 (Cont'd). *In our bacteria example, for the stable model I (cf. Example 3.2) we have $\mathbb{K}(I) = \{K_7, K_{12}, K_{28}\}$, where each knot is from Figure 3.3. Note that the maximum term depth in I is 2, and that K_{28} has no child nodes.*

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. An easy observation is that stability of a knot is preserved under substitutions.

Proposition 3.18. *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.*

Example 3.19 (Cont'd). *Recall that the knots K_1 and K_2 in Figure 3.2 are stable w.r.t. P, and so are their $\mathbf{x}$-grounded versions. In total, there exist 28 $\mathbf{x}$-grounded knots that are stable w.r.t. P, which are shown in Figure 3.3.*

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., the terms t and s satisfy the same unary predicates.

$$\begin{aligned}
K_1 &= \emptyset,\ K_2 = \{W(\mathbf{x})\},\ K_3 = \{M(\mathbf{x})\},\ K_4 = \{Y(\mathbf{x})\},\ K_5 = \{M(\mathbf{x}), Y(\mathbf{x})\},\\
K_6 &= \{W(\mathbf{x}), Y(\mathbf{x}), Ch(\mathbf{x}, g(\mathbf{x})), M(g(\mathbf{x})), W(g(\mathbf{x}))\,\}\\
K_7 &= \{W(\mathbf{x}), Y(\mathbf{x}), Ch(\mathbf{x}, g(\mathbf{x})), M(g(\mathbf{x})), C(g(\mathbf{x}))\,\}\\
K_8 &= \{M(\mathbf{x}), W(\mathbf{x}), Ch(\mathbf{x}, c_1(\mathbf{x})), Y(c_1(\mathbf{x})), W(c_1(\mathbf{x})), Ch(\mathbf{x}, c_2(\mathbf{x})), Y(c_2(\mathbf{x})), W(c_2(\mathbf{x}))\,\}\\
K_9 &= \{M(\mathbf{x}), Y(\mathbf{x}), W(\mathbf{x}), Ch(\mathbf{x}, c_1(\mathbf{x})), Ch(\mathbf{x}, c_2(\mathbf{x})),\\
&\qquad Ch(\mathbf{x}, g(\mathbf{x})), Y(c_1(\mathbf{x})), W(c_1(\mathbf{x})), Y(c_2(\mathbf{x})), W(c_2(\mathbf{x})), M(g(\mathbf{x})), C(g(\mathbf{x}))\,\},\\
K_{10} &= \{M(\mathbf{x}), W(\mathbf{x}), Y(\mathbf{x}), Ch(\mathbf{x}, g(\mathbf{x})), Ch(\mathbf{x}, c_1(\mathbf{x})),\\
&\qquad Ch(\mathbf{x}, c_2(\mathbf{x})), Y(c_1(\mathbf{x})), Y(c_2(\mathbf{x})), M(g(\mathbf{x})), W(g(\mathbf{x})), W(c_1(\mathbf{x})), W(c_2(\mathbf{x}))\,\}\\
K_{11} &= \{C(\mathbf{x}), Y(\mathbf{x}), Ch(\mathbf{x}, d(\mathbf{x})), C(d(\mathbf{x})), D(d(\mathbf{x}))\,\}\\
K_{12} &= \{M(\mathbf{x}), C(\mathbf{x}), Ch(\mathbf{x}, d(\mathbf{x})), C(d(\mathbf{x})), D(d(\mathbf{x}))\,\}\\
K_{13} &= \{M(\mathbf{x}), C(\mathbf{x}), Y(\mathbf{x}), Ch(\mathbf{x}, d(\mathbf{x})), C(d(\mathbf{x})), D(d(\mathbf{x}))\,\}\\
K_{14} &= \{C(\mathbf{x}), Ch(\mathbf{x}, d(\mathbf{x})), C(d(\mathbf{x})), D(d(\mathbf{x}))\,\}\\
K_i &= K_{i-14} \cup \{D(\mathbf{x})\},\ i = 15, \ldots, 24\\
K_j &= K_{j-14} \cup \{D(\mathbf{x})\} \setminus \{Ch(\mathbf{x}, d(\mathbf{x})), C(d(\mathbf{x})), D(d(\mathbf{x}))\},\ j = 25, \ldots, 27\\
K_{28} &= \{C(\mathbf{x}), D(\mathbf{x})\}
\end{aligned}$$

Figure 3.3: All stable $\mathbf{x}$-grounded knots of the bacteria program

Definition 3.20 (Founded Knot Set). *Let L be a set of* $\mathbf{x}$*-grounded knots. Then L is* founded *w.r.t. a program P and a set of states S, if the following hold:*

1. *each knot $K \in L$ is stable w.r.t. P;*
2. *for each $U \in S$, there exists $K \in L$ such that $U \approx \mathsf{st}(K, \mathbf{x})$;*
3. *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 $\langle K_0, \ldots, K_n\rangle$ of knots in L such that:*
      - *$K_n = K$,*
      - *K_0 is such that $\mathsf{st}(K_0, \mathbf{x}) \approx U$ for some $U \in S$, and*
      - *for each $0 \le i < n$, there exists $s \in \mathsf{succ}(K_i)$ s.t. $\mathsf{st}(K_i, s) \approx \mathsf{st}(K_{i+1}, \mathbf{x})$.*

Example 3.21 (Cont'd). *In our example, the set of all* $\mathbf{x}$*-grounded stable knots (see Figure 3.3) is founded w.r.t. P and $S = \{\mathsf{st}(I) \mid I$ is an interpretation of $P\}$. Indeed, for any interpretation I and $c \,\hat{\in}\, I$, some knot K_i exists such that $\mathsf{st}(I, c) \approx \mathsf{st}(K, \mathbf{x})$; hence, Condition 1) is satisfied. As easily seen, Condition 2) also holds.*

The following is easy to verify (recall $\mathsf{st}(I)$ from Definition 3.6).

Proposition 3.22. *Let $I \in SM(P)$. Then $\mathbb{K}(I)$ is a set of knots that is founded w.r.t. P and* $\mathsf{st}(I)$.

Example 3.23 (Cont'd). *Recall that for the stable model I (cf. Example 3.2) we have $\mathbb{K}(I) = \{K_7, K_{12}, K_{28}\}$. It is easily checked that $\mathbb{K}(I)$ is founded w.r.t. P and* $\mathsf{st}(I)$, *which contains the single state $\{\mathit{Young}(b),\ \mathit{Warm}(b)\}$: the knot K_7 satisfies condition 1), and considering K_7, K_{12}, and K_{28} in this order we can verify condition 2) (note that* $\mathsf{succ}(K_{28}) = \emptyset$*).*

In what follows, we give a construction of stable models out of knots in a founded set. Moreover, we characterize the set of stable models via founded knot sets.

Generating Stable Models using Knots

To construct stable models as forest-shaped interpretations from knots in a founded knot set, we start with constructing respective trees, which are represented as usual by prefix-closed sets of words. For a sequence of elements $p = [e_1, \dots, e_n]$, let $\tau(p)$ denote the last element e_n, and $[p|e_{n+1}]$ denote the sequence $[e_1, \dots, e_n, e_{n+1}]$.

Definition 3.24 (Tree Construction). *Given a set L of $\mathbf{x}$-grounded knots and a state U^t, a set T of sequences $[e_1, \dots, e_n]$, whose elements $e_i = \langle K_i, t_i\rangle$ are pairs of knots K_i and terms t_i, is a* tree induced by L with root state U^t*, if:*

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

(b) For every $p \in T$ with $\tau(p) = \langle K, t\rangle$ and $f(\mathbf{x}) \in$ $\mathsf{succ}(K)$, *T contains some $[p|\langle K', f(t)\rangle]$ s.t. $K' \in L$ and* $\mathsf{st}(K, f(\mathbf{x})) \approx \mathsf{st}(K', \mathbf{x})$.

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

Intuitively, each path $p \in T$ is a node in the tree. If $\tau(p) = \langle K, t\rangle$, then p represents the term t and the $\mathbf{x}$-grounded knot K; to obtain an interpretation, K will be instantiated with t. To obtain stable models, we require for closure under successor knots (see 3.a in Definition 3.20) which is achieved via (b) above.

Example 3.25 (Cont'd). *Let $L = \mathbb{K}(I)$ for the stable model I of P in Example 3.2. Then the tree $T = \{\, [\langle K_7, b\rangle],\ [\langle K_7, b\rangle, \langle K_{12}, \mathit{grow}(b)\rangle],\ [\langle K_7, b\rangle, \langle K_{12}, \mathit{grow}(b)\rangle, \langle K_{28}, \mathit{die}(\mathit{grow}(b))\rangle]\}$ is induced by L with root state $\{\mathit{Young}(b), \mathit{Warm}(b)\} \approx$* $\mathsf{st}(K_7, \mathbf{x})$.

A tree T induced by some $\mathbf{x}$-grounded knot set L with root state U^t is transformed into a set of ground atoms defined by $T_{\downarrow} = \bigcup\{K_{\downarrow t} \mid p \in T \text{ with } \tau(p) = \langle K, t\rangle\}$. This is generalized to collections of trees whose roots are connected as follows.

Definition 3.26 (Forest Model Construction)**.** *Let G be a set of function-free ground atoms and let L be a set of knots founded w.r.t. P and a set of states $S \supseteq \mathsf{st}(G)$. 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} is a tree induced by L with root state $\mathsf{st}(G, c_i)$.

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

Theorem 3.27. *If $G \in SM(\mathsf{gp}(P))$, and L is a set of knots that is founded w.r.t. P and some $S \supseteq \mathsf{st}(G)$, 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 3.18, 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 $G \in SM(\mathsf{gp}(P))$, Theorem 3.14 implies that I is a stable model of P. $\square$

Example 3.28 (Cont'd)**.** *The set $G = \{Young(b), Warm(b)\}$ is the single stable model of $\mathsf{gp}(P)$, and $\mathbb{K}(I) = \{K_7, K_{12}, K_{28}\}$ is founded w.r.t. P and $\mathsf{st}(I) = \mathsf{st}(G)$ (= $\{G\}$) for the stable model I of P. The tree T in Example 3.25 is induced by $\mathbb{K}(I)$, and in fact it is the only tree induced by $\mathbb{K}(I)$ with root state G. Hence, $\mathcal{F}(G, \mathbb{K}(I))$ contains the single interpretation $G \cup (T^b)_{\downarrow}$, which coincides with I.*

We showed that stable model existence can be proved by checking that some suitable founded knot set 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 knot sets is obvious.

Proposition 3.29. *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. Recall $\mathsf{st}(P)$ from Definition 3.6.

Definition 3.30 ($\mathbb{K}_P$)**.** *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 $S \subseteq \mathsf{st}(\mathsf{gp}(P))$.*

Due to Proposition 3.29 and Definition 3.30, the following is immediate.

Proposition 3.31. *For the program P, the following hold:*

(a) $\mathbb{K}_P$ *is founded w.r.t.* P *and some* $S \subseteq \mathsf{st}(\mathsf{gp}(P))$.

(b) If L *is a set of knots that is founded w.r.t.* P *and some* $S \subseteq \mathsf{st}(\mathsf{gp}(P))$*, 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*, for every* $S \subseteq \mathsf{st}(\mathsf{gp}(P))$.

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 3.20.

Example 3.32 (Cont'd)**.** *In our bacteria example,* $\mathbb{K}_P = \{K_6, K_7, K_8, K_{12}, K_{28}\}$*. Note that* $\mathbb{K}_P$ *contains besides the knots* K_7*,* K_{12}*,* K_{28} *in* $\mathbb{K}(I)$ *for the stable model* I *in Example 3.2 also the knots* K_6 *and* K_8*; the initial part of the stable model shown in Figure 3.1 is built using instances of* K_6 *and* K_8*.*

Proposition 3.33. *If* $I \in SM(P)$*, then* $I \in \mathcal{F}(\mathit{ffa}(I), L)$ *for every set of knots* $L \supseteq \mathbb{K}(I)$ *that is founded w.r.t.* P *and some state set* $S \supseteq \mathsf{st}(I)$*.*

The following will be helpful.

Definition 3.34 (Compatible $\mathbb{K}_P$)**.** *We call* $\mathbb{K}_P$ compatible *with a set of states* S*, if for every state* $U \in S$ *some* $K \in \mathbb{K}_P$ *exists 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 $\mathsf{gp}(P)$, it represents the latter.

Theorem 3.35. *Let* I *be an interpretation for* P*. Then,* $I \in SM(P)$ *iff* $I \in \mathcal{F}(G, \mathbb{K}_P)$*, for some* $G \in SM(\mathsf{gp}(P))$ *such that* $\mathbb{K}_P$ *is compatible with* $\mathsf{st}(G)$*.*

Proof. If $I \in SM(P)$, then, by Proposition 3.22, $\mathbb{K}(I)$ is founded w.r.t. P and $\mathsf{st}(I)$. By definition, $\mathbb{K}(I) \subseteq \mathbb{K}_P$. By Proposition 3.31, $\mathbb{K}_P$ is founded w.r.t. P and some $S \supseteq \mathsf{st}(I)$. By Proposition 3.33, $I \in \mathcal{F}(\mathit{ffa}(I), \mathbb{K}_P)$. Note that $\mathit{ffa}(I) \in SM(\mathsf{gp}(P))$. The other direction is proved by Theorem 3.27. $\square$

We have obtained a finite representation of the stable models of an $\mathbb{FDNC}$ program P. Indeed, each of its stable models can be generated out of some stable model of $\mathsf{gp}(P)$ and the knot set $\mathbb{K}_P$.

Example 3.36 (Cont'd)**.** *From* $\mathbb{K}_P = \{K_6, K_7, K_8, K_{12}, K_{28}\}$ *and the only stable model* $G = \{\mathit{Young}(b), \mathit{Warm}(b)\}$ *of* $\mathsf{gp}(P)$*, we can construct the stable model* I *from Example 3.2, as well as any other stable model of* P*.*

We can view $\mathsf{gp}(P)$ together with $\mathbb{K}_P$ as a compilation of the program P that can be exploited for reasoning and stable model building (see Sections 3.3 and 3.4).

Problem	$\mathbb{F}$	$\mathbb{FD}$	$\mathbb{FC}$	$\mathbb{FDC}$, $\mathbb{FN}$, $\mathbb{FNC}$, $\mathbb{FDNC}$
Consistency	Trivial	Trivial	PSPACE (3.4.2)	EXPTIME (3.3.2, 3.4.1)
$P \models_b A(\vec{t})$	P (3.4.3)	Σ_2^P (3.4.3)	PSPACE (3.4.2)	EXPTIME (3.3.3)
$P \models_b \exists\vec{x}.A(\vec{x})$	PSPACE (3.4.3)	PSPACE (3.4.3)	PSPACE (3.4.2)	EXPTIME (3.3.3)
$P \models_c A(\vec{t})$	P (3.4.3)	co-NP (3.4.3)	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 (3.3.4)	PSPACE	EXPSPACE (3.3.4)

Table 3.1: Complexity of $\mathbb{FDNC}$ and Fragments (Completeness Results)

3.2 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 3.1. 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 infinite 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-shaped 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 reoccurrences of patterns, e.g., states of terms, non-isomorphic labeled arcs, trees of depth 1, etc. To find algorithms for $\mathbb{FDNC}$, we could resort to methods of description logics, which often have the forest-shaped model property and are decided by tableau methods with blocking. Unfortunately, such methods are not well-suited in our case. First, they cannot easily handle minimality testing and are generally not worst-case optimal. Second, tableau methods are designed for consistency testing, while some important tasks from nonmonotonic reasoning (e.g. brave reasoning) cannot always be polynomially reduced to consistency testing (see Table 3.1).

Therefore, our algorithms for $\mathbb{FDNC}$ rely on the finite representation of stable models in terms of maximal founded sets of knots. In Section 3.3.1, 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 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 3.3.2), brave entailment of ground and existential queries (Section 3.3.3), 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 nonmonotonic 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. The EXPTIME-hardness of consistency testing already in $\mathbb{FDC}$ is proved in Section 3.3.2 by an encoding of an EXPTIME-hard description logic $\mathcal{ALC}$, which is extended to $\mathbb{FN}$ in Section 3.4.1. The hardness of consistency testing directly provides lower bounds for brave and cautious entailment of ground and existential queries. We note also that the EXPTIME-completeness results for $\mathbb{FDC}$ and $\mathbb{FN}$ show that these fragments are equal in terms of problem solving capacity: unlike in the propositional setting (cf. [DEGV01]), negation alone can polynomially compensate disjunction and constraints, and vice versa.

For the fragment $\mathbb{FC}$ of $\mathbb{FDNC}$, the picture is different. Its programs have the unique model property, i.e., if a stable model exists, it is unique. For the standard reasoning tasks, a procedure thus 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 Theorem [Sav70], turned into deterministic polynomial space algorithms. The PSPACE-hardness of consistency testing is shown by a Turing machine encoding, which is extended to other standard reasoning tasks (see Section 3.4.2).

If we disallow nonmonotonic 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 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 3.4.3 discusses the results for $\mathbb{F}$ and $\mathbb{FD}$.

The last row in Table 3.1 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 3.3.4 provides the details. As noted in the preliminaries, in case of brave entailment, the semantics of open and existential queries coincide, and hence the complexity results on existential queries carry over to open queries.

The main entries in Table 3.1 are presented with references to the sections that discuss the respective problems in detail. The other entries are justified as follows:

1. Programs in $\mathbb{F}$ and $\mathbb{FD}$ are positive and without constraints, hence consistent.

2. 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 $\mathbb{FDC}$) holds as consistency checking with constraints in $\mathbb{FC}$ (resp. $\mathbb{FDC}$) is reducible to cautious inference. On the other hand, completeness also holds as cautious inference is reducible to inconsistency testing in the standard way.

3. Similarly, PSPACE (resp. EXPTIME) membership of $P \models_c A(\vec{t})$, where P is a program in $\mathbb{FC}$ (resp. $\mathbb{FDC}$, $\mathbb{FN}$, $\mathbb{FNC}$ or $\mathbb{FDNC}$), holds as the problem amounts to checking consistency of $P \cup \{\leftarrow A(\vec{t})\}$. On the other hand, hardness holds as P is inconsistent iff $P \models_c A'(t)$, where A' is a fresh symbol and t is arbitrary.

4. PSPACE-completeness of $P \models_c \lambda\vec{x}.A(\vec{x})$ in $\mathbb{F}$ and $\mathbb{FC}$ holds 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 focus on unary queries.

Lemma 3.37. *Let C be a complexity class in Table 3.1, and let $\mathcal{L}$ be from the $\mathbb{F}$ family. Then:*

(i) If deciding program consistency for $\mathcal{L}$ is C-hard, then deciding brave entailment of queries (ground or existential, unary or binary) is also C-hard for $\mathcal{L}$.

(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 $\mathcal{L}$.

For a proof of Lemma 3.37, we refer to the appendix. Intuitively, the first statement holds as brave reasoning involves a construction of a stable model containing a certain atom, which cannot be easier than constructing (or checking the existence of) an arbitrary stable model. The second statement hinges on the fact that the class $\mathbb{F}$ allows rules $A(y) \leftarrow R(x,y)$ and $R(x, f(x)) \leftarrow A(x)$, by which brave entailment of binary queries can be reformulated in terms of unary queries, and vice versa. Similarly, with rules in the syntax of $\mathbb{F}$, one constructs reductions proving *(iii)*.

3.3 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 $SM(\mathsf{gp}(P))$, captures all the stable models of an $\mathbb{FDNC}$ program P (see Theorem 3.35).

3.3.1 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 that surely contains $\mathbb{K}_P$. In the second phase, we remove some knots from it to ensure that it satisfies Definition 3.30.

To ease the presentation, for any knot set L, let $\mathsf{st}^{+1}(L) = \{\mathsf{st}(K, s) \mid K \in L, s \in \mathsf{succ}(K)\}$, i.e., $\mathsf{st}^{+1}(L)$ is the set of all states of the successor terms of knots in L.

Definition 3.38 ($\mathsf{All}(P)$)**.** *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 $U \in \mathsf{st}(\mathsf{gp}(P))$ and $K \in SM(P(U))$, then $K_{\downarrow\mathbf{x}} \in \mathsf{All}(P)$.

b) If $U \in \mathsf{st}^{+1}(\mathsf{All}(P))$ 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 $S \subseteq \mathsf{st}(P)$. The first problem is that $\mathsf{All}(P)$ might contain knots K that lack some successor knots, i.e., for some $s \in \mathsf{succ}(K)$ no K' is in $\mathsf{All}(P)$ s.t. $\mathsf{st}(K, s) \approx \mathsf{st}(K', \mathbf{x})$ (condition (3.a) in Definition 3.20). On the other hand, each knot in $\mathbb{K}_P$ must be reachable from a state in $\mathsf{st}(P)$ (condition (3.b)). These requirements are ensured by removing some knots from $\mathsf{All}(P)$.

Definition 3.39 ($\mathsf{bad}(L)$)**.** *For any set of $\mathbf{x}$-grounded knots L, $\mathsf{bad}(L)$ is the smallest subset of L such that $K \in \mathsf{bad}(L)$, if for some $s \in \mathsf{succ}(K)$, either*

a) no $K' \in L$ fulfills $\mathsf{st}(K, s) \approx \mathsf{st}(K', \mathbf{x})$, or

b) for all $K' \in L$, $\mathsf{st}(K, s) \approx \mathsf{st}(K', \mathbf{x})$ implies $K' \in \mathsf{bad}(L)$.

Intuitively, obtaining the set $\mathsf{All}(P) \setminus \mathsf{bad}(\mathsf{All}(P))$ corresponds to iteratively removing from $\mathsf{All}(P)$ the knots that have no successors (note that removing a knot from a set L might leave some other knots in L without a successor).

The following notion will help to ensure satisfaction of (3.b) of Definition 3.20.

Definition 3.40 ($\mathsf{reach}_S(L)$)**.** *For any set of $\mathbf{x}$-grounded knots L and set of states S, $\mathsf{reach}_S(L)$ 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 $K \in \mathsf{reach}_S(L)$, and

b) if $U \in \mathsf{st}^{+1}(\mathsf{reach}_S(L))$, $K \in L$ and $U \approx \mathsf{st}(K, \mathbf{x})$, then $K \in \mathsf{reach}_S(L)$.

Intuitively, $\mathsf{reach}_S(L)$ are the knots in L reachable from the states in S. Indeed, if $\mathsf{reach}_S(L) = L$, then L fulfills condition (3.b) of Definition 3.20 w.r.t. S.

Theorem 3.41. *If P is an $\mathbb{FDNC}$ program, then $\mathbb{K}_P = \mathsf{reach}_{\mathsf{st}(\mathsf{gp}(P))}(L_P)$, where $L_P = \mathsf{All}(P) \setminus \mathsf{bad}(\mathsf{All}(P))$.*

Proof. Let $L = \mathsf{reach}_{\mathsf{st}(\mathsf{gp}(P))}(\mathsf{All}(P) \setminus \mathsf{bad}(\mathsf{All}(P)))$. We verify that L satisfies the conditions in Definition 3.30, i.e., L is the single $\subseteq$-minimal set which contains each knot set L' that is founded w.r.t. P and some $S \subseteq \mathsf{st}(\mathsf{gp}(P))$.

Indeed, every such L' fulfills $L' \subseteq \mathsf{All}(P)$; by construction of L, no $K \in L'$ is removed, thus $L' \subseteq L$. To prove the result, it is thus sufficient to show that L itself is founded w.r.t. P and some $S \subseteq \mathsf{st}(\mathsf{gp}(P))$ (cf. Definition 3.20). The definition of $\mathsf{All}(P)$ ensures that every $K \in L$ is stable w.r.t. P. Furthermore, the removal of knots in $\mathsf{bad}(\mathsf{All}(P))$ and restriction to reachable knots ensures that every $K \in L$ has proper successors as in (3.a) of Definition 3.20, and has a proper predecessor sequence as in (3.b) that reaches a state in $\mathsf{st}(\mathsf{gp}(P))$ (we can set S suitably). ☐

It is easy to see that we can compute $\mathbb{K}_P$ in time at most single exponential in the size of an $\mathbb{FDNC}$ program P. This 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 $b(P) = 2^{n+k\cdot(n+m)}$, when P has k function, n unary, and m binary predicate symbols.
- Computing $\mathsf{All}(P)$ requires adding at most $b(P)$ $\mathbf{x}$-grounded knots. Each such knot has polynomial size and its stability is verifiable using a $\Sigma_2^P = \mathrm{NP}^{\mathrm{NP}}$ oracle. Thus, $\mathsf{All}(P)$ is computable in time single exponential in the size of P.
- Computing $\mathsf{bad}(L)$ is polynomial in the size of L. Thus, $\mathsf{All}(P) \setminus \mathsf{bad}(\mathsf{All}(P))$ is computable in time single exponential in the size of P.
- The size of $\mathsf{st}(\mathsf{gp}(P))$ is bounded by a single exponential in the size of P.
- Computing $\mathsf{reach}_S(L)$ is polynomial in the combined size of L and S.

3.3.2 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 3.42. *For every $\mathbb{FDNC}$ program P, the following are equivalent:*

(i) P *is consistent.*

(ii) For some $G \in SM(\mathsf{gp}(P))$*, the set* $\mathbb{K}_P$ *is compatible with* $\mathsf{st}(G)$.

Proof. If $I \in SM(P)$, then by Theorem 3.35 some $G \in SM(\mathsf{gp}(P))$ exists such that $\mathbb{K}_P$ is compatible with $\mathsf{st}(G)$. The converse is proved by Theorem 3.27. ☐

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

Theorem 3.43. *Deciding whether an* $\mathbb{FDNC}$ *program is consistent is in* EXPTIME.

Proof. Deciding whether $\mathbb{K}_P$ is compatible with $\mathsf{st}(G)$, for some $G \in SM(\mathsf{gp}(P))$, is feasible in time polynomial in $n + m$, where m is the size of $\mathbb{K}_P$ and n is the size of $SM(\mathsf{gp}(P))$. 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 $SM(\mathsf{gp}(P))$ is computable in single exponential time, the result follows from Theorem 3.42. ☐

As we have pointed earlier already, we can see $\mathbb{K}_P$ together with $\mathsf{gp}(P)$ 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 $\mathsf{gp}(P)$ using knots from $\mathbb{K}_P$ (and every stable model of P results by proper choices). Here the hard part is computing such a G, which depending on the complexity of function-free 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 $\mathsf{st}(G)$ is polynomial, and each tree expansion step using a knot from $\mathbb{K}_P$ is feasible with low (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 stable model engines like DLV or Smodels.

In the following, we show that the algorithm emerging from Theorem 3.42 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 3.44 ($\mathcal{ALC}$ Syntax)**.** *Let* $\mathbf{C} \supseteq \{\top, \bot\}$, $\mathbf{R}$ *and* $\mathbf{I}$ *denote the sets of* concept, role, *and* individual names, *respectively. Inductively, each* $C \in \mathbf{C}$ *is a concept, and 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 concepts. If* $C \in \mathbf{C}$*, then* C *and* $\neg C$ *are* literal concepts.

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

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

Figure 3.4: Semantics of the DL $\mathcal{ALC}$ by mapping to first-order logic

A general concept inclusion axiom *(GCI) is of form $C \sqsubseteq D$ where C, D are concepts. An* assertion *is of form $C(a)$ or $R(a,b)$, where $a, b \in \mathbf{I}$, $R \in \mathbf{R}$, and $C \in \mathbf{C}$. 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 3.4 (see e.g. [HSG04] for details). The major reasoning task in $\mathcal{ALC}$ is deciding the *consistency* of a given $\mathcal{K}$, i.e., that 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 3.45 ($\mathcal{ALC}$ 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)	$A_0 \sqcap \ldots \sqcap A_n \sqsubseteq B_0 \sqcup \ldots \sqcup B_m,$	*(T4)*	$A_0 \sqsubseteq \exists R.B_0,$
(T2)	$A_0 \sqcap \ldots \sqcap A_n \sqsubseteq \bot,$	*(T5)*	$A_0 \sqsubseteq \forall R.B_0,$
(T3)	$\top \sqsubseteq B_0 \sqcup \ldots \sqcup B_m,$		

where $n, m \geq 0$, and each A_i and B_j is from $\mathbf{C}$, but is neither $\top$ nor $\bot$.[1] *Moreover, if $\mathcal{K}$ is in normal form and does not contain axioms of type (T3), then $\mathcal{K}$ is* safe.[2]

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

[1]For a similar normal form for the weaker description logic $\mathcal{EL}^{++}$, see [BBL05].

[2]We require safety because applying the transformation Π to an axiom $\top \sqsubseteq B_0 \sqcup \ldots \sqcup B_m$ leads to the formula $\forall x.B_0(x) \vee \ldots \vee B_m(x)$ which, in turn, cannot be stated as a safe rule in $\mathbb{FDNC}$ syntax.

	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 3.2: Translating $\mathcal{ALC}$ into $\mathbb{FDNC}$

Proposition 3.46. *Given any $\mathcal{ALC}$ knowledge base $\mathcal{K}$, we can obtain in linear time a safe knowledge base $\mathcal{K}'$ in normal form that is consistent iff $\mathcal{K}$ is consistent.*

The proof, which is based on well-known *definitional form transformations*, is given in the appendix. 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 3.2.

Proposition 3.47. *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 [Her71], $P^{\mathcal{K}}$ is consistent iff $\Theta(\mathcal{K})$ consistent. □

Note that $P^{\mathcal{K}}$ is in fact positive and constructible in linear time from $\mathcal{K}$. Hence, Propositions 3.46–3.47 and the well-known EXPTIME-hardness of $\mathcal{ALC}$ [Sch91] imply that deciding consistency of $\mathbb{FDC}$ and $\mathbb{FDNC}$ programs is EXPTIME-hard. Combined with Theorem 3.43, we establish the completeness result.

Theorem 3.48. *Deciding consistency of $\mathbb{FDC}$ and of $\mathbb{FDNC}$ programs is* EXPTIME-*complete.*

3.3.3 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 of ground queries. The idea behind the method is to perform some "back-propagation" of unary predicate symbols in the set of knots.

Definition 3.49 ($\mathsf{reach}_A(L)$)**.** *For every set L of $\mathbf{x}$-grounded knots and unary predicate symbol A, let $\mathsf{reach}_A(L)$ be the smallest subset of L such that:*

(a) if $K \in L$ and $A(\mathbf{x}) \in K$, then $K \in \mathsf{reach}_A(L)$, and

(b) if $K' \in \mathsf{reach}_A(L)$ is a possible successor of $K \in L$, i.e., for some $s \in \mathsf{succ}(K)$ we have $\mathsf{st}(K, s) \approx \mathsf{st}(K', \mathbf{x})$, then $K \in \mathsf{reach}_A(L)$.

Intuitively, $K \in \mathsf{reach}_A(L)$ 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 $SM(\mathsf{gp}(P))$ captures the stable models of P, we have the following:

Theorem 3.50. *For every program P, the following statements are equivalent:*

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

(B) Some $G \in SM(\mathsf{gp}(P))$ exists such that (i) $\mathbb{K}_P$ is compatible with $\mathsf{st}(G)$, and (ii) for some constant c and $K \in \mathbb{K}_P$, $\mathsf{st}(G, c) \approx \mathsf{st}(K, \mathbf{x})$ and $K \in \mathsf{reach}_A(\mathbb{K}_P)$.

Proof. Suppose (A) holds, i.e., there exists some $I \in SM(P)$ such that $A(t) \in I$, for some term t. By Theorem 3.14, $\mathit{ffa}(I) \in SM(\mathsf{gp}(P))$. Let $G = \mathit{ffa}(I)$. By Proposition 3.22, $\mathbb{K}(I)$ is a set of knots that is founded w.r.t. P and $\mathsf{st}(G)$. Due to the definition of $\mathbb{K}_P$ and Proposition 3.31, we have that $\mathbb{K}_P$ is compatible with $\mathsf{st}(G)$. It remains to show that (ii) in (B) holds. Consider $\mathsf{reach}_A(\mathbb{K}(I))$. Due to the fact that $A(t) \in I$ and the construction of $\mathsf{reach}_A(\mathbb{K}(I))$, for some constant c there exists $K \in \mathbb{K}(I)$ such that $\mathsf{st}(G, c) \approx \mathsf{st}(K, \mathbf{x})$ and $K \in \mathsf{reach}_A(\mathbb{K}(I))$. 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 $K \in \mathsf{reach}_A(\mathbb{K}(I))$ implies $K \in \mathsf{reach}_A(\mathbb{K}_P)$. Therefore, (ii) holds.

Suppose (B) holds. The facts that $G \in SM(\mathsf{gp}(P))$, and that $\mathbb{K}_P$ is compatible with $\mathsf{st}(G)$ 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 3.27). 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. $\square$

Theorem 3.50 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 program P. Indeed, computing $\mathsf{reach}_A(\mathbb{K}_P)$ requires time quadratic in the size of $\mathbb{K}_P$, or single exponential in the size of P. Once $\mathbb{K}_P$, $\mathsf{reach}_A(\mathbb{K}_P)$, and $SM(\mathsf{gp}(P))$ are computed, the conditions in (B) are verifiable in time polynomial in the combined size of $\mathbb{K}_P$, $\mathsf{reach}_A(\mathbb{K}_P)$, and $SM(\mathsf{gp}(P))$. 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 3.48 and Lemma 3.37, we know that brave entailment of unary existential queries is ExpTime-hard already for $\mathbb{FDC}$. Therefore, we conclude the following.

Theorem 3.51. *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 3.37).*

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

Definition 3.52 ($\mathsf{goal}_q(L)$)**.** *Let $q = A(t)$ be a ground atom and L be a set of $\mathbf{x}$-grounded knots. Let T be the set of subterms of the term t. Then $\mathsf{goal}_q(L)$ is the smallest relation over $L \times T$ such that:*

(a) if $K \in L$ and $A(\mathbf{x}) \in K$, then $\langle K, t\rangle \in \mathsf{goal}_q(L)$, 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 $\langle K', f(v)\rangle \in \mathsf{goal}_q(L)$, then $\langle K, v\rangle \in \mathsf{goal}_q(L)$.

Intuitively, $\mathsf{goal}_q(L)$ tries to construct proofs of q by backward chaining in L; a proof succeeds, if some pair $\langle K, c\rangle$ is obtained where c is constant symbol. Due to the properties of $\mathbb{K}_P$, we obtain:

Theorem 3.53. *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) Some $G \in SM(\mathsf{gp}(P))$ exists such that (i) $\mathbb{K}_P$ is compatible with $\mathsf{st}(G)$, and (ii) some $K \in \mathbb{K}_P$ exists such that $\mathsf{st}(G, c) \approx \mathsf{st}(K, \mathbf{x})$ and $\langle K, c\rangle \in \mathsf{goal}_q(\mathbb{K}_P)$.

Proof. Similar to the proof of Theorem 3.50, and thus omitted. ☐

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 $\mathsf{goal}_q(\mathbb{K}_P)$ 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 $\mathbb{K}_P$, $\mathsf{goal}_q(\mathbb{K}_P)$, and $SM(\mathsf{gp}(P))$ are computed, the conditions in (B) can be verified in time polynomial in the combined size of $\mathbb{K}_P$, $\mathsf{goal}_q(\mathbb{K}_P)$, and $SM(\mathsf{gp}(P))$, each of which is single exponential in the size of P and q. Thus brave entailment of unary ground queries by $\mathbb{FDNC}$ programs is in ExpTime. To establish completeness, recall Theorem 3.48 and item (ii) in Lemma 3.37.

Theorem 3.54. *For $\mathbb{FDC}$ and $\mathbb{FDNC}$ programs, brave entailment of unary ground queries (resp., binary ground queries) is* ExpTime*-complete.*

3.3.4 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 3.2, 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. We introduce the following notion.

Definition 3.55 (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, we use the following notion for knots that can start models. For a constant c, let $\mathsf{seeds}(c, P)$ be the set of all knots $K \in \mathbb{K}_P$ such that for some $G \in SM(\mathsf{gp}(P))$ we have $\mathsf{st}(K, \mathbf{x}) \approx \mathsf{st}(G, c)$ and $\mathbb{K}_P$ is compatible with $\mathsf{st}(G)$.

Proposition 3.56. *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 some converging sequence $s = [\frac{L_0}{c}, \frac{L_1}{f_1}, \ldots, \frac{L_n}{f_n}]$ for A exists, where $L_0 = \mathsf{seeds}(c, P)$.*

Proof. For the "only if" direction, suppose a ground 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 3.55. 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 $\mathbb{K}_P$ with root S such that $N \in T$. Consider the stable model $I \in SM(P)$ where $\mathsf{st}(I, c) \approx S$. Such I must exists due to the way we defined L_0. By the semantic characterization (see Theorem 3.35), I can be represented as $I = \mathit{ffa}(I) \cup (T^{c_1})_{\downarrow} \cup \ldots \cup (T^{c_n})_{\downarrow}$, where $\{c_1, \ldots, c_n\}$ is the set of all constants of P, and each T^{c_i} is a tree induced by $\mathbb{K}_P$ with root $\mathsf{st}(\mathit{ffa}(I), c_i)$. W.l.o.g., $c_1 = c$. Simply define $I' = \mathit{ffa}(I) \cup (T)_{\downarrow} \cup (T^{c_2})_{\downarrow} \cup \ldots \cup (T^{c_n})_{\downarrow}$. By Theorem 3.27, 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 3.55 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 3.55.

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 3.55 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 3.55.

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

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 is double exponentially bounded in the size of the initial program.

Proposition 3.57. *For every converging sequence* $[\frac{L_0}{c}, \frac{L_1}{f_1}, \dots, \frac{L_n}{f_n}]$ *for* A*, there exists a converging sequence* $[\frac{L_0}{c}, \frac{L'_1}{f'_1}, \dots, \frac{L'_m}{f'_m}]$ *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 $s = [\frac{L_0}{c}, \frac{L_1}{f_1}, \dots, \frac{L_n}{f_n}]$ 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 $[\frac{L_0}{c}, \frac{L_1}{f_1}, \dots, \frac{L_k}{f_k}, \frac{L_{l+1}}{f_{l+1}}, \dots, \frac{L_n}{f_n}]$ 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 a converging sequence s' for A exists 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. $\square$

The next theorem follows directly from Proposition 3.56 and Proposition 3.57.

Theorem 3.58. *Let* P *be an* $\mathbb{FDNC}$ *program and let* $\lambda x.A(x)$ *be an open query. Then* $P \models_c \lambda x.A(x)$ *iff* P *is inconsistent or some converging sequence* $[\frac{L_0}{c}, \frac{L_1}{f_1}, \dots, \frac{L_n}{f_n}]$ *for* A *exists, 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 3.3.4 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 guesses a sequence of functions symbols and verifies the conditions in Definition 3.55. It 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 Theorem [Sav70], we can turn the algorithm into an algorithm using exponential space, which establishes the ExpSpace-membership. By a generic Turing machine encoding, we show that the problem is ExpSpace-hard, even for $\mathbb{FD}$ and $\mathbb{FN}$ programs (see Appendix).

Theorem 3.59. *Cautious entailment of open queries in* $\mathbb{FD}$*,* $\mathbb{FN}$*,* $\mathbb{FNC}$*,* $\mathbb{FDC}$ *and* $\mathbb{FDNC}$ *programs is* ExpSpace*-complete.*

3.4 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 Section 3.3 and the discussion in Section 3.2, including cautious entailment of existential queries in $\mathbb{FD}$ (cf. Theorem 3.48 and Lemma 3.37) and of open queries in general (cf. Theorem 3.59).

```
**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 $f(\mathbf{x}) \notin \mathsf{succ}(K)$ **then**
        **return** false
    **else**
        $L^{aux} := \{K' \in \mathbb{K}_P \mid \mathsf{st}(K', \mathbf{x}) \approx \mathsf{st}(K, f(\mathbf{x})) \wedge K \in L\}$;
        $L := L^{aux}$;
        $i := i + 1$
    **end if**
**until** $i = |F| \times 2^{|\mathbb{K}_P|} + 1$
**return** $false$
```

Figure 3.5: 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.

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.

3.4.1 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 $\mathbb{FDC}$ and $\mathbb{FDNC}$, 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 $\mathsf{fr}(P)$ (the *frame* program) whose stable models intuitively coincide with the minimal (forest-shaped) Herbrand interpretations for P. We then structurally transform P into an $\mathbb{FN}$ program P' such that P is consistent iff the $\mathbb{FN}$ program $P' \cup \mathsf{fr}(P)$ is consistent.

Definition 3.60 (Frame Program $\mathsf{fr}(P)$)**.** *For each predicate Q of P, let $\bar{Q}$ be a fresh predicate symbol of the same arity, Let Dom and S be fresh unary and binary predicate symbols, respectively. Then $\mathsf{fr}(P)$ is the $\mathbb{FN}$ program with the rules*

(F1)	$Dom(c) \leftarrow,$	*(F5)*	$A(x) \leftarrow Dom(x), not\ \bar{A}(x)$
(F2)	$S(c,d) \leftarrow,$	*(F6)*	$\bar{A}(x) \leftarrow Dom(x), not\ A(x)$
(F3)	$S(x, f(x)) \leftarrow Dom(x),$	*(F7)*	$R(x,y) \leftarrow S(x,y), not\ \bar{R}(x,y),$
(F4)	$Dom(y) \leftarrow S(x,y),$	*(F8)*	$\bar{R}(x,y) \leftarrow S(x,y), not\ R(x,y),$

for each pair c, d of constants of P, each function symbol f of P, and each unary and binary predicate symbol A and R of P, respectively.

Proposition 3.61. *Given an interpretation $I \subseteq \mathcal{HB}^{\mathsf{fr}(P)}$, $I \in SM(\mathsf{fr}(P))$ iff I is forest-shaped, and*

(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.*

Proof. Let I be a stable model of $\mathsf{fr}(P)$. The properties (1), (2) and (3) hold by the construction of $\mathsf{fr}(P)$, i.e., due to the fact that I is a stable model that satisfies (F1)-(F4) rules of $\mathsf{fr}(P)$. 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 $\{A(t) \leftarrow Dom(t); \bar{A}(t) \leftarrow Dom(t)\} \subseteq \mathsf{fr}(P)^I$. Since $Dom(t) \in I$, I is not a model of $\mathsf{fr}(P)^I$. In case (b), by construction of $\mathsf{fr}(P)$, there is no rule in $\mathsf{fr}(P)^I$ that has head $A(t)$ or $\bar{A}(t)$, and hence I is not a minimal model of $\mathsf{fr}(P)^I$. In both cases, we arrive to a contradiction to the assumption that I is a stable model of $\mathsf{fr}(P)$. Analogously to the argument for (4), one can show that the property (5) holds.

For the other direction, assume an interpretation I of $\mathsf{fr}(P)$ for which the given properties hold. It is easy to see that such an interpretation satisfies each of the rules in $\mathsf{fr}(P)^I$. We verify that I is a minimal model of $\mathsf{fr}(P)^I$. By the construction of $\mathsf{fr}(P)$, each minimal model of $\mathsf{fr}(P)$ has to satisfy (1), (2) and (3). Therefore, if I is not a minimal model of $\mathsf{fr}(P)^I$, there should exists a model $H \subset I$ of $\mathsf{fr}(P)^I$ for which (4) or (5) does not hold. We arrive to a contradiction. Due to the rules of types (F5-F8) in $\mathsf{fr}(P)$, H cannot be a model of $\mathsf{fr}(P)^I$. ☐

Intuitively, $\mathsf{fr}(P)$ 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 non-disjunctive. It is easy to see that such rules can be eliminated from P in linear time while preserving consistency: replace each rule $R_1(x, f_1(x)) \vee \ldots \vee R_n(x, f_k(x)) \leftarrow B_0(x), \ldots, B_l(x)$ by rules $A_1(x) \vee \ldots \vee A_n(x) \leftarrow B_0(x), \ldots, B_l(x)$ and $R_i(x, f_i(x)) \leftarrow A_i(x)$ for each $i \in \{1, \ldots, n\}$, where each A_i is a fresh predicate symbol.

Definition 3.62 (Transformation)**.** *For an $\mathbb{FDC}$ program P as described, we denote by $\mathtt{tf}(P)$ the $\mathbb{FN}$ program $\mathsf{fr}(P) \cup P'$, 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 $n, m \geq 0$.

We note that for simplicity, all rules are rewritten, including non-disjunctive rule and constraints. Indeed, $\mathtt{tf}(P)$ 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 rules (R6). The following holds:

Proposition 3.63. *The program P is consistent iff $\mathtt{tf}(P)$ is consistent.*

Proof. Suppose P is consistent, and I is a minimal model of P. We know that I is forest-shaped. Let J be a Herbrand interpretation for $\mathtt{tf}(P)$ 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 $t \mathrel{\hat{\in}} J$, then $Dom(t) \in J$,

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

e) if $t \mathrel{\hat{\in}} J$ 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 $\mathsf{tf}(P)$. Assume that it is not the case. There are two possibilities.

1) J is not a model of $\mathsf{tf}(P)^J$. Since $\mathsf{fr}(P) \subseteq \mathsf{tf}(P)$, we have $\mathsf{fr}(P)^J \subseteq \mathsf{tf}(P)^J$. From the construction of J it follows that J is a model of $\mathsf{fr}(P)^J$. Then $\mathsf{tf}(P)^J$ must contain some ground rule

$$C(\vec{v}_1) \leftarrow Q_1(\vec{v}_1), \dots, Q_m(\vec{v}_m), \bar{W}_1(\vec{t}_1), \dots, \bar{W}_n(\vec{t}_n)$$

such that ($\star$) $\{Q_1(\vec{v}_1), \dots, Q_m(\vec{v}_m), \bar{W}_1(\vec{t}_1), \dots, \bar{W}_n(\vec{t}_n)\} \subseteq J$ and $C(\vec{v}_1) \notin J$. By construction, P contains the rule $W_1(\vec{t}_1) \vee \ldots \vee W_n(\vec{t}_n) \leftarrow Q_1(\vec{v}_1), \dots, Q_m(\vec{v}_m)$, where $n, m \geq 0$. Since I is a model of P, then either (a) $\{Q_1(\vec{v}_1), \dots, Q_m(\vec{v}_m)\} \not\subseteq I$ or (b) $\{W_1(\vec{t}_1), \dots, W_n(\vec{t}_n)\} \cap I \neq \emptyset$. 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.

2) J is a model but is not a minimal model of $\mathsf{tf}(P)^J$. Since $\mathsf{fr}(P) \subseteq \mathsf{tf}(P)$, we have $\mathsf{fr}(P)^J \subseteq \mathsf{tf}(P)^J$. Then we also have that J is a model of $\mathsf{fr}(P)^J$, but is not minimal. However, by construction of J, we have $J \subseteq \mathcal{HB}^{\mathsf{fr}(P)}$ and J satisfies the conditions in Proposition 3.61, and hence by the same proposition, J must be a minimal model of $\mathsf{fr}(P)^J$. Contradiction.

For the other direction, let $I \in \mathit{SM}(\mathsf{tf}(P))$. Let J be the restriction of I to the predicates of P. Suppose $J \not\models P$. Then $\mathsf{Ground}(P)$ contains a rule

$$W_1(\vec{t}_1) \vee \ldots \vee W_n(\vec{t}_n) \leftarrow Q_1(\vec{v}_1), \dots, Q_m(\vec{v}_m),$$

where $n, m \geq 0$, such that $\{Q_1(\vec{v}_1), \dots, Q_m(\vec{v}_m)\} \subseteq J$ and $\{W_1(\vec{t}_1), \dots, W_n(\vec{t}_n)\} \cap J = \emptyset$. By construction, $\mathsf{Ground}(\mathsf{tf}(P))$ contains

$$r = C(\vec{t}_1) \leftarrow Q_1(\vec{v}_1), \dots, Q_m(\vec{v}_m), \bar{W}_1(\vec{t}_1), \dots, \bar{W}_n(\vec{t}_n), \mathit{not}\ C(\vec{t}_1).$$

By hypothesis, $I \in \mathit{SM}(\mathsf{tf}(P))$. Clearly, $C(t_1) \notin I$ (otherwise, $I \notin \mathit{MM}(\mathsf{tf}(P)^I)$ as no rule in $\mathsf{tf}(P)^I$ would have $C(t_1)$ in the head). Hence, $\mathsf{tf}(P)^I$ contains the rule resulting from r by removing $\mathit{not}\ C(\vec{t}_1)$. Since $\{Q_1(\vec{v}_1), \dots, Q_m(\vec{v}_m)\} \subseteq I$ and $I \models \mathsf{tf}(P)^I$, it follows that $\bar{W}_i(\vec{t}_i) \notin I$ for some $i \in \{1, \dots, n\}$. As I is forest-shaped, by the rules in $\mathsf{fr}(P)$ we have $W_i(\vec{t}_i) \in I$, and thus $W_i(\vec{t}_i) \in J$. However, this contradicts that $\{W_1(\vec{t}_1), \dots, W_n(\vec{t}_n)\} \cap J = \emptyset$. □

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 $\mathsf{fr}(P)$; the rest is linear). Therefore, recalling EXPTIME-completeness of consistency testing in $\mathbb{FDNC}$ (Theorem 3.48), we conclude.

Corollary 3.64. *Checking consistency of $\mathbb{FN}$ and $\mathbb{FNC}$ programs 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 3.37), and since brave entailment of existential and ground queries is EXPTIME-complete (see Theorems 3.51 and 3.54), we obtain:

Corollary 3.65. *For $\mathbb{FN}$ and $\mathbb{FNC}$ programs, brave entailment of unary (resp., binary, by Lemma 3.37) ground or existential queries is* EXPTIME*-complete.*

3.4.2 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 large. Nevertheless, the semantic characterization centering around Theorem 3.14 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 3.66. *Let P be an $\mathbb{FDNC}$ program. The following two are equivalent:*

(i) $SM(P) \neq \emptyset$.

(ii) There exists some $G \in SM(\mathsf{gp}(P))$ such that, for each constant c of P, a set of knots exists that is founded w.r.t. P and $\{\mathsf{st}(G, c)\}$.

Proof. For the "(i) to (ii)" direction, assume that I is a stable model of P. By Theorem 3.14, $\mathit{ffa}(I)$ is a stable model of $\mathsf{gp}(P)$. So let $G = \mathit{ffa}(I)$. By Proposition 3.22, we know that the set of knots $\mathbb{K}(I)$ is founded w.r.t. P and $\mathsf{st}(G)$. Simply take some $\subseteq$-minimal set L of knots closed under the following rules:

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

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

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 3.29, the set $L = \bigcup_{c \in C} L_c$ is a set of knots that is founded w.r.t. P and $\mathsf{st}(G)$. Then Theorem 3.27 proves the claim. $\square$

The key feature of $\mathbb{FC}$ is the unique model property, i.e., if an $\mathbb{FC}$ program has a minimal model, then it is unique. From Theorem 3.66, we know that to decide whether a $\mathbb{FC}$ program is consistent we can proceed in two steps:

```
**func** $checkCondition$ (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** $i = b(P)$;
**return** $false$
```

Figure 3.6: Non-deterministic procedure for PSPACE algorithms

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

(2) Check whether for each constant c of P, a set of knots exists that is founded w.r.t. P and $\{\mathsf{st}(G, c)\}$. If so then P is consistent, otherwise 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 a unique set L that is founded w.r.t. P and $\{U\}$, where U is a state.

To decide the second step, in Figure 3.6 we present a generic non-deterministic procedure $checkCondition$. 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 $b(P)$ is the number of distinct $\mathbf{x}$-grounded knots over the signature of P. As it was already argued, $b(P) = 2^{n+k\cdot(n+m)}$, 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 3.67. *Assume an $\mathbb{FC}$ program P, and let U be a state. Then, 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 $\{U\}$. □

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

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 $0 \leq \pi < \pi' \leq sb(I)$:
$Sym_{\sigma,\pi}(y) \leftarrow N(x, y), Sym_{\sigma,\pi}(x), Cur_{\pi'}(x)$
$Sym_{\sigma,\pi'}(y) \leftarrow N(x, y), Sym_{\sigma,\pi'}(x), Cur_\pi(x)$

Table 3.3: $\mathbb{FC}$ program $P(T, I)$ for simulating a DTM T on input I.

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 [Sav70], we know co-NPSPACE $=$ PSPACE.

Lemma 3.68. *Deciding whether a given* $\mathbb{FC}$ *program is consistent is in* PSPACE.

On the other hand, PSPACE-hardness of the problem is shown by a Turing machine simulation.

Lemma 3.69. *Deciding whether a given* $\mathbb{FC}$ *program is consistent is* PSPACE*-hard.*

Proof. 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 $\lvert I \rvert$. The computation of T on I can be simulated by an $\mathbb{F}$ program $P(T, I)$ (see Table 3.4.2). 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.

As the translation is clearly polynomial in the size of T and I, deciding $I \in \mathcal{L}$ is reducible in polynomial time to consistency checking of an $\mathbb{FC}$ program. ☐

Thus we obtain the following.

Theorem 3.70. *For $\mathbb{FC}$ programs, checking consistency is* PSPACE*-complete.*

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

Proposition 3.71. *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 $\mathbb{FC}$. Recalling that the task is PSPACE-hard (Lemma 3.37), we conclude the following.

Corollary 3.72. *For $\mathbb{FC}$ programs, brave entailment of unary existential queries is* PSPACE*-complete. The same holds for binary existential queries (see Lemma 3.37).*

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

Proposition 3.73. *Let P be an $\mathbb{FC}$ program and let $A(t)$ be a ground atom such that $t = f_n(\ldots f_1(c_0)\ldots)$. Let P' result from P by adding the following rules: (a) $C_0(c_0) \leftarrow$, (b) $R(x, f_{i+1}(x)) \leftarrow C_i(x)$, for $0 \leq i < n$, (c) $C_{i+1}(y) \leftarrow C_i(x), R(x,y)$, for $0 \leq i < n$, (d) $D(x) \leftarrow C_n(x), A(x)$, and (e) $\leftarrow D(x)$, where $C_0, \ldots, C_n$, R and D are fresh predicates. 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 3.37, we have the following result.

Theorem 3.74. *For $\mathbb{FC}$ programs, brave entailment of unary ground queries is* PSPACE*-complete. The same holds for binary queries (see Proposition 3.37).*

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

As $\mathbb{F}$ and $\mathbb{FD}$ programs are positive and constraint-free, they 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 of existential queries in $\mathbb{F}$ and $\mathbb{FD}$, respectively, follows from the results for consistency testing in $\mathbb{FC}$ and $\mathbb{FDC}$ (see observation (2) in Section 3.2).

For a given $\mathbb{F}$ program P, deciding $P \models_b \exists x.A(x)$ is feasible in polynomial space (see Corollary 3.72). On the other hand, the problem is easily seen to be PSPACE-hard; this can be shown by a simple adaptation of the Turing machine simulation for Theorem 3.70.

Lemma 3.75. *For $\mathbb{F}$ programs, brave entailment of unary existential queries is* PSPACE-*hard.*

Proof. Recall the program $P(T, I)$ in Table 3.4.2 that simulates a computation of a DTM T on input I. Note that it is an $\mathbb{F}$ program and has size polynomial in the size of T and I. To check whether T accepts I, we can pose the brave query whether $St_{accept}(t)$ is in the minimal model of $P(T, I)$ for some term t, i.e., T accepts I iff $P(T, I) \models_b \exists x.St_{accept}(x)$. ☐

Theorem 3.76. *For $\mathbb{F}$ programs, brave entailment of unary existential queries is* PSPACE*-complete. The same holds for binary existential queries (see Lemma 3.37).*

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 $\mathbb{F}$. The use of constraints leads to $\mathbb{FDC}$, where consistency testing is already EXPTIME-complete.

The strategy is to use the non-deterministic procedure $checkCondition$ from Section 3.4.2 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 3.77. *Let P be a $\mathbb{FD}$ program. The following two are equivalent:*

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

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

of Theorem 3.77. If (i) holds, then, due to Theorem 3.14, we can easily define G and L such that the conditions in (ii) are satisfied. On the other hand, if (ii) is satisfied, the fact that for each $G \in MM(\mathsf{gp}(P))$ there is some $M \in MM(P)$ such that $G = \mathit{ffa}(M)$ and Theorem 3.14 imply that a minimal model of P such that $A(t) \in P$ for some term t is constructible. Indeed, take some M for G as described. By Theorem 3.14, M is forest-shaped. As L is founded w.r.t. P and $\{\mathsf{st}(G, c)\}$, the tree in M rooted at c can be replaced by some tree that is built with instances of knots from L only, and such that some instance of a knot K containing $A(\mathbf{x})$ is used. As the resulting model I is forest-shaped, by Theorem 3.14 we obtain that $I \in SM(P)$. ☐

Let $q = \exists x.A(x)$ be a query and P be an $\mathbb{FD}$ program. The theorem above suggests a method to decide whether $P \models_b q$ holds. The crucial point is to have a procedure for deciding whether for a given state U over P, there exists a set of knots L that is founded w.r.t. P and $\{U\}$ and contains 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 3.78. *Let U be a state, and P be an $\mathbb{FD}$ program. The following two are equivalent.*

(i) Some knot set L founded w.r.t. P and $\{U\}$ and $K \in L$ exist such that $A(\mathbf{x}) \in K$.

(ii) Some execution of $checkCondition(P, U, cond_2)$ returns $true$.

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

(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 missing successors knots. □

By similar arguments as for consistency checking in $\mathbb{FC}$, $checkCondition$ runs in polynomial space in the input size. Note that traversing the states of constants that occur in the minimal models of $\mathsf{gp}(P)$ is feasible in polynomial space. Hence, condition (ii) in Theorem 3.77 is testable in polynomial space using a PSPACE oracle; overall, this amounts to polynomial space. As brave entailment of existential queries for $\mathbb{FD}$ programs is PSPACE-hard (see Lemma 3.37), we conclude:

Theorem 3.79. *For $\mathbb{FD}$ programs, brave entailment of unary existential queries is* PSPACE*-complete. The same holds for binary existential queries (see Lemma 3.37).*

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 3.80. *Suppose P be is an $\mathbb{FD}$ program and $A(t)$ a ground atom in which as single constant c occurs. Let $l = \langle s_1, \ldots, s_n \rangle$ be the list of subterms of t ordered by increasing term depth, i.e., $s_1 = c$ and $s_n = t$. Then the following hold:*

1. *$P \models_b A(t)$ if and only if ($\star$) there exists some stable model G of $\mathsf{gp}(P)$ and a sequence $\langle K_1, \ldots, K_n \rangle$ of stable knots with roots $s_1, \ldots, s_n$, resp., 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* $1 \leq i < n$, *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 $\mathsf{gp}(P)$ and a sequence $\langle K_1, \ldots, K_n \rangle$ of knots with roots $s_1, \ldots, s_n$, respectively, 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* $1 \leq i < n$,

(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 3.14, we can simply define $G = \mathit{ffa}(I)$ and $K_i = \mathcal{HB}_{s_i} \cap I$, where $1 \leq i \leq n$. For the if direction, since for $\mathbb{FD}$ programs each $G \in SM(\mathsf{gp}(P))$ is extendible to some $M \in SM(P)$, we can similarly as in the proof of Theorem 3.77 construct using Theorem 3.14 a stable model of P containing $A(s_n)$; simply start with $G \cup K_1 \cup \ldots \cup K_n$ and extend the set with the necessary stable knots.

For the only-if direction of the second claim, the argument is as for the first claim. If $I \in SM(P)$ such that $A(t) \notin I$, then, by Theorem 3.14, we can easily define G and the sequence of knots. Again, take $G = \mathit{ffa}(I)$ and $K_i = \mathcal{HB}_{s_i} \cap I$, where $1 \leq i \leq n$. For the if 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 3.14, 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 3.80 allows us to derive complexity results for ground queries. To ease presentation, the characterization above is via witnessing knot sequences: for brave entailment via a witness for entailment and for cautious entailment via a witness of a counter-model. Note that in (2.c), the knots are not necessarily stable. Stability is not needed, and in fact would hinder finding counter-models in nondeterministic polynomial time (due to stability testing).

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 $\mathsf{gp}(P)$ is also computable in polynomial time. Hence, $P \models_b A(t)$ can be decided according to Proposition 3.80 by constructing in polynomial time the least model of $\mathsf{gp}(P)$ 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.

Next suppose that P is an $\mathbb{FD}$ program and $A(t)$ a ground query. As easily seen, testing condition ($\star$) for P is in Σ_2^P: indeed, guess an interpretation I for $\mathsf{gp}(P)$ and a suitable knot sequence $\langle K_1, \ldots, K_n \rangle$ over P's signature; this results in a polynomial-size structure. One can then check in polynomial time with an NP oracle whether I is minimal and each knot K_i satisfies the conditions in ($\star$).

To decide $P \not\models_c A(t)$, it suffices to verify the condition ($\star\star$) in Proposition 3.80. Since ($\star\star$) does not involve minimality of models, it is decidable in NP. Indeed, we may guess an interpretation for $\mathsf{gp}(P)$ and a candidate sequence of knots over the signature of P. Deciding then whether the structure satisfies ($\star\star$) is feasible in polynomial time. 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 L 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 logic programs are L-reducible to brave/cautious entailment of ground unary queries in $\mathbb{F}$ and $\mathbb{FD}$ programs respectively. Since brave entailment over positive propositional disjunctive programs is Σ_2^P-complete and cautious entailment is co-NP-complete, while both tasks are P-complete for Horn programs (see e.g. [DEGV01]), we obtain completeness results for our formalisms.

Theorem 3.81. *For $\mathbb{FD}$ programs, brave (resp., cautious) entailment of unary ground queries is Σ_2^P-complete (resp. co-*NP*-complete). For $\mathbb{F}$ programs, both problems are* P*-complete. All results extend to binary queries (see Lemma 3.37).*

3.5 Reasoning about Actions and Planning

In Section 3.3, we have already encountered an application of $\mathbb{FDNC}$ programs to Description Logics. In this section, we consider a further application of $\mathbb{FDNC}$ programs in the area of reasoning about actions and planning; recall that nonmonotonic 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 $\mathcal{K}$ [EFL$^+$04].

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 thesis, we refer to [Bar02] for background and a study of these problems based on logic programs under answer set semantics.

Example 3.82. *As for temporal projection in Example 3.2, 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 $Young(cell_1(grow(b)))$ 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 3.73 an auxiliary fact $C_0(b)$. and rules $R(x, grow(x)) \leftarrow C_0(x)$; $C_1(y) \leftarrow C_0(x), R(x, y)$; $R(x, cell_1(x)) \leftarrow C_1(x)$; and $\leftarrow R(x, y), not\ Change(x, y)$, 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 $Young(cell_1(grow(b)))$, 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 $\mathcal{A}$ [GL92] (which was extensively studied in [Bar02]), $\mathcal{C}$ [GL98], and $\mathcal{K}$ [EFL$^+$04]. 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 nonmonotonic 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.[3] At the core, it distinguishes two kinds of predicates: *fluents* and *actions*. A *state* is 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 (or credulous) plan* for a given planning domain PD and a goal G is a sequence of action occurrences $\langle \mathcal{A}_1, \ldots, \mathcal{A}_n \rangle$, $n \geq 0$, that legally transforms the initial state I into some state that satisfies the goal G, i.e., for some sequence of states, we have (i) $S_0 = I$, (ii) each $S_i, \mathcal{A}_{i+1}, S_{i+1}$ is a legal transition, and (iii) S_n satisfies G. For our concerns, $\mathcal{A}_i$ is a single action.

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* (also known 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 char-

[3]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 [EFL$^+$04].

acterized 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

$$\begin{array}{rl} \texttt{caused}\, D & \texttt{if}\, B_1, \ldots, B_n, \mathit{not}\ B_{n+1}, \ldots, \mathit{not}\ B_k \\ & \texttt{after}\, C_1, \ldots, C_m, \mathit{not}\ C_{m+1}, \ldots, \mathit{not}\ C_l \\ & \quad A_1, \ldots, A_v, \mathit{not}\ A_{v+1}, \ldots, \mathit{not}\ A_w \end{array} \tag{3.1}$$

$k, l, w \geq 0$, 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 (3.1) describes the (incomplete) knowledge state after action execution, where the knowledge depends on fluents that hold ($C_1, \ldots, C_m$) and do not hold ($C_{m+1}, \ldots, C_l$) in the old state, fluents that hold ($B_1, \ldots, B_n$) and do not hold ($B_{n+1}, \ldots, B_k$) in the new state, and actions that were executed ($A_1, \ldots, A_v$) respectively were not executed ($A_{v+1}, \ldots, A_w$) in parallel.[4]

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

$$\begin{array}{rl} \texttt{executable}\, A\ \texttt{if} & B_1, \ldots, B_n, \mathit{not}\ B_{n+1}, \ldots, \mathit{not}\ B_k, \\ & A_1, \ldots, A_m, \mathit{not}\ A_{m+1}, \ldots, \mathit{not}\ A_l, \end{array} \tag{3.2}$$

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

$$\texttt{initially caused}\, D\ \texttt{if}\, B_1, \ldots, B_n, \mathit{not}\ B_{n+1}, \ldots, \mathit{not}\ B_k \tag{3.3}$$

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. As in Section 3.6, we enhance $\mathbb{FDNC}$ programs with "strong" negation $\neg P(\vec{x})$ [GL91], which is expressed in the core language as usual.

- 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 $S(\mathit{init}) \leftarrow$ denoting that the constant

[4]For our concerns, we may restrict to $v \leq 1$.

$init$ 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 $a(x, f_A(x)) \leftarrow \mathit{Exec}_A(x)$ 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 (3.1) is transformed into the following rule:

$$\begin{aligned} D(y) \leftarrow \quad & B_1(y), \ldots, B_n(y), \mathit{not}\ B_{n+1}(y), \ldots, \mathit{not}\ B_k(y), \\ & C_1(x), \ldots, C_m(x), \mathit{not}\ C_{m+1}(x), \ldots, \mathit{not}\ C_l(x), \\ & A_1(x, y), \ldots, A_v(x, y), \mathit{not}\ A_{v+1}(x, y), \ldots, \mathit{not}\ A_w(x, y), Tr(x, y) \end{aligned}$$

- The executability condition (3.2) is transformed into the following rule:

$$\begin{aligned} \mathit{Exec}_a(y) \leftarrow \quad & S(y), B_1(y), \ldots, B_n(y), \mathit{not}\ B_{n+1}(y), \ldots, \mathit{not}\ B_k(y), \\ & A_1(x, y), \ldots, A_v(x, y), \mathit{not}\ A_{v+1}(x, y), \ldots, \mathit{not}\ A_w(x, y). \end{aligned}$$

  Here, we assume for simplicity as in [EFL$^+$03] that there are no positive cyclic interdependencies between actions.

- The initial state constraint (3.3) is transformed into the following rule:

$$d(\mathit{init}) \leftarrow \quad B_1(\mathit{init}), \ldots, B_n(\mathit{init}), \mathit{not}\ B_{n+1}(\mathit{init}), \ldots, \mathit{not}\ B_k(\mathit{init}),$$

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

$$G_1, \ldots, G_n, \mathit{not}\ G_{n+1}, \ldots, \mathit{not}\ G_k \tag{3.4}$$

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

$$\mathit{Plan}(x) \leftarrow G_1(x), \ldots, G_n(x), \mathit{not}\ G_{n+1}(x), \ldots, \mathit{not}\ G_k(x) \tag{3.5}$$

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 $S_0, \mathcal{A}_1, S_1, \mathcal{A}_2, \ldots$ in PD, i.e., alternating sequences of states S_i and action occurrences $\mathcal{A}_{i+1}$ such that each $S_i, \mathcal{A}_{i+1}, S_{i+1}$ is a legal transition, $0 \leq i < n$, 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 [HJ99]; 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 [EFL$^+$04]).

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 $t = f_{A_k}(f_{\mathcal{A}_{k-1}} \cdots (f_{A_1}(init)))$ where $P(PD)$ is $P(PD, G)$ except the rules (3.4) and (3.5). Whether D is necessarily true when $\vec{A}$ would have happened can be expressed, using again a similar technique as in Proposition 3.73, as cautious entailment of $D(t)$ from $P(D)$ augmented with the auxiliary fact $C_0(init) \leftarrow$ and rules

$$\begin{aligned} R(x, f_{\mathcal{A}_{i+1}}(x)) &\leftarrow C_i(x), \text{ for } 0 \leq i < k, \\ C_{i+1}(y) &\leftarrow C_i(x), R(x, y), \text{ for } 0 \leq i < k-1, \\ &\leftarrow R(x, y), not\, Tr(x, y), \end{aligned}$$

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 [Bar02] can be similarly expressed.

Example 3.83. *Table 3.4 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 [HM87]. 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*

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

Table 3.4: Example of Planning Domain Encoding (mapping via $\leadsto$)

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.*

To provide a procedure for deciding plan existence in the planning domains of $\mathcal{K}$, the authors of [EFL$^+$04] 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 from 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.

We remark that using higher arity $\mathbb{FDNC}$, we can represent the Yale-Shooting scenario alternatively using a generic predicate $holds(f, x)$ to express truth of the fluent f in a situation x, where f is reified using a constant symbol, in the style of [Bar02]; e.g., $holds(Loaded, init)$ corresponds then to $Loaded(init)$. Further predicates, e.g. $abnormal(f, x)$, 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 $holds(on(A, B), x)$, which is also used [Bar02], this can be easily accommodated with tailored predicates, e.g. $holds_{on}(A, B, x)$; 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.

3.6 Higher-arity $\mathbb{FDNC}$

We present here a decidable extension of $\mathbb{FDNC}$ that supports predicate and function symbols of higher arities, and allows for more succinct and convenient knowledge representation in practice. This will be illustrated by a plain planning scenario (for a more detailed discussion, see the previous Section 3.5).

As already illustrated by the previous examples, $\mathbb{FDNC}$ supports naturally modeling of possibly infinite evolutions of a set of propositions. Indeed, for a term t, the unary predicates (i.e., the propositions) that hold for it can be used via rules of $\mathbb{FDNC}$ to define the unary predicates that hold for the term $f(t)$, where $f(t)$ can be viewed as a follow-up time point. However, a propositional setting is inconvenient for many action domains, and the use for parameters for more compact representation is needed. They allow to represent actions that, for instance, move an object x from a location l_1 to the location l_2. Hence, special predicate names for each possible combination of x, l_1, and l_2 as in a propositional setting can be avoided. (This is, e.g., widely used in [Bar02].)

In what follows, we assume that $\mathcal{N}_1$ and $\mathcal{N}_2$ are disjoint sets of predicate names of arities at least 1 and 2, respectively.

Definition 3.84 (Global/local positions)**.** *Given an atom $A(t_1, \dots, t_n)$ with $A \in \mathcal{N}_1$ or a term $f(t_1, \dots, t_n)$, its* local positions *are $1, \dots, n-1$ and its* global position *is n. Similarly, given an atom $A(t_1, \dots, t_n)$ with $A \in \mathcal{N}_2$, its* local positions *are $1, \dots, n-2$ and its* global positions *are $n-1$ and n. An atom $A(\vec{t})$ with $A \in \mathcal{N}_1$ (resp., $A \in \mathcal{N}_2$) is* g-unary *(resp.,* g-binary*).*

Definition 3.85 (Higher-arity $\mathbb{FDNC}$ program)**.** *A* higher-arity $\mathbb{FDNC}$ *program is a finite disjunctive logic program whose rules are of the following forms:*

(R1) $$\bigvee_{i=1}^{k} A_i(\vec{v_i}, x) \leftarrow B_0(\vec{u_0}, x), \bigwedge_{j=1}^{l} B_j^{\pm}(\vec{u_j}, x)$$

(R2) $$\bigvee_{i=1}^{k} R_i(\vec{v_i}, x, y) \leftarrow P_0(\vec{u_0}, x, y), \bigwedge_{j=1}^{l} P_j^{\pm}(\vec{u_j}, x, y)$$

(R3) $$\bigvee_{i=1}^{k} R_i(\vec{v_i}, x, f_i(\vec{t_i}, x)) \leftarrow P_0(\vec{u_0}, x, g_0(\vec{w_0}, x)), \bigwedge_{j=1}^{l} P_j^{\pm}(\vec{u_j}, x, g_j(\vec{w_j}, x))$$

(R4) $$\bigvee_{i=1}^{k} A_i(\vec{v_i}, y) \leftarrow P_0(\vec{u_0}, x, y), \bigwedge_{j=1}^{l} P_j^{\pm}(\vec{u_j}, x, y), \bigwedge_{j=1}^{m} B_j^{\pm}(\vec{w_j}, x), \bigwedge_{j=1}^{n} C_j^{\pm}(\vec{t_j}, y)$$

(R5) $$\bigvee_{i=1}^{k} A_i(\vec{v_i}, f(\vec{v}, x)) \leftarrow P_0(\vec{u_0}, x, f(\vec{v}, x)), \bigwedge_{j=1}^{l} P_j^{\pm}(\vec{u_j}, x, f(\vec{v}, x)), \bigwedge_{j=1}^{m} B_j^{\pm}(\vec{w_j}, x), \bigwedge_{j=1}^{n} C_j^{\pm}(\vec{t_j}, f(\vec{v}, x))$$

(R6) $$\bigvee_{i=1}^{k} R_i(\vec{v_i}, x, f_i(\vec{t_i}, x)) \leftarrow B_0(\vec{u_0}, x), \bigwedge_{j=1}^{l} B_j^{\pm}(\vec{u_j}, x)$$

(R7) $$\bigvee_{i=1}^{k} A_i(\vec{t_i}, b) \vee \bigvee_{i=1}^{l} R_i(\vec{v_i}, c, d) \leftarrow \bigwedge_{j=1}^{m} B_j^{\pm}(\vec{u_j}, b'), \bigwedge_{j=1}^{n} P_j^{\pm}(\vec{w_j}, c', d'),$$

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

- *all A_i and B_j are from $\mathcal{N}_1$, and all R_i and P_j are from $\mathcal{N}_2$,*
- *the tuples $\vec{v}$, $\vec{w}$, and all $\vec{v_i}, \vec{t_i}$, $\vec{u_j}$, $\vec{w_j}$ are tuples of variables or constants.*
- *b, c, d, b', c', d' are constants,*
- *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* $\mathsf{body}^+(r)$.

The restrictions on the variable interaction allow us to transform higher-arity $\mathbb{FDNC}$ programs naturally into ordinary $\mathbb{FDNC}$ programs, which enables the usage of reasoning methods from the previous sections. In the following, we present the transformation and a use case of a higher-arity $\mathbb{FDNC}$ program.

Definition 3.86 ($\mathbb{FDNC}$ reduction)**.** *Let P be a higher-arity $\mathbb{FDNC}$ program. Let $ld(P)$ be the set of constants occurring in the local positions of atoms in P. Given a set of constants C, every rule r' that results from r by substituting each variable occurring in a local position of some atom of r with some $c \in C$ is a* parameter-ground *instance of r w.r.t. C; the set of all such r' is denoted by $gr(r, C)$. 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, $red(P)$, results from P by replacing each g-unary atom $A(t_1, \dots, t_n)$ (resp., g-binary atom $R(t_1, \dots, t_n)$) occurring in $pgr(P)$ with an atom $A_{t_1,\dots,t_{n-1}}(t_n)$ (resp., $R_{t_1,\dots,t_{n-2}}(t_{n-1}, t_n)$). Similarly, the* $\mathbb{FDNC}$-reduction of an interpretation *I for P is defined as $red(I) = \{A_{t_1,\dots,t_{n-1}}(t_n) \mid A(t_1, \dots, t_n) \in I,\ A \in \mathcal{N}_1\} \cup \{R_{t_1,\dots,t_{n-2}}(t_{n-1}, t_n) \mid R(t_1, \dots, t_n) \in I,\ R \in \mathcal{N}_2\}$.*

The following result is then not difficult to establish.

Theorem 3.87. *Let P be a higher-arity $\mathbb{FDNC}$ program. Then an interpretation I is a stable model of P iff $red(I)$ is a stable model of $red(P)$.*

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

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

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

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

Due to the syntax of higher-arity $\mathbb{FDNC}$, given any (Herbrand) interpretation I of P, a rule $r \in P^I$ contains a non-proper atom iff it contains a non-proper atom in the body. Hence, every $J \in MM(P^I)$ such that $J \subseteq I$ must satisfy $J \subseteq patoms(P)$. Let P' result from $\mathsf{Ground}(P)$ by deleting each rule that contains some atom $A \notin patoms(P)$. Then $MM(P^I) = MM(P'^I)$ holds. This implies that $SM(P) = SM(P')$. Moreover, only proper atoms can be justified in stable models of P, i.e., $I \subseteq patoms(P)$ holds for each $I \in SM(P)$. Trivially, $SM(P') = \{I \mid red(I) \in SM(red(P'))\}$.

On the other hand, it is easily seen that $red(P') = \mathsf{Ground}(red(pgr(P)))$. Since $SM(\mathsf{Ground}(red(pgr(P))) = SM(red(pgr(P))))$, we obtain that $I \in SM(P)$ iff $red(I) \in SM(red(pgr(P)))$, as claimed. $\square$

Since $red(P)$ is finite, higher-arity $\mathbb{FDNC}$ programs inherit decidability from ordinary $\mathbb{FDNC}$ programs. The standard reasoning tasks can be decided by employing the parameter-grounding of a program and the algorithms for $\mathbb{FDNC}$ and its fragments. In general, $red(P)$ is of size exponential in the size of P, and the complexity of the higher-arity versions of fragments of $\mathbb{FDNC}$ is unavoidably higher by one exponential w.r.t. the parameter-free case (recall Table 3.1, which summarizes our results for the fragments of ordinary $\mathbb{FDNC}$). 2-EXPTIME-hardness of consistency in higher-arity $\mathbb{FDNC}$ can be shown by encoding an alternating Turing machine operating in exponential space (recall that AEXPSPACE = 2-EXPTIME). Intuitively, each stable model of a higher-arity program can be viewed as a tree whose nodes are ordinary databases over constants. In case of unbounded arities, each such database may be of exponential size. Thus computations of the machine can be simulated by encoding exponentially long configurations as databases. The latter can be done using standard techniques (see, e.g., the EXPTIME-hardness proof of cautious inference in pure DATALOG [DEGV01], or alternatively the 2-EXPTIME-hardness proofs in [Grä99, CGK08]). In a similar manner, our reductions for proving PSPACE and EXPSPACE lower bounds can be lifted to EXPSPACE and 2-EXPSPACE in the higher-arity case. On the other hand, the hardness results for EXPTIME, CO-NEXPTIME and CO-NEXPTIME$^{\text{NP}}$, corresponding to P, co-NP and Σ_2^P in Table 3.1, follow from the complexity of ordinary function-free logic programs [DEGV01].

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. For this purpose, we enhance $\mathbb{FDNC}$ programs with “strong” negation $\neg P(\vec{x})$ [GL91], which is expressed in the core language as usual: view $\neg P$ as a fresh predicate symbol and add constraints $\leftarrow P(\vec{x}), \neg P(\vec{x})$.

Example 3.88 (adapted from [EFL$^+$04])**.** *We assume that initially we have 3 blocks a, b, and c. In the initial state, a and b are on the table ($table$), while c is on top of a. This is formalized by the following facts:*

$$\begin{array}{rrr} Block(a,0) \leftarrow & On(a, table, 0) \leftarrow & Loc(table, 0) \leftarrow \\ Block(b,0) \leftarrow & On(b, table, 0) \leftarrow & \\ Block(c,0) \leftarrow & On(c, a, 0) \leftarrow & \end{array}$$

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:

$$\begin{aligned} Block(z,y) &\leftarrow Block(z,x), Change(x,y)\\ Loc(z,y) &\leftarrow Loc(z,x), Change(x,y)\\ Loc(z,x) &\leftarrow Block(z,x)\\ Occupied(z,x) &\leftarrow On(z_1,z,x), Block(z,x) \end{aligned}$$

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:

$$\begin{aligned} On(y,z,move(y,z,x)) &\leftarrow Block(y,x), Loc(z,x), Change(x, move(y,z,x))\\ \neg On(y,z',move(y,z,x)) &\leftarrow Block(y,x), Loc(z',x),\\ & Change(x, move(y,z,x)), On(y,z',x), Neq(z,z')\\ On(y,z,x') &\leftarrow On(y,z,x), Change(x,x'), not\ \neg On(y,z,x') \end{aligned}$$

We use an inequality predicate $Neq(x,y)$ over parameters, which we axiomatize by adding for each distinct $c_1, c_2 \in ld(P)$ 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.

$$Change(x, move(y,z,x)) \vee \neg Change(x, move(y,z,x)) \leftarrow Block(y,x), Loc(z,x)$$

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 actions simultaneously.

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:

$$\begin{aligned} \neg Change(x, move(y,z,x)) &\leftarrow Occupied(y,x)\\ \neg Change(x, move(y,z,x)) &\leftarrow Occupied(z,x) \end{aligned}$$

We ask 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 constructed higher-arity program. 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 $On(b, table, 0) \lor On(b, c, 0) \leftarrow$, 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 allow to encode (fragments of) the predicate version of the action language $\mathcal{K}$, but omit further discussion here.

3.7 Discussion

In line with efforts to pave the way for effective Answer Set Programming with function symbols, we presented $\mathbb{FDNC}$ programs as a decidable class of disjunctive logic programs with function symbols under stable model semantics. $\mathbb{FDNC}$ and its subclasses 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 (cf. [HJ99, EFL$^+$04, Rin04])

Furthermore, we have characterized the complexity of reasoning in $\mathbb{FDNC}$ programs, which is summarized in Table 3.1. $\mathbb{FDNC}$ and its subclasses provide *effective syntax* for expressing problems in PSPACE, EXPTIME, and EXPSPACE using logic programs with function symbols. 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 of a programs using *knots* from the maximal founded knot set of the program. The finite representation technique also allowed us to define elegant decision procedures for brave reasoning and cautious entailment of open queries in $\mathbb{FDNC}$. The technique 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.

$\mathbb{FDNC}$ and, in particular, the finite representation of stable models is a promising basis for developing algorithms that answer more complex queries than those considered in this thesis. Since $\mathbb{FDNC}$ easily captures some basic DLs, the algorithms developed for $\mathbb{FDNC}$ may be applicable also in other domains. In general, query answering algorithms need to examine a set of models in order to answer the query. Algorithms using knot sets as 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. This was already applied to answering conjunctive queries over description logic knowledge bases [OŠE08b, OŠE08a].

An implementation of $\mathbb{FDNC}$ programs is a subject of future work. However, since stable knots are defined as stable models of local programs (which are finite propositional disjunctive logic programs), the implementation will certainly export 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 [EIST05] featuring external function calls (by which limited Skolemization could be simulated), may be attractive for this. Another direction is to consider reducing reasoning in $\mathbb{FDNC}$ to reasoning in other fragments of programs with function symbols for which implementations exists, e.g., to *finitely ground* programs which are implemented in the DLV-COMPLEX system [CCIL08a].

We note here that $\mathbb{FDNC}$ programs can be seen as a fragment of finitely recursive programs [Bon04].[5] First, by employing Proposition 3.4 we can eliminate rules (R2) and (R4) by replacing them with polynomially many instances of (R3), (R5) and (R7), obtaining an equivalent program. As easily seen, for an $\mathbb{FDNC}$ program P with no rules of type (R2) and (R4), each atom in $A \in \mathcal{HB}_P$ depends only on finitely many other atoms in $\mathcal{HB}_P$.

As a limitation of $\mathbb{FDNC}$ programs we observe that they do not allow to propagate information from children to parents in the forest-shaped stable models. For example, rules $A(x) \leftarrow R(x, f(x)), B(f(x))$ or $A(x) \leftarrow A(f(x))$ are not allowed in $\mathbb{FDNC}$. In the context of planning, this prohibits reasoning about the past, e.g., the values of fluents in the past cannot be changed. This also bars us from a natural encoding of description logics with *inverse relations* (cf. [BCM$^+$03]). While the rules above do not alter the forest-shaped model property, they break finite recursiveness and testing minimality in such programs 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 develop *bidirectional* programs in the next chapter.

[5] Finitely recursive programs as defined in [Bon04] are normal programs. However, the property of finite recursivness is independent from the presence of disjunction.

Chapter 4

Bidirectional Programs

$\mathbb{FDNC}$ programs, which were introduced in the previous chapter, address many of the problems considered in this thesis. In particular, the presence of function symbols allows to generate arbitrarily large (but tree-shaped) structures and to reason about them by employing the power of the stable model semantics. Recall that $\mathbb{FDNC}$ programs are finitely recursive [BBC09] (see Section 3.7 for a discussion), and thus in $\mathbb{FDNC}$ programs an atom R can be derive only from atoms that are structurally not more complex than R (e.g., the rule $A(x) \leftarrow A(f(x))$ is not allowed). Attractively, finite recursivness allows stable models to be built in stages. In the case of $\mathbb{FDNC}$, we do this by employing knots as model building blocks.

However, finite recursiveness of $\mathbb{FDNC}$ programs also implies some limitations. We can see immediately that in reasoning about actions, $\mathbb{FDNC}$ programs allow to talk naturally about the future, but not the past; there are no means to propagate information from terms to their subterms. We can also identify a more general problem. $\mathbb{FDNC}$ does not allow to impose finiteness of stable models. That is, $\mathbb{FDNC}$ does not provide means to filter out infinite stable models of a program. This is a limitation since in certain application domains one is interested in arbitrarily large but still finite structures. For instance, in a planning domain this could correspond to filtering out infinite action sequences. The above limitation of $\mathbb{FDNC}$ is genuine: there does not exist an $\mathbb{FDNC}$ program P with the following properties: (a) P has infinitely many stable models, and (b) each stable model of P is finite. This follows from the characterization of stable models via knot sets. P has a finite knot set $\mathbb{K}_P$ from which the stable models of P can be generated. Since P has infinitely many stable models, the successor relation in $\mathbb{K}_P$ must have a cycle. Due to such a cycle one can build an infinite stable model for P.

In this chapter we present *bidirectional ($\mathbb{BD}$) programs*, which circumvent the above limitations of $\mathbb{FDNC}$ by allowing for atoms to also be inferred from structurally more complex ones; e.g., rules $A(x) \leftarrow B(f(x))$ are allowed and thus finite recursiveness is broken in general. $\mathbb{BD}$ programs allow to talk about both the future and the past, and to elegantly require finiteness of stable models.

As in the case of $\mathbb{FDNC}$, the class of $\mathbb{BD}$ programs is defined using syntactic restrictions, which modularly apply on the rules, are easy to test, and ensure that the stable models of a program are tree-shaped. However, bidirectionality of atom dependencies makes recognizing the stable models much more complicated. To address this, we provide a semantic characterization of stable models of $\mathbb{BD}$ programs in terms of specially labeled trees. Based on it, we present algorithms for the basic reasoning tasks, including consistency testing, and brave/cautious entailment of ground/existential queries. The algorithms are different from those for $\mathbb{FDNC}$; we use automata-theoretic methods. On the down side, our automata approach is less direct compared to model construction via knots for $\mathbb{FDNC}$ programs. On the positive side, automata running on infinite trees is a well-explored field with many results, which, in the end, allow us to arrive at optimal complexity results for $\mathbb{BD}$ programs.

To ease the development of algorithms, we work on *core* $\mathbb{BD}$ programs, which are a subset of full $\mathbb{BD}$ programs; each $\mathbb{BD}$-program can be transformed into a core program. Furthermore, any $\mathbb{FDNC}$ program can be encoded in polynomial time into a core program while preserving correspondence between stable models. It turns out that the aforementioned reasoning tasks are 2EXPTIME-complete for core $\mathbb{BD}$ programs. Thus $\mathbb{BD}$ programs are not only more expressive than $\mathbb{FDNC}$, they are also provably harder in terms of complexity.

As a means to decrease the complexity, we consider syntactic restrictions. We show that the complexity can be reduced by restricting the number of function symbols, disallowing disjunction, or limiting recursion via a restriction that we call *function-safeness*. For normal core $\mathbb{BD}$ programs, the complexity drops to EXPTIME-completeness and is thus in line with the complexity of reasoning in $\mathbb{FDNC}$ programs. If only one function symbol is allowed in a core $\mathbb{BD}$ program, the complexity drop to completeness for EXPSPACE and PSPACE in the disjunctive and normal case, respectively. For function-safe programs the complexity ranges from NEXPTIME to NP depending on the presence of disjunction and the number of function symbols in a program.

The rest of this chapter is organized as follows. In Section 4.1 we introduce full $\mathbb{BD}$-programs and core programs. Since reasoning over $\mathbb{BD}$-programs can be reduced to reasoning over core programs, we concentrate on core programs. In Section 4.2 we develop an automata-based method to reason over normal core programs, and in Section 4.3 we extend it to the disjunctive case. We then proceed to other syntactic restrictions in Section 4.4. In Section 4.5 we discuss our results.

4.1 Bidirectional Programs

We now introduce $\mathbb{BD}$ programs and discuss some possible applications for them, including an encoding of $\mathbb{FDNC}$ programs. Afterwards, we define *core* $\mathbb{BD}$ programs, which are as expressive as full $\mathbb{BD}$ programs, but allow for an easier presentation of algorithms.

We start by introducing the atoms that are allowed in $\mathbb{BD}$ programs.

Definition 4.1. *Let X be a designated variable and c be a designated constant. An atom $R(t, s_1, \ldots, s_n)$ is called a* $\mathbb{BD}$-atom *if*

(i) $t = X$, $t = c$, or $t = f(X)$ for some function symbol f;

(ii) for all $1 \leq i \leq n$, s_i is either a constant or a variable $y \neq X$.

Example 4.2. *Let y, z be variables, and d, e be constants. Then*

- *$R(X, e, d, y)$ and $P(f(X), e, d, y)$ are $\mathbb{BD}$-atoms;*
- *$R(g(X), z, X)$ is not a $\mathbb{BD}$-atom because X is allowed to occur in the first position only;*
- *$R(X, d, f(y))$ is not an $\mathbb{BD}$-atom because functional terms can appear in the first position only.*

$\mathbb{BD}$ programs are built from $\mathbb{BD}$-atoms.

Definition 4.3. *($\mathbb{BD}$ programs) A* $\mathbb{BD}$-program *P consists of rules r of the form*

$$A_1 \vee \ldots \vee A_n \leftarrow B_1, \ldots, B_m, not\ C_1, \ldots, not\ C_k$$

such that each atom in r is a $\mathbb{BD}$-atom. Furthermore, we assume that each rule r in a $\mathbb{BD}$ program is safe*, i.e., each of its variables occurs in* $\mathsf{body}^+(r)$.

The above syntactic restrictions can be explained as follows. The allowed functional terms are only of the form $f(X)$ and they can only occur in the first position of an atom. The condition (ii) in Definition 4.1 also ensures that an application of a rule can never transfer a ground functional term from the first position to another position. Using the same arguments as for higher-arity $\mathbb{FDNC}$, we can obtain the following:

Proposition 4.4. *If I is a stable model of a $\mathbb{BD}$-program P, then every atom in I is of the form $R(t, c_1, \ldots, c_n)$, where $c_1, \ldots, c_n$ are constants and t is of the form $f_n(\ldots f_1(c)\ldots)$.*

Note that the above property allows to view stable models of P as labeled trees. Indeed, the set of all terms $f_n(\ldots f_1(c)\ldots)$ forms a tree with root c, where a node $f(t)$ is a child of t. Thus, any I can be seen as a labeled tree in which each $R(t, c_1, \ldots, c_n) \in I$ is associated to the node t (see Figure 4.1).

Consider the following program, which is a safe variant of a program in [BBC09], originally due to F. Fages [Fag94].

Example 4.5. *Assume the program P consisting of the following rules:*

$$\begin{array}{llll} (1) & D(c) \leftarrow, & (3) & Q(x) \leftarrow Q(f(x)), \\ (2) & D(f(x)) \leftarrow D(x), & (4) & Q(x) \leftarrow D(x), not\ Q(f(x)). \end{array}$$

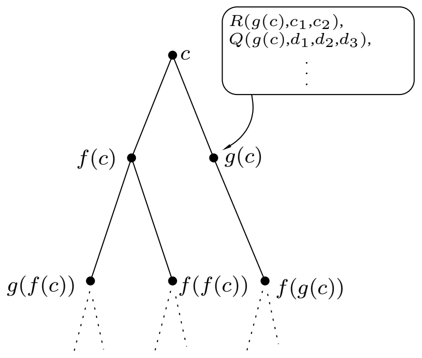


Figure 4.1: The tree-shaped structure of stable models of $\mathbb{BD}$ programs.

Observe that the program P is not *finitely recursive: due to the rule (3), the atom $Q(c)$ depends on the atoms $Q(f(c))$, $Q(f(f(c))), \ldots$, i.e., on infinitely many atoms (see the definition in [BBC09], which we recalled in Section 2.3). On the other hand, P is a $\mathbb{BD}$-program.*

We note that P is inconsistent. Indeed, any model I of P^I, by the rules (1) and (2), must contain $D(t)$ for each ground term $t \in \mathcal{HU}^P$. Furthermore, by the rules (3) and (4), I must also contain $Q(t)$ for each ground term $t \in \mathcal{HU}^P$. Therefore, P^I contains only ground instances of the rules (1)-(3). It follows that I is not a minimal model of P^I: indeed, removing all atoms $Q(t)$ from I would result in a model of P^I.

If we replace (4) by the rule $Q(x) \vee Q'(x) \leftarrow D(x), not\ Q(f(x))$, we obtain a consistent $\mathbb{BD}$-program with one stable model I_n for each natural number $n \geq 0$. Each I_n consists of:

- *$Q(f^i(c))$ for each $i < n$;*

- *$Q'(f^i(c))$ for each $i \geq n$;*

- *$D(f^i(c))$ for each $0 \leq i$.*

As we noted previously, rules in $\mathbb{BD}$ programs allow to impose finiteness of stable models:

Example 4.6. *(Finiteness filter) Let P be a $\mathbb{BD}$-program. Assume fresh unary predicates Dom, OK, and OK_f for each function symbol f of P. Take the program P' consisting of the following rules:*

1. *$Dom(X) \leftarrow A(X, y_1, \ldots, y_n)$ for all predicates A of P with arity $n+1$;*
2. *$OK(X) \leftarrow OK_{f_1}(X), \ldots, OK_{f_n}(X)$ where $\{f_1, \ldots, f_n\}$ is the set of function symbols in P;*

3. $OK_f(X) \leftarrow Dom(X), not\ Dom(f(X))$ *for each function* f *of* P*;*

4. $OK_f(X) \leftarrow OK(f(X))$ *for each function* f *of* P*;*

5. $\leftarrow not\ OK(c)$*.*

Using P' *we can filter out infinite stable models of* P*. Via the rule (1) we collect in* Dom *all the functional terms occurring in an interpretation. Since the stable models of* P *are always tree-shaped, it suffices to define rules to test for non-existence of an infinite branch. This is done via the rules (2-5). Using the rule (2) a node is marked as "good" if starting from any of its functional successors there is no infinite path. For the latter via (3-4) we define the* OK_f *predicate that is true for a node if it has no* f*-successors, or the* f*-successor is it self a "good" node. It is immediate to see that* $OK(c)$ *is motivated in an interpretation iff starting from* c *there is no infinite path.*

Take a finite stable model I *of* P*. Then*

$$I' = I \cup \{Dom(t), OK(t), OK_{f_1}(t), \ldots, OK_{f_n}(t) \mid t \textit{ occurs in } I\}$$

is a stable model $P \cup P'$*. On the other hand, any stable model* I' *of* $P \cup P'$ *restricted to the predicates of* P *is a stable model of* P*. Furthermore, each stable model of* $P \cup P'$ *must be finite because in an infinite interpretation* $OK(c)$ *cannot be proven. Thus the stable models of* $P \cup P'$ *are in one-to-one correspondence with the finite stable models of* P*.*

We note that $\mathbb{BD}$ programs can emulate $\mathbb{FDNC}$ programs. In fact, via $\mathbb{BD}$ programs we can extend $\mathbb{FDNC}$ with features of description logics with *inverse roles* (see [BCM+03]).

Example 4.7. *(Relations to* $\mathbb{FDNC}$*) We consider an extension of* $\mathbb{FDNC}$ *programs with rules of the form* $A(x) \leftarrow R(x, f(x)), B(f(x))$ *that, intuitively, allow for back-propagation of information in the forest-shaped stable models. The extension, which breaks finite recursiveness of* $\mathbb{FDNC}$*, allows for a direct reduction of consistency of knowledge bases in the description logic* $\mathcal{ALCI}$ *(*$\mathcal{ALC}$ *equipped with inverse roles) along the lines of Table 3.2.*

A program P *in the above fragment can be encoded using a* $\mathbb{BD}$*-program as follows. First, we can assume that* P *does not contain 2-variable rules of the form (R2) and (R4). Recall that rules of type (R2) can be replaced by quadratically many instances of rules (R3) and (R7). In the same manner, rules (R4) can be eliminated using (R5) and (R7).*

Consider the program P' *that is obtained by replacing in* P *each atom* $R(x, f(x))$ *with* $R_f(x)$*, where* R_f *is a fresh predicate name. It is immediate to see that the stable models of* P *and* P' *are in one-to-one correspondence.*

Observe that the program P' *may have two kinds of rules: rules in the syntax of* $\mathbb{BD}$ *programs, and rules of type (R7). In order to obtain a* $\mathbb{BD}$ *program, we transform* P'

into a program with only one designated constant c. We preserve the correspondence between stable models by encoding the 'graph' part of a stable model using c as an artificial root whose children correspond to the constants of the original program (see [EGOŠ08] for a similar correspondence-preserving encoding). Let G be the set of all function-free rules in $\mathsf{Ground}(P')$. *Note that G contains the rules (R7) of P', and is of size quadratic in the size of P. For the encoding we use a fresh unary predicate name A_d for each unary predicate A and constant d of P, and a fresh binary predicate $R_{d,e}$ for each binary R and each pair of constant d, e of P. First, remove from P' every rule $r \in G$ (note that P' is then a $\mathbb{BD}$ program with no facts). Second, for each $r \in G$ add to P' the rule r', where r' obtained by substituting each $A(d)$ in r by $A_d(c)$ and each $R(d, e)$ by $R_{d,e}(c)$. Finally, for each unary A and constant d of P, add to P' the pair $A(f_d(X)) \leftarrow A_d(X)$ and $A_d(X) \leftarrow A(f_d(X))$, where f_d is fresh. Intuitively, the last rules provide a bridge between the unary atoms encoded in the root and the children of the root that correspond to constants. Overall, we get a polynomial time encoding of extended $\mathbb{FDNC}$ programs into $\mathbb{BD}$ programs that preserves a one-to-one correspondence between stable models.*

Core Programs

To ease the presentation of our algorithms for $\mathbb{BD}$ programs, we work in the following on *core* programs. They capture the main features of full $\mathbb{BD}$ programs, and have the following properties: (a) all predicates are unary, (b) rules have at most one function symbol, (c) disjunction is only allowed in rules with no function symbols. In more detail, core programs are as follows:

Definition 4.8 (Core programs)**.** *A $\mathbb{BD}$-program P is a* core program *if it consists of* core rules*, which have the following forms:*

a) $A(c) \leftarrow$ *, where c is the special constant,*

b) $A(f(X)) \leftarrow B(X)$*, called f-*forward rule*,*

c) $A(X) \leftarrow B(f(X))$*, called f-*backward rule*, or*

d) $A_1(X) \vee \ldots \vee A_m(X) \leftarrow \mathit{not}\ B_1(X), \ldots, \mathit{not}\ B_n(X), C_1(X), \ldots, C_k(X)$*, called* local rule.

Core programs are structurally simple, but as expressive as full $\mathbb{BD}$ programs. Using a structural transformation, we can reshape a $\mathbb{BD}$-program P into a core program P' in such a way that the stable models of P and of P' are in correspondence.

Definition 4.9 (From full $\mathbb{BD}$ programs to core programs)**.** *Assume a $\mathbb{BD}$-program P. The core program* $\mathsf{core}(P)$ *is obtained from P in 3 steps as follows:*

(S1) *Replace each rule $r \in P$ by the set of rules G^r, where G^r is the set of all rules that can be obtained from r by replacing each variable other than X by a constant in P. Note that for any atom $S(t, \vec{v})$ occurring in P, $\vec{v}$ is a tuple of constants.*

(S2) *In each rule r of P replace each atom $S(t, \vec{v})$ by the atom $S_{\vec{v}}(t)$, where $S_{\vec{v}}$ is a fresh predicate name.*

(S3) *For each predicate name $S_{\vec{c}} \in \mathsf{preds}(P)$ and each function symbol f of P take a fresh unary predicate symbol $AUX_f^{S_{\vec{c}}}$ and rewrite P as follows:*

(a) *replace in P each occurrence of an atom $S_{\vec{c}}(f(X))$ by the atom $AUX_f^{S_{\vec{c}}}(X)$, and*

(b) *add the rules:*

$$AUX_f^{S_{\vec{c}}}(X) \leftarrow S_{\vec{c}}(f(X)), \text{ and}$$

$$S_{\vec{c}}(f(X)) \leftarrow AUX_f^{S_{\vec{c}}}(X).$$

The rewriting to a core program ensures a correspondence between stable models that leads us to the following result:

Proposition 4.10. *Assuming the number of variables in rules is bounded by a constant, a disjunctive (resp., normal) $\mathbb{BD}$-program P can be transformed in polynomial time into a disjunctive (resp., normal) core program P' such that P is consistent iff P' is consistent.*

Proof (Sketch). As easily seen, the partial grounding in step (S1) preserves equivalence. Indeed, due to Proposition 4.4, it suffices to concentrate on rules where functional terms occur only in the first position of an atom. Note that in case the number of variables in every rule of P is bounded by a constant, this rewriting step is feasible in polynomial time. The second step (S2) simply gives a separate predicate name $S_{\vec{v}}$ for each original predicate name S of arity $n+1$ and each n-tuple $\vec{c}$ of constants. This step preserves stable models under renaming of atoms, i.e., a ground atom $S_{\vec{v}}(t)$ corresponds to $S(t, \vec{v})$. Note that the resulting program has unary predicate symbols only. The step (S3) allows us to move out functional terms to separate rules. Overall, under bounded number of variables, the rewriting into $\mathsf{core}(P)$ is polynomial in the size of the original P.

To sum up, the translations leads to the following correspondence of stable models. Given an arbitrary stable model I of P, the corresponding stable model I' of $\mathsf{core}(P)$ is defined as $I' = I_1 \cup I_2$, where

- $I_1 = \{S_{\vec{c}}(t) \mid S(t, \vec{c}) \in I\}$, and
- $I_2 = \{AUX_f^{S_{\vec{c}}}(v) \mid S_{\vec{c}}(f(v)) \in I_1\}$.

On the other hand, given a stable model I of $\mathsf{core}(P)$, the corresponding stable model I' of P is obtained from I by removing all auxiliary atoms $AUX_f^{S_{\vec{c}}}(v)$ and replacing each $S_{\vec{c}}(t)$ by $S(t, \vec{c})$. $\square$

Reasoning Tasks

Recall that apart from consistency testing, we are interesting brave and cautious entailment of ground and existential queries. It is not hard to see that these tasks can be reduced in linear time to consistency testing in $\mathbb{BD}$ programs.

Assume a $\mathbb{BD}$-program P, a ground query $q_1 = A(\vec{w})$ and an existential query $q_2 = \exists \vec{x}.B(t, \vec{v})$, where all the variables in t and $\vec{v}$ are from $\vec{x}$. In line with the previous characterization for $\mathbb{FDNC}$, we assume q_1, q_2 do not have functional terms. It also suffices to restrict our attention to the case where $A(\vec{w})$ and $B(t, \vec{v})$ are $\mathbb{BD}$-atoms. Indeed, if $A(\vec{w})$ is not a $\mathbb{BD}$-atom, then by Proposition 4.4 it is false in any stable model of P. For $B(t, \vec{v})$, either it can be reshaped into a $\mathbb{BD}$-atom or it must be false by Proposition 4.4. If t is a constant $d \neq c$, then $P \not\models_b q_2$ and $P \not\models_c q_2$. If t is a variable $y \neq X$ that does not occur in $\vec{v}$, then just rename y to X. Otherwise, if y occurs twice, then replace every occurrence of y by c.

- For cautious reasoning, we can state q_1 and q_2 as constraints. That is, $P \models_c q_1$ iff $P \cup \{\leftarrow A(\vec{w})\}$ is inconsistent, and $P \models_c q_2$ iff $P \cup \{\leftarrow B(t, \vec{v})\}$ is inconsistent.
- For brave entailment of the ground query q_1, we can use a constraint with negation. As easily seen, $P \models_b q_1$ iff $P \cup \{\leftarrow \mathit{not}\ A(\vec{w})\}$ is consistent.
- For brave entailment of the existential query q_2, we can use a fresh unary predicate name C to track the existence of a proper variable assignment for q_2. If $t = c$, then $P \models_b q_2$ iff $P \cup \{\leftarrow \mathit{not}\ C(c), C(c) \leftarrow B(c, \vec{v})\}$ is consistent. If $t = X$, then $P \models_b q_2$ iff $P \cup \{\leftarrow \mathit{not}\ C(c), C(X) \leftarrow C(f(X)), C(X) \leftarrow B(X, \vec{v})\}$ is consistent.

Due to Proposition 4.10, under bounded number of variables query answering in $\mathbb{BD}$ programs reduces in polynomial time to checking consistency of a core program. Thus in the following we concentrate on consistency in core $\mathbb{BD}$ programs.

Unlike in $\mathbb{FDNC}$, the complexity of reasoning in normal and disjunctive core programs differs. In particular, consistency of normal core programs is EXPTIME-complete, while for disjunctive programs the complexity jumps by an exponential to completeness for 2-EXPTIME. We elaborate on this in the following two sections.

4.2 Consistency in Normal Core Programs

In this section, we develop an algorithm for testing consistency of normal core programs. Roughly, the presentation consists of two parts: the characterization of stable models via specially labeled trees, and the development of a tree automaton recognizing/generating such labeled trees.

In particular, we first introduce the notion of a *block tree*: this is a labeled tree that encodes a ground positive disjunction-free program together with an interpretation for it. We then define *minimal* block trees, which are the ones where the encoded interpretation coincides with the least model of the encoded program. Then each stable model I of a normal core program P can be seen as a minimal block tree where I is the encoded interpretation, while the encoded program equals the Gelfond-Lifschitz reduct P^I.

To provide an algorithm, we show that minimal block trees can be recognized using an alternating 2-way tree automata (2ATA) running over infinite trees (see Section 2.4 for the definition). By this we obtain that consistency of a normal core program can be reduced to checking nonemptiness of a 2ATA, and we also show that the resulting algorithm is worst-case optimal.

We note in advance that the characterization and the automaton for normal programs will be used latter to decide consistency of disjunctive programs. For this, we will exploit some special properties of disjunctive programs and employ transformations of the automaton developed here.

4.2.1 Minimal Block Trees

We next characterize stable models of normal core programs via labeled trees. To this end, we view each node in a tree as a term that can be constructed using the constant c and some unary function symbols. Recall that formally trees are subsets of $\mathbb{N}^*$, i.e., each tree is a set of words over natural numbers. To establish a correspondence between such words and terms, we assume that each function symbol f is indexed by $\mathsf{i}(f)$ and that the function i is bijective. We translate words into terms as follows:

Definition 4.11. *Let $w = k_1 \cdots k_n$ be a word over $\mathbb{N}$. We let*

$$\mathsf{term}(w) = f_n(\ldots f_1(c)\ldots),$$

where $\mathsf{i}(f_j) = k_j$ for every $j \in \{1, \ldots, n\}$ (note that $\mathsf{term}(\epsilon) = c$).

We can now define the labeling of trees. Each node (and thus, term) is a associated to a *block,* which essentially consists of two parts: a set of unary predicate names and a set of rules. Intuitively, the former stores the predicates satisfied by the term associated to a node, while the latter rules allow for "communication" between a node, its parent and its children.

Definition 4.12 (Block). *A* block *is any tuple $b = (\alpha, \mathcal{D}, \mathcal{R})$, where α is a constant or a function symbol, $\mathcal{D}$ is a set of unary predicates, and $\mathcal{R}$ is a set of positive disjunction-free core rules such that:*

(i) if α is a constant, then $\mathcal{R}$ has no f-backward rule for any function f, and

(ii) if α is a function symbol f, then for each g-backward rule in $\mathcal{R}$ we have $g = f$.

In case (i), b is a root block, *and in case (ii) b is a* child block.

Example 4.13. *Consider the following 3 blocks:*

$$b_1 = (c, \{A, C\}, \{A(x) \leftarrow; B(f(x)) \leftarrow A(x); D(g(x)) \leftarrow C(x)\}),$$
$$b_2 = (f, \{B\}, \{C(x) \leftarrow B(f(x))\}), \qquad b_3 = (g, \{D\}, \emptyset).$$

Note that b_1 is a root block, b_2, b_3 are child blocks; b_1 has 2 forward rules, b_2 has a backward rule, and that b_3 has no rules.

Intuitively, a root block can be used as a root of the tree and a child block inside the tree. In case (ii), the stored function symbol will correspond to the outermost function symbol of the term associated to a node.

In the following, whenever we talk about a set $\mathcal{B}$ of blocks we assume without loss of generality that the set of indices of all functions in $\mathcal{B}$ is an initial segment of the positive integers. We can now formally define block trees:

Definition 4.14 (Block tree). *Let $\mathcal{B}$ be a block set where k function symbols occur. Then a $\mathcal{B}$-*tree *is any k-ary $\mathcal{B}$-labeled tree $\mathcal{T} = (T, \mathcal{L})$ satisfying the following properness conditions:*

(i) $\mathcal{L}(\epsilon)$ is a root block, and

(ii) for all $x{\cdot}c \in T$, $\mathcal{L}(x{\cdot}c) = (\alpha, \mathcal{D}, \mathcal{R})$ is a child block with $\mathsf{i}(\alpha) = c$.

Example 4.15 (Cont'd). *For an example of a block tree, lets assume $\mathsf{i}(f) = 1$ and $\mathsf{i}(g) = 2$. Assume also the set of blocks $\mathcal{B} = \{b_1, b_2, b_3, b_f, b_g\}$, where b_1, b_2, b_3 are from Example 4.13 and $b_f = (f, \emptyset, \emptyset)$ and $b_g = (g, \emptyset, \emptyset)$.*

Take a binary tree T with labeling

- *$\mathcal{L}(\epsilon) = b_1$, $\mathcal{L}(1) = b_2$, $\mathcal{L}(2) = b_3$,*

- *$\mathcal{L}(y) = b_f$ for all $y \in T$ with $y = x'{\cdot}1$ and $x' \neq \epsilon$, and*

- *$\mathcal{L}(y) = b_g$ for all $y \in T$ with $y = x'{\cdot}2$ and $x' \neq \epsilon$.*

Then $\mathcal{T} = (T, \mathcal{L})$ is a $\mathcal{B}$-tree. Another $\mathcal{B}$-tree $\mathcal{T}'$ can be obtained, e.g., by setting $\mathcal{L}(11) = b_2$ instead of $\mathcal{L}(11){=}b_f$. The two $\mathcal{B}$-trees are graphically depicted in Figure 4.2.

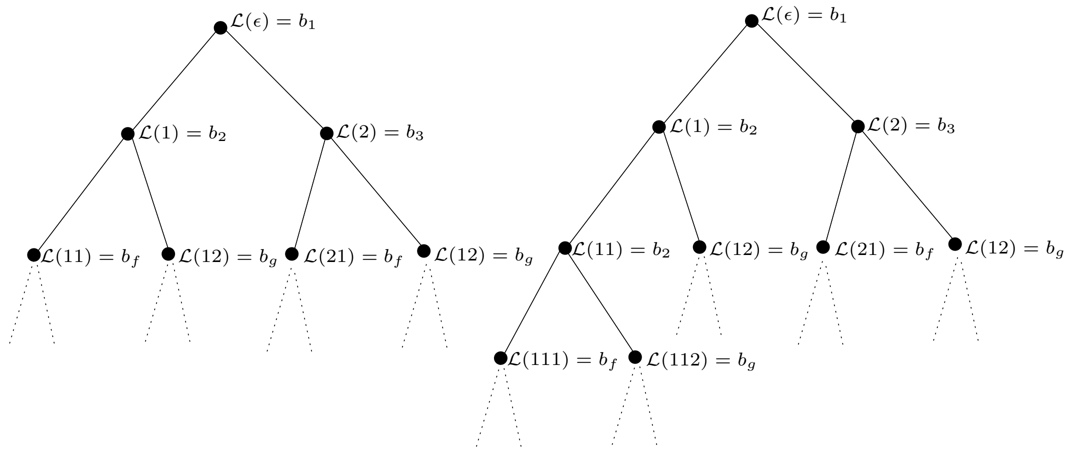


Figure 4.2: Block trees $\mathcal{T}$ (left) and $\mathcal{T}'$ (right) from Example 4.15.

As note previously, each block tree encodes an interpretation and a ground program. The former is obtained by transforming nodes into terms and predicate names into atoms.

Definition 4.16 (Associated interpretation)**.** *For a $\mathcal{B}$-tree $\mathcal{T} = (T, \mathcal{L})$, we define* its associated interpretation $\mathsf{int}(\mathcal{T})$ *as follows:*

$$\mathsf{int}(\mathcal{T}) = \{A(\mathsf{term}(x)) \mid x \in T \wedge \mathcal{L}(x) = (\alpha, \mathcal{D}, \mathcal{R}) \wedge A \in \mathcal{D}\}.$$

Example 4.17 (Cont'd)**.** *Recall the $\mathcal{B}$-trees $\mathcal{T}$ and $\mathcal{T}'$ from the previous example. We have:*

(i) $\mathsf{int}(\mathcal{T}) = \{A(c), C(c), B(f(c)), D(g(c))\}$*;*

(ii) $\mathsf{int}(\mathcal{T}') = \mathsf{int}(\mathcal{T}) \cup \{B(f(f(c)))\}$.

The ground program associated to a block tree is obtained in a similar manner as the associated program. For this, each node in a block tree is converted into a set of ground rules, which are in turn obtained by properly grounding the rules associated to the node.

Definition 4.18 (Associated program)**.** *For a core rule r, let $r_{\downarrow t}$ be the rule obtained by replacing in r each occurrence of X by t (note that X is the single variable in r and thus $r_{\downarrow t}$ is a ground rule). Then for a $\mathcal{B}$-tree $\mathcal{T} = (T, \mathcal{L})$* its associated program $\mathsf{prog}(\mathcal{T})$ *is the smallest program closed under the following rules:*

a) if $x \in T$, $\mathcal{L}(x) = (\alpha, \mathcal{D}, \mathcal{R})$, $r \in \mathcal{R}$, and r is a local or a forward rule, then $r_{\downarrow \mathsf{term}(x)} \in \mathsf{prog}(\mathcal{T})$;

b) *if $x \in T$, $\mathcal{L}(x) = (\alpha, \mathcal{D}, \mathcal{R})$, $r \in \mathcal{R}$, r is a backward rule, and $x = y{\cdot}c$ with $c \in \mathbb{N}$, then $r_{\downarrow \mathsf{term}(y)} \in \mathsf{prog}(\mathcal{T})$;*

Example 4.19 (Cont'd)**.** *Recall the $\mathcal{B}$-trees $\mathcal{T}$ and $\mathcal{T}'$ from the previous example. We have:*

(i) $\mathsf{prog}(\mathcal{T})=\{A(c) \leftarrow;\ B(f(c)) \leftarrow A(c);\ D(g(c)) \leftarrow C(c);\ C(c) \leftarrow B(f(c))\}$*;*

(ii) $\mathsf{prog}(\mathcal{T}') = \mathsf{prog}(\mathcal{T}) \cup \{C(f(c)) \leftarrow B(f(f(c)))\}$.

Note that for any block tree $\mathcal{T}$, the program $\mathsf{prog}(\mathcal{T})$ is positive and does not allow disjunction. Thus, if $\mathsf{prog}(\mathcal{T})$ is consistent, it has a unique minimal model, i.e., has the least model. Then *minimal* block trees are defined as follows:

Definition 4.20 (Minimal $\mathcal{B}$-tree)**.** *Given a $\mathcal{B}$-tree $\mathcal{T}$, we say $\mathcal{T}$ is* minimal *if $\mathsf{int}(\mathcal{T})$ is the least model of $\mathsf{prog}(\mathcal{T})$.*

Example 4.21 (Cont'd)**.** *Observe that $\mathcal{T}$ is minimal. On the other hand, $\mathcal{T}'$ is not minimal because $\mathsf{int}(\mathcal{T}) \subset \mathsf{int}(\mathcal{T}')$ and $\mathsf{int}(\mathcal{T})$ is a model of $\mathsf{prog}(\mathcal{T}')$.*

We can now turn to core programs. Stable models of a core program P can be seen as minimal block trees constructed using specifically selected blocks. In particular, we select the blocks in such a way that the program encoded in a block tree is the Gelfond-Lifschitz reduct of P with respect to the encoded interpretation. Clearly, via this each minimal tree encodes a stable model.

Definition 4.22. *A* block for a normal core program *P is any block $(\alpha, \mathcal{D}, \mathcal{R})$, where α is a constant or a function from P, $\mathcal{D}$ is a set of predicates of P, and $\mathcal{R}$ is a rule set consisting of:*

a) *All f-forward rules in P, for all functions f of P.*

b) *In case $\alpha = c$, the rule $A(x) \leftarrow$ for each fact $A(c) \leftarrow$ in P. Otherwise, if α is a function f, all f-backward rules of P.*

c) *(Reduct) For each local rule $r \in P$ such that $B \notin \mathcal{D}$ for all $B(x) \in \mathsf{body}^-(r)$, the rule $\mathsf{head}(r) \leftarrow \mathsf{body}^+(r)$.*

We arrive at the main result of this section:

Theorem 4.23. *If $\mathcal{B}$ is the set of all blocks for a normal core program P, then*

$$SM(P) = \{\mathsf{int}(\mathcal{T}) \mid \mathcal{T} \textit{ is a minimal } \mathcal{B}\textit{-tree}\}.$$

Therefore, P is consistent iff there exists a minimal $\mathcal{B}$-tree.

Proof. Let $\mathcal{B}$ be the set of all blocks for P. Due to the definition of block trees and blocks for P, we have $\mathsf{prog}(\mathcal{T}) = P^{\mathsf{int}(\mathcal{T})}$. Hence, if $\mathcal{T}$ is minimal, then $\mathsf{int}(\mathcal{T}) \in SM(P)$.

On the other hand, for any $I \in SM(P)$ we can build a minimal $\mathcal{B}$-tree $\mathcal{T}$ with $\mathsf{int}(\mathcal{T}) = I$. To see this, take an arbitrary $I \in SM(P)$. Take a k-ary tree T where k is the number of function symbols in P. For a ground rule r, let $r_{t \leftarrow v}$ be the rule that results after substituting t with v. Define the labeling function $\mathcal{L}$ that assigns to each $n \in T$ the block $b = (\alpha, \mathcal{D}, \mathcal{R})$ as follows:

- $\alpha = c$ in case $n = \epsilon$. Otherwise, if $|n| > 0$, $\alpha = \mathsf{i}^{-1}(s)$ where s is the last symbol in n.
- $\mathcal{D} = \{A \mid A(\mathsf{term}(n)) \in I\}$.
- Let $t = \mathsf{term}(n)$. Then $\mathcal{R}$ consists of:
  - (i) $r_{t \leftarrow X}$ for each rule $r \in P^I$ where all atoms in r have t as an argument (i.e., r originates from a local rule or a fact);
  - (ii) $r_{t \leftarrow X}$ for each rule $r \in P^I$ with $r = A(f(t)) \leftarrow B(t)$ for some A, B and f, i.e., r stems from an f-forward rule;
  - (iii) $r_{v \leftarrow X}$ for each rule $r = A(v) \leftarrow B(f(v))$ in P^I with $f(v) = t$ and $f = \alpha$.

Clearly, $\mathsf{int}(\mathcal{T}) = I$. It is also immediate to see that each assigned block is a block for P (i.e., they satisfy Definition 4.22) and that $\mathsf{prog}(\mathcal{T}) = P^I$. Finally, since I is a stable model of P, $\mathcal{T}$ is a minimal $\mathcal{B}$-tree. ☐

Via the above theorem, we can check consistency of a normal core program P by checking whether a minimal block tree can be constructed out of blocks for P. The latter can be checked by employing a tree-automaton that we build in the next section.

4.2.2 Generating Minimal Trees

In order to generate minimal block trees, we employ a characterization of minimal models via *proof trees*. In various forms, the characterization exists in the literature (e.g., in [MNR99, MR03]).

Proposition 4.24. *(Proof trees) I is the least model of a ground positive disjunction-free program P iff I is a model of P and for each $p \in I$ there exists a finite node-labeled tree $T_p = (N, V, L)$ satisfying the following conditions:*

1. *$L(\epsilon) = p$*
2. *for each node n, $L(p) \in I$.*

3. *for each node n, there is a rule $p \leftarrow q_1, \dots, q_n \in P$ such that $L(n) = p$ and $\{q_1, \dots, q_n\} = \{L(w) \mid w$ is a child of $n\}$.*

Let $\mathcal{B}$ be an arbitrary set of blocks. In this section we provide a method to decide whether some blocks from $\mathcal{B}$ can be arranged into a minimal $\mathcal{B}$-tree. We do this by defining a 2ATA $A^{\mathcal{B}}$ that accepts exactly the minimal $\mathcal{B}$-trees, and thus existence of a minimal $\mathcal{B}$-tree reduces to the nonemptiness test for $A^{\mathcal{B}}$ (please see Section 2.4 for the definition of 2ATAs).

We define the automaton $A^{\mathcal{B}} = \langle \Sigma, Q, \delta, q_0, F \rangle$ in stages, from the alphabet to the acceptance condition. To this end, let $\mathsf{rules}(\mathcal{B})$ and $\mathsf{preds}(\mathcal{B})$ denote the set of all rules and the set of all predicate names occurring in $\mathcal{B}$, respectively. Furthermore, let k be the number of function symbols occurring in $\mathcal{B}$. Recall that by Definition 4.14 all $\mathcal{B}$-trees are k-ary trees.

Clearly, the alphabet of $A^{\mathcal{B}}$ is $\Sigma = \mathcal{B}$. The set Q of states is provided in Table 4.1 together with a description of specific states. The transition function δ is defined next.

(Initial state) From the initial state q_0 the automaton switches to the states for testing consistency and minimality, and also for testing if the symbol at the root is indeed a root block and all the successors are proper child blocks. The switch is realized by setting for each $\sigma \in \Sigma$ the following transition from q_0:

$$\delta(q_0, \sigma) = (0, q^c) \wedge (0, q^j) \wedge (0, q_0^p) \wedge \bigwedge_{i=1}^{k} \big((i, q_i^p) \wedge (i, q^p)\big).$$

(Properness) To finalize the properness test we need two kinds of transitions. First of all, we define the behavior of the states $q_0^p, \dots, q_k^p$. The automaton fails if in the state q_0^p (resp., q_i^p, where $i > 0$) it reads a block that is not a root block (resp., not a child block with function f such that $\mathsf{i}(f) = i$). This is implemented by the following transition for each $\sigma = (\alpha, \mathcal{D}, \mathcal{R})$ in Σ and $1 \leq i \leq k$:[1]

$$\delta(q_0^p, \sigma) = [\alpha \text{ is a constant}],$$
$$\delta(q_i^p, \sigma) = [\alpha \text{ is a function with } \mathsf{i}(\alpha) = i].$$

Finally, we need to initiate the properness test at each descendant of the root.[2] This is done using the state q^p that is recursively propagated to all the descendants. For each $\sigma \in \Sigma$, define the following transition:

$$\delta(q^p, \sigma) = \bigwedge_{i=1}^{k} \big((i, q_i^p) \wedge (i, q^p)\big).$$

[1] $[E]$ stands for $\mathbf{true}$, if E evaluates to true, and else for $\mathbf{false}$.

[2] Note that for the root node the test is initiated in the transition from the initial state.

States		Description
q_0		Initial state
q^c		State for testing satisfaction of all the rules, i.e., ensuring that $\mathsf{int}(\mathcal{T})$ is a model of $\mathsf{prog}(\mathcal{T})$
q^c_r	for all $r \in \mathsf{rules}(\mathcal{B})$	States for testing satisfaction of specific rules
$q^{\in}_r$	for all $r \in \mathsf{rules}(\mathcal{B})$	Auxiliary states for testing whether a rule is contained in a block
$q^{\in}_A$, $q^{\notin}_A$	for all $A \in \mathsf{preds}(\mathcal{B})$	Auxiliary states for testing whether a predicate is present or absent in a block, resp.
q^j		State for testing whether all atoms are justified, i.e., ensuring that $\mathsf{int}(\mathcal{T})$ is the least model of $\mathsf{prog}(\mathcal{T})$
q^j_A	for all $A \in \mathsf{preds}(\mathcal{B})$	States for testing whether an atom with predicate name A is justified
q^p		State for testing properness of $\mathcal{T}$ (see Definition 4.14)
q^p_0		State for testing whether a block is a root block
$q^p_1, \ldots, q^p_k$		State for testing whether a block has function symbol f with $\mathsf{i}(f) = j$, for $1 \leq j \leq k$

Table 4.1: States of the automaton $A^{\mathcal{B}}$ running over a $\mathcal{B}$-tree $\mathcal{T}$.

(Consistency) The test for consistency is defined in 3 steps. First, via the test state q^c, the rules that need to be satisfied are selected, and the state q^c itself is propagated to the children. To this end, for every $\sigma = (\alpha, \mathcal{D}, \mathcal{R})$ in Σ we have:

$$\delta(q^c, \sigma) = \bigwedge_{r \in \mathcal{R}} (0, q^c_r) \wedge \bigwedge_{i=1}^{k} (i, q^c).$$

In the second step we move to testing the satisfaction of the selected rules. Recall that each block may have 3 kinds of rules: local, f-forward and f-backward rules. For every $\sigma \in \Sigma$ and $r \in \mathsf{rules}(\mathcal{B})$, we define:

$$\delta(q_r^c, \sigma) = \begin{cases} \bigvee_{i=1}^{n} (0, q_{B_i}^{\notin}) \vee (0, q_A^{\in}), & \text{if } r = A(x) \leftarrow B_1(x), \ldots, B_n(x), \\ (0, q_B^{\notin}) \vee (\mathsf{i}(f), q_A^{\in}), & \text{if } r = A(f(x)) \leftarrow B(x), \\ (0, q_B^{\notin}) \vee (-1, q_A^{\in}), & \text{if } r = A(x) \leftarrow B(f(x)). \end{cases}$$

Clearly, the rule r is satisfied, if either A is in the label of the node (resp., of some child or the parent), or some B_i (resp., B) is not. Finally, the test for containment of labels is as follows: for each $\sigma = (\alpha, \mathcal{D}, \mathcal{R})$ in Σ and each $A \in \mathsf{preds}(P)$ we have:

$$\delta(q_A^{\in}, \sigma) = [A \in \mathcal{D}], \qquad \delta(q_A^{\notin}, \sigma) = [A \notin \mathcal{D}].$$

(Minimality testing) Recall that by Proposition 4.24, we can test minimality by checking whether a finite justification tree exists for each atom in the interpretation. The latter can be done in a recursive manner: we choose a ground rule that fires the atom, and then try to find justification trees for all body atoms of the rule. This can be easily implemented in our automaton: apart from additional transitions this requires an acceptance condition ensuring finiteness of justification trees.

We use the state q^j to trigger the justification check for each predicate name in each node of the tree (this corresponds to all atoms in the encoded interpretation). For each $\sigma = (\alpha, \mathcal{D}, \mathcal{R})$ in Σ we define the following transition:

$$\delta(q^j, \sigma) = \bigwedge_{A \in \mathcal{D}} (0, q_A^j) \wedge \bigwedge_{i=1}^{k} (i, q^j).$$

Intuitively, when in state q^j, the automaton simultaneously enters the states q_A^j in order to find a justification for each predicate A in $\mathcal{D}$, and also propagates q^j to all the children.

For the next step, we let $M(A)$ denote the set of all tuples $\langle d, r, L\rangle$, where $d \in \{-1, 0, 1, \ldots, k\}$, $r \in \mathsf{rules}(\mathcal{B})$ has the predicate A in its head, and $L \subseteq \mathsf{preds}(\mathcal{B})$ such that:

- for local r, $d = 0$ and $L = \{B \mid B(x) \in \mathsf{body}(r)\}$;

- for f-forward r, $d = -1$ and $L = \{B \mid B(x) \in \mathsf{body}(r)\}$;

- for f-backward r, $d = \mathsf{i}(f)$ and $L = \{B \mid B(f(x)) \in \mathsf{body}(r)\}$.

Intuitively, we collect in $M(A)$ the predicates and the direction that provide the justification. When in a state q_A, the automaton guesses a tuple in $M(A)$ and proceeds to finding justifications for prescribed labels. This is done by defining the following transition for each $A \in \mathsf{preds}(\mathcal{B})$ and each $\sigma \in \Sigma$:

$$\delta(q^j_A, \sigma) = \bigvee_{(d,r,L)\in M(A)} \big((d, q^{\in}_r) \wedge \bigwedge_{B\in L} (d, q^j_B) \wedge (d, q^{\in}_B)\big).$$

Finally, we need the transition for the rule-containment state $q^{\in}_r$: for each symbol $\sigma = (\alpha, \mathcal{D}, \mathcal{R})$ in Σ and rule $r \in \mathsf{rules}(\mathcal{B})$, we have

$$\delta(q^{\in}_r, \sigma) = [r \in \mathcal{R}].$$

(Acceptance Condition) Observe that runs of $A^{\mathcal{B}}$ can have 2 types of infinite paths: (i) paths where exactly one of q^p, q^c, q^j occurs infinitely often, and (ii) paths where for some subset $\{A_1, \dots, A_n\} \subseteq \mathsf{preds}(\mathcal{B})$ only the justification states $q^j_{A_1}, \dots, q^j_{A_n}$ occur infinitely often. To ensure minimality, paths (ii) must be forbidden as they postpone justification indefinitely. This is done using the following parity acceptance condition

$$F = (\emptyset, \{q^p, q^c, q^j\}, Q).$$

By construction, the automaton $A^{\mathcal{B}}$ accepts $\mathcal{B}$-trees $\mathcal{T}$ such that $\mathsf{int}(\mathcal{T})$ is the least model of $\mathsf{prog}(\mathcal{T})$. We arrive at the desired result:

Proposition 4.25. *Given a set $\mathcal{B}$ of blocks, $A^{\mathcal{B}}$ accepts exactly the minimal $\mathcal{B}$-trees, i.e., $L(A^{\mathcal{B}})$ is the set of all minimal $\mathcal{B}$-trees.*

Using the above result we can characterize the complexity of consistency testing in normal core programs. Assume a normal core program P and the set $\mathcal{B}$ of all blocks for P (recall Definition 4.22). Observe the number of states in $A^{\mathcal{B}}$ is linear in $|P|$, and $\Sigma = \mathcal{B}$ is exponential in $|P|$. As easily seen, $|\mathcal{B}| = (k + 1) \times 2^{|\mathsf{preds}(P)|}$. Note that due to Definition 4.22, each $\mathcal{D} \subseteq \mathsf{preds}(P)$ induces a unique rule component in the block. Recall that due to Vardi's result, testing non-emptiness of 2ATAs is feasible in time exponential in the number of states, the out-degree of a tree and the index of the parity condition, and polynomial in the size of the input alphabet (Theorem 2.24). Then by Propositions 4.25 and 4.10, we get that checking consistency of P is feasible in exponential time in $|P|$. The matching lower-bound for the problem easily follows from the EXPTIME-hardness of $\mathbb{FDNC}$. Indeed, while preserving consistency and in polynomial time we can convert an $\mathbb{FDNC}$ program into a $\mathbb{BD}$ program with one variable per rule (recall Example 4.7), which in turn can be converted in a core $\mathbb{BD}$ program (see Proposition 4.10).

We arrive at the main complexity result of this section:

Theorem 4.26. *Testing consistency of normal core programs and of normal $\mathbb{BD}$ programs under bounded number of variables is* EXPTIME*-complete.*

Using Proposition 4.25 we can also obtain a worst-case optimal upper bound for the case where only one function symbol is allowed. In this case, the $\mathcal{B}$-trees for a program P degenerate to words over $\mathcal{B}$, and the automaton $A^{\mathcal{B}}$ is a 2-way alternating word automaton. Checking nonemptiness of $A^{\mathcal{B}}$, and thus checking consistency of P, is feasible in space polynomial in the size of P. For this we view $A^{\mathcal{B}}$ as a Büchi automaton (see, e.g., [Tho90]). We simply replace the parity acceptance condition $F = (\emptyset, \{q^p, q^c, q^j\}, Q)$ by the set $F = \{q^p, q^c, q^j\}$. The latter Büchi condition has exactly the same effect as the original parity condition. In particular, every infinite path of a run must have an infinite number of occurrences of q^p, q^c or q^j. This effectively prohibits paths where some justification state q^j_A, $A \in \mathsf{preds}(P)$, occurs infinitely often. Thus the replacement of the acceptance condition preserves the accepted language. Using the translation in [KPV01], the automaton $A^{\mathcal{B}}$ can be translated into an equivalent nondeterministic 1-way Büchi word automaton A'. The translation preserves the alphabet but the size of state set in A' is exponential in the number of states in $A^{\mathcal{B}}$, i.e., overall, A' is of size exponential in the size of P. It is well-known that nonemptiness of nondeterministic 1-way Büchi word automata is feasible in NL [VW94] (see also [Tho90]). Thus emptiness of $A^{\mathcal{B}}$ can be decided in polynomial space by running the algorithm in [VW94] on the automaton A'. We note that A' does not need to be built explicitly (which would require exponential space): the emptiness algorithm can be supplied with the relevant parts of A' within polynomial space.[3]

Theorem 4.27. *In case only 1 function symbol is allowed, testing consistency of normal core programs, and of normal $\mathbb{BD}$ programs under bounded number of variables, is* PSPACE*-complete.*

The matching lower bound can be obtained by a reduction from a word problem for Turing Machines with polynomially bounded space. The reduction is almost identical to the one for $\mathbb{FC}$ programs (see Lemma 3.69), and is thus omitted.

4.3 Consistency in Disjunctive Core Programs

We analyze here the disjunctive case, and extend the method of the previous section to disjunctive core programs. To this end, we first characterize the minimal models of positive disjunctive ground programs in terms of *split programs*.

Definition 4.28 (Split)**.** *Let I be an interpretation for a positive (disjunctive) ground program P. A non-disjunctive positive program P' is called a* split of P w.r.t. I *if P' results from P by*

(a) replacing each rule $r \in P$ such that $|\mathsf{head}(r) \cap I| \geq 1$ with a rule $h \leftarrow \mathsf{body}(r)$, where $h \in \mathsf{head}(r) \cap I$ is picked arbitrarily, and

[3]This is a standard observation that is often omitted in the literature.

(b) replacing each rule $r \in P$ such that $|\mathsf{head}(r) \cap I| = \emptyset$ with the constraint $\leftarrow \mathsf{body}(r)$.

By $SP(P, I)$ we denote the set of all splits of P w.r.t. I.

Intuitively, a split P' is obtained from P in two steps. First, in each rule where one or more head atoms are true, we leave only one such atom and delete the rest. Then the rules with no head atoms true in I are transformed into constraints. We can then characterize the minimal models of disjunctive ground programs as follows.

Theorem 4.29. *For any positive disjunctive ground program P it holds that $I \in MM(P)$ iff I is the least model of every $P' \in SP(P, I)$.*

Proof. For the "only if" case, observe that if $I \in MM(P)$, then I is also a model of every $P' \in SP(P, I)$, and that an arbitrary model of P' is also a model of P.

For the "if" case, suppose I is the least model of every $P' \in SP(P, I)$ and $I \notin MM(P)$. As already observed, I is a model of P. Hence, there must exist another model $J \subset I$ of P. Simply build a split P' of P w.r.t. I in two steps:

1. replace each rule $r \in P$ such that $|\mathsf{head}(r) \cap I| \geq 1$ with a rule $h \leftarrow \mathsf{body}(r)$, where
   (i) $h \in \mathsf{head}(r) \cap I$, if $\mathsf{head}(r) \cap J = \emptyset$, and
   (ii) $h \in \mathsf{head}(r) \cap J$, if $\mathsf{head}(r) \cap J \neq \emptyset$;
2. replace each rule $r \in P$ such that $|\mathsf{head}(r) \cap I| = \emptyset$ with the constraint $\leftarrow \mathsf{body}(r)$.

As easily seen, J is a model of P', and thus I is not the least model of P'. Contradiction. ☐

Due to the theorem above, minimal model checking for positive disjunctive programs reduces to minimal model checking over a set of non-disjunctive programs. Building on this, we show decidability of $\mathbb{BD}$ programs using trees whose nodes are labeled with *sets of blocks* (or *hyperblocks*) instead of a single block. Intuitively, each *projection* of such a tree, obtained by arbitrarily discarding all but one block in each node, provides us with a $\mathcal{B}$-tree that encodes a different single split of a program. Consistency testing for a program then amounts to finding a tree whose all projections are minimal $\mathcal{B}$-trees.

We first formally define the notion of projection of a tree.

Definition 4.30 (Projection). *Let Σ be an alphabet, and let $\Sigma' \subseteq 2^{\Sigma}$. A tree $(T, \mathcal{L})$ over Σ is called a* Σ-projection of *a tree $(T, \mathcal{L}')$ over Σ', if for every node $n \in T$, $\mathcal{L}(n) \in \mathcal{L}'(n)$.*

Trees having block trees as projections are defined next.

Definition 4.31 (Hyperblock). *A* hyperblock *is any set h of blocks obeying the following: $(\alpha_1, \mathcal{D}_1, \mathcal{R}_1) \in h$ and $(\alpha_2, \mathcal{D}_2, \mathcal{R}_2) \in h$ imply $\alpha_1 = \alpha_2$ and $\mathcal{D}_1 = \mathcal{D}_2$.*

In other words, a hyperblock is any set of blocks $(\alpha, \mathcal{D}, \mathcal{R})$ sharing the same α and $\mathcal{D}$.

Definition 4.32 (Hyperblock tree). *Let $\mathcal{B}$ be a set of blocks with k function symbols occurring in it, and let $\mathcal{H} \subseteq 2^{\mathcal{B}}$ be a set of hyperblocks. Then an $\mathcal{H}$-*tree *is any $\mathcal{H}$-labeled k-ary tree $\mathcal{T} = (T, \mathcal{L})$ satisfying the following conditions:*

(i) the blocks in $\mathcal{L}(\epsilon)$ are root blocks, and

(ii) for all $x{\cdot}c \in T$, the blocks in $\mathcal{L}(x{\cdot}c)$ are child blocks $(\alpha, \mathcal{D}, \mathcal{R})$ with $\mathsf{i}(\alpha) = c$.

Note that $\mathcal{B}$-projections of $\mathcal{T}$ above are (proper) $\mathcal{B}$-trees. We let $\mathsf{int}(\mathcal{T}) = \mathsf{int}(\mathcal{T}')$ for an arbitrary $\mathcal{B}$-projection $\mathcal{T}'$ of $\mathcal{T}$ (observe that for any other projection $\mathcal{T}''$, $\mathsf{int}(\mathcal{T}'') = \mathsf{int}(\mathcal{T}')$).

Definition 4.33 (Minimal hyperblock tree). *Let $\mathcal{T}$ be an $\mathcal{H}$-tree with $\mathcal{H} \subseteq 2^{\mathcal{B}}$. We say $\mathcal{T}$ is* minimal *if each $\mathcal{B}$-projection of $\mathcal{H}$ is minimal, i.e., is a minimal $\mathcal{B}$-tree.*

To characterize stable models of disjunctive core programs via hyperblock trees, we need to select suitable blocks and hyperblocks. The blocks are similar to the ones for normal core programs, except that disjunctive rules are represented via splits.

Definition 4.34. *A* block for a (disjunctive) core program *P is any block $(\alpha, \mathcal{D}, \mathcal{R})$, where α is the constant c or a function of P, $\mathcal{D}$ is a set of predicates of P, and $\mathcal{R}$ is a rule set consisting of:*

a) All f-forward rules in P, for all functions f of P.

b) In case $\alpha = c$, the rule $A(x) \leftarrow$ for each fact $A(c) \leftarrow$ in P. Otherwise, if α is a function f, all f-backward rules of P.

c) (Reduct) Assuming $Z = \{A(X) \mid A \in \mathcal{D}\}$, for each local rule $r \in P$ such that $\mathsf{body}^-(r) \cap Z = \emptyset$:

(i) a rule $h \leftarrow \mathsf{body}^+(r)$ for an arbitrary $h \in \mathsf{head}(r) \cap Z$, in case $\mathsf{head}(r) \cap Z \neq \emptyset$, and

(ii) the constraint $\leftarrow \mathsf{body}^+(r)$, in case $\mathsf{head}(r) \cap Z = \emptyset$.

A hyperblock for *P is any $\subseteq$-maximal set of blocks $(\alpha, \mathcal{D}, \mathcal{R})$ for P sharing the same α and $\mathcal{D}$.*

The following characterization is now an easy consequence of Theorem 4.29 and the above Definitions 4.33 and 4.34.

Theorem 4.35. *Let P be a disjunctive core program, let $\mathcal{B}$ be the set of all blocks for P, and let $\mathcal{H} \subseteq 2^{\mathcal{B}}$ be the set of all hyperblocks for P. Then*

$$SM(P) = \{\mathsf{int}(\mathcal{T}) \mid \mathcal{T} \textit{ is a minimal } \mathcal{H}\textit{-tree}\}.$$

Proof. Let $\mathcal{T}$ be a minimal $\mathcal{H}$-tree and $I = \mathsf{int}(\mathcal{T})$. Then I is the least model of $\mathsf{prog}(\mathcal{T}')$ for each $\mathcal{B}$-projection $\mathcal{T}'$ of $\mathcal{T}$. Furthermore, due to the definition of hyperblocks for P, $\{\mathsf{prog}(\mathcal{T}') \mid \mathcal{T}' \text{ is a } \mathcal{B}\text{-projection of } \mathcal{T}\} = SP(P^I, I)$. Thus, I is the least model of every $P' \in SP(P^I, I)$, and, by Theorem 4.29, a minimal model of P^I and a stable model of P.

On the other hand, using Theorem 4.29, for any $I \in SM(P)$ we can easily build an $\mathcal{H}$-tree $\mathcal{T}$ with $\mathsf{int}(\mathcal{T}) = I$. This is done along the lines of the construction in the proof of Theorem 4.23. □

Using the above theorem, consistency of disjunctive core programs reduces to finding a minimal hyperblock tree. To decide the latter, we will resort to the automaton of the previous section. Recall that, given any set $\mathcal{B}$ of blocks, the automaton $A^{\mathcal{B}}$ accepts exactly the minimal $\mathcal{B}$-trees (Proposition 4.25). Thus using the above Theorem 4.35 stable models of disjunctive core programs can be characterized as follows:

Theorem 4.36. *Let P be a (disjunctive) core program and let I an interpretation for P. Furthermore, assume $\mathcal{B}$ is the set of all blocks for P, and $\mathcal{H} \subseteq 2^{\mathcal{B}}$ is the set of all hyperblocks for P. Then I is a stable model of P iff there exists an $\mathcal{H}$-tree $\mathcal{T}$ such that:*

(i) $\mathsf{int}(\mathcal{T}) = I$*, and*

(ii) $A^{\mathcal{B}}$ *accepts each* $\mathcal{B}$*-projection of* $\mathcal{T}$*.*

Using the above theorem and the automaton $A^{\mathcal{B}}$ from the previous section we can obtain an automata-based algorithm for consistency testing in disjunctive core programs. As we shall see next, by reshaping $A^{\mathcal{B}}$ using automata transformations we can decide consistency of a program in double exponential time in the size of the program. We dedicate the next Section 4.3.1 to proving that the obtained upper bound is worst-case optimal: the hardness part is shown by an encoding of an alternating Turing machine with exponentially bounded space.[4]

Theorem 4.37. *Testing consistency of disjunctive core programs is in* 2EXPTIME.

Proof. Let P be a core program, $\mathcal{B}$ be the set of all blocks for P, and $\mathcal{H} \subseteq 2^{\mathcal{B}}$ be the set of all hyperblocks for P. By Theorem 4.36, P is consistent iff there exists an $\mathcal{H}$-tree $\mathcal{T}$

[4] We note here that 2EXPTIME-hardness can be proven already for brave queries over disjunctive core programs without negation.

such that $A^{\mathcal{B}}$ accepts each $\mathcal{B}$-projection of $\mathcal{T}$. We transform $A^{\mathcal{B}}$ into a tree automaton A' with the language

$$L(A') = \{\mathcal{H}\text{-tree } \mathcal{T} \mid A^{\mathcal{B}} \text{ accepts each } \mathcal{B}\text{-projection of } \mathcal{T}\}.$$

Clearly, P is consistent iff $L(A') \neq \emptyset$. We build A' as follows:

(1) We first transform $A^{\mathcal{B}}$ into a 2ATA A_1 that accepts the complement of $L(A^{\mathcal{B}})$, i.e., $L(A_1)$ is the set of all k-ary trees $\mathcal{T}$ over $\mathcal{B}$ such that $\mathcal{T} \notin L(A^{\mathcal{B}})$. This is done in the standard way: the connectives in the transitions are inverted and the index of the sets in the parity condition increased by one. The translation preserves the states, the alphabet, and is linear in the size of the input automaton.

(2) We then transform A_1 into an equivalent nondeterministic 1-way tree automaton (1NTA) A_2 using the translation in [Var98]. The translation is exponential in the size of A_1. We have an exponential blow-up in the number of states, the alphabet is preserved, while the acceptance condition increases linearly. Thus the state set of A_2 is of at most exponential size in the size of P. In contrast to 2ATAs, the automaton A_2 moves only forward and its transitions are disjunctions of conjunctions (see Section 2.4 for more details).

(3) Building on A_2, we define a 1NTA $A_3 = \langle \Sigma_3, Q_3, \delta_3, q_0, F_3 \rangle$ that accepts exactly the k-ary trees $\mathcal{T}$ over $\mathcal{H}$ such that some $\mathcal{B}$-projection $\mathcal{T}'$ of $\mathcal{T}$ is not accepted by $A^{\mathcal{B}}$. The components of A_3 are the ones of A_2 except the alphabet and the transition relation. We set $\Sigma_3 = \mathcal{H}$, $Q_3 = Q_2$ and $F_3 = F_2$. For each state $q \in Q_3$ and symbol $\sigma \in \Sigma_3$, the transition function δ_3 is defined as follows:

$$\delta_3(q, \sigma) = \bigvee_{\alpha \in \sigma} \delta_2(q, \alpha).$$

Intuitively, when scanning an $\mathcal{H}$-labeled tree $\mathcal{T}$, A_3 simulates a run of A_2 on some $\mathcal{B}$-projection $\mathcal{T}'$ of $\mathcal{T}$. A_3 accepts $\mathcal{T}$ iff A_2 accepts some $\mathcal{T}'$. Note that the alphabet $\mathcal{H}$ is of size exponential in the size of P (observe that each set of predicate names from P together with a function or the constant c induces one hyperblock). Thus A_3 can be obtained in time single exponential in the size of P.

(4) By complementing A_3 using the same method as in step (1), we obtain a 1ATA A_4 that accepts a tree $\mathcal{T}$ over $\mathcal{H}$ iff $A^{\mathcal{B}}$ accepts each $\mathcal{B}$-projection of $\mathcal{T}$. A_4 is the desired automaton A'.

To sum up, in time exponential in the size of P we can build the desired automaton A'. The number of states and the alphabet in A' are exponential in the size of P, while the size of the parity condition is bounded by constant. Emptiness of a 1ATA is decidable in exponential time in the number of states, the out-degree of the tree and the index of the acceptance condition, and polynomial time in the size of the alphabet (see Theorem 2.24). Overall, this yields double exponential time in the size of P. ☐

As consequence of the above construction, we also have an EXPSPACE upper bound for disjunctive core programs P that allow for 1 function symbol only. Indeed, for such a program P, the automaton A' above is an alternating 1-way word automaton. For such an automaton, nonemptiness can be decided in space polynomial in the size of A' (see, e.g., [Ser06]). Since the alphabet and the state set of A' are exponential in $|P|$, while the parity condition has bounded size, consistency of P can be decided in exponential space. The matching lower bound is developed in the next section.

Theorem 4.38. *Testing consistency of disjunctive core programs with one function symbol only is in* EXPSPACE.

4.3.1 Lower Bound: 2EXPTIME-hardness of Disjunctive Core Programs

In this section we provide a matching lower bound for the 2-EXPTIME upper bound in Theorem 4.37. For this we develop a reduction from the word problem for an alternating Turing Machine with an exponentially bounded tape size into the problem of consistency in disjunctive core programs.

Recall that an alternating Turing Machine (see Definition 2.15) is given by a tuple

$$M = (Q_\exists, Q_\forall, \Sigma, q_0, \delta),$$

where $Q_\exists$ is a set of existential states containing the accepting state q_{accept} and the rejecting state q_{reject}, $Q_\forall$ is a set of universal states, Σ is an alphabet containing the blank symbol $\llcorner\!\lrcorner$, q_0 is the initial state, and

$$\delta \subseteq Q \times \Sigma \times Q \times \Sigma \times \{+1, 0, -1\}$$

is the transition relation, where $Q = Q_\exists \cup Q_\forall$.

In the following, for an alternating M with an exponential space bound and an input word w, we construct in polynomial time a disjunctive core program $P_{M,w}$ such that M accepts w iff $P_{M,w}$ is consistent.

To simplify the presentation, we without loss of generality assume the following:

- For both existential and universal states there are always 2 successive configurations. We can thus assume that $\delta = \delta_r \cup \delta_l$ such that for each pair σ, q, one transition is read from δ_r and another from δ_l, i.e., formally, in δ_r and δ_l the first two components form a key.

- We also assume an existential state always leads to a universal state, and vice versa.

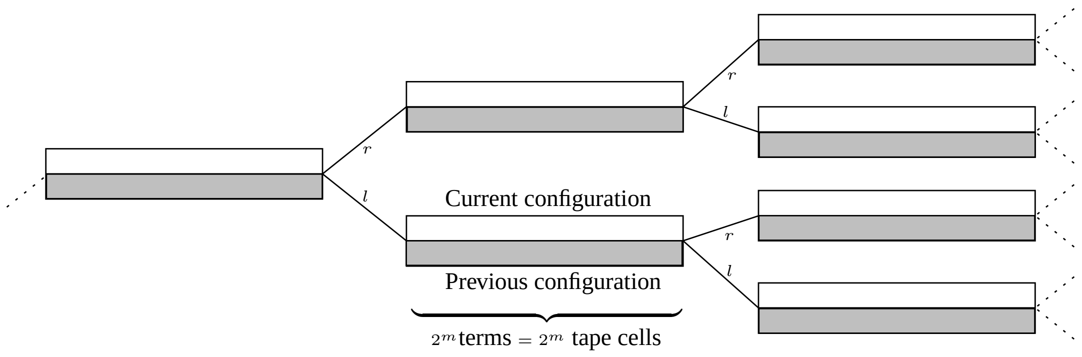


Figure 4.3: Computations of M on w.

- The initial state q_0 is existential.
- With each transition M moves the RW head to the left or to the right, i.e., for each $(\alpha, s, \alpha', s', d) \in \delta$, $d \in \{-1, +1\}$.

Let $p(\cdot)$ be a polynomial such that $2^{p(|w|)}$ bounds the space used by M running on w. We let $m = p(|w|)$. Observe that each position of the tape (fragment used by M on w) can now be identified by an m-bit address.

We represent the computations of M on w in models of $P_{M,w}$. We use three function symbols f, l and r. Roughly, each configuration of M is encoded using a sequence of 2^m terms of the form $f(t)$, while the functions l and r are used for splitting into two successive configurations. See Figure 4.3 for this structure. For technical reasons, along with the configuration of M we also store its previous configuration.

We start by defining rules to generate the tree-shaped structure depicted in Figure 4.3.

Basic structure. We will use a marker (the predicate $Break$) to separate the exponentially long term sequences corresponding to configurations (the latter "configuration" terms will be identified using the predicate C). To identify particular cells of M, we use m-bit addresses. In particular, we employ unary predicate names $B_1^1, \ldots, B_m^1$ and $B_1^0, \ldots, B_m^0$ to represent each possible address. Intuitively, B_i^b, where $b \in \{1, 0\}$, means that the value of the ith bit in the address is b.

We start by marking the constant c with $Break$, i.e., label it as a break point. Each break point initiates an exponentially long sequence of configuration terms. To this end, the term that follows a break point is given the initial address value $\underbrace{0,\ldots,0}_{m}$:

$$\begin{aligned}
Break(c) &\leftarrow && && (4.1)\\
C(f(X)) &\leftarrow Break(X), && && (4.2)\\
B_i^0(f(X)) &\leftarrow Break(X), && i \in \{1,\ldots,m\} && (4.3)
\end{aligned}$$

To generate the required sequences of configuration terms, we will implement a counter that counts up to $2^m - 1$. For this, we use unary predicates $inv_1^1, \ldots, inv_m^1$ and $inv_1^0, \ldots, inv_m^0$ to perform addition. Intuitively, inv_i^1 tells us that the ith bit must be inverted, while inv_i^0 means that the value states the same. For $b \in \{1,0\}$, let $\bar{b} = 1$ if $b = 0$, and $\bar{b} = 0$ otherwise.

Depending on the counter value (address of the node), the decision is made on which bits have to be inverted to obtain the next address. To this end, for each $i \in \{1,\ldots,m-1\}$ and $b \in \{1,0\}$, we add the following:

$$\begin{aligned}
inv_m^1(X) &\leftarrow C(X), && (4.4)\\
inv_i^b(X) &\leftarrow C(X), B_{i+1}^1(X), inv_{i+1}^b(X), && (4.5)\\
inv_i^0(X) &\leftarrow C(X), B_{i+1}^0(X). && (4.6)
\end{aligned}$$

In case the first bit need not be inverted, the value of the counter is not $2^m - 1$, and hence we continue counting. Otherwise, we split into two branches and restart counting. We also use predicates L and R to keep track in which branch (left or right) we are in.

$$\begin{aligned}
C(f(X)) &\leftarrow inv_1^0(X), && (4.7)\\
Break(l(X)) &\leftarrow inv_1^1(X), && (4.8)\\
Break(r(X)) &\leftarrow inv_1^1(X), && (4.9)\\
L(l(X)) &\leftarrow inv_1^1(X), && (4.10)\\
L(f(X)) &\leftarrow L(X), C(f(X)), && (4.11)\\
R(r(X)) &\leftarrow inv_1^1(X), && (4.12)\\
R(f(X)) &\leftarrow R(X), C(f(X)). && (4.13)
\end{aligned}$$

Using the following rules we define the address of the follow-up configuration term. For each $i \in \{1,\ldots,m\}$ and $b \in \{1,0\}$ we add:

$$\begin{aligned}
B_i^{\bar{b}}(f(X)) &\leftarrow C(f(X)), B_i^b(X), inv_i^1(X), && (4.14)\\
B_i^b(f(X)) &\leftarrow C(f(X)), B_i^b(X), inv_i^0(X), && (4.15)
\end{aligned}$$

Configurations. In each of the segments of configuration terms, we encode two configurations of M: the current and the previous one. Let $\mathcal{Z} = \Sigma \times (Q \cup \{nil\})$. For all

$L \in \mathcal{Z}$, we assume predicate names Z_L and Z'_L, where Z_L is used for the previous and Z'_L for the current configuration. Intuitively, $Z'_{(\alpha,q)}(t)$ is true for a configuration term t iff in the current configuration the cell corresponding to t has symbol α, and, in case q is a state, the machine's RW head is over the symbol and the machine is in state q. We generate the two configurations as follows:

$$\bigvee_{L\in\mathcal{Z}} Z_L(X) \leftarrow C(X); \qquad \bigvee_{L\in\mathcal{Z}} Z'_L(X) \leftarrow C(X); \tag{4.16}$$

We note here that above rules do not rule out incorrect configurations, e.g., where the RW head is simultaneously in two different positions. We will later define rules to deal with this.

Initial configuration. In the initial configuration of M, the tape consists of the input word w, the machine is in state s_0 and the RW head is on the first symbol of w. Assume w is of the form $w = a_1 \cdots a_n$ where each $a_i \in \Sigma$. Let $Blank$ be a new predicate symbol which intuitively corresponds to $\llcorner\!\lrcorner$, and let $f^i(s)$ denote $f_i(\ldots f_1(s)\ldots)$. We add the following rules to $P_{M,w}$:

$$\begin{aligned}
Z'_{a_1,s_0}(f(c)) &\leftarrow & & \qquad (4.17)\\
Z'_{a_i,nil}(f^i(c)) &\leftarrow & i \in \{2,\ldots,n\} & \qquad (4.18)\\
Blank(f^{n+1}(s)) &\leftarrow & & \qquad (4.19)\\
Blank(f(X)) &\leftarrow\ C(f(X)), Blank(X) & & \qquad (4.20)\\
Z'_{\llcorner\!\lrcorner,nil}(X) &\leftarrow\ Blank(X) & & \qquad (4.21)
\end{aligned}$$

The rules above write the input symbols into their positions, while the rest of the tape is filled with blank symbols.

Transitions. We can now describe the transitions of M: the current configuration is obtained by a transition from the previous one. Recall that, by assumption, for any configuration we have exactly two successive configurations, where each one of them is given in δ_l and δ_r.

For each $(\alpha, s, \alpha', s', d) \in \delta_l$ we add:

$$Z'_{(\alpha',s')}(f(X)) \leftarrow L(X), C(X), Z_{(\alpha,s)}(X), \quad \text{if } d = +1, \tag{4.22}$$
$$Z'_{(\alpha',s')}(X) \leftarrow L(f(X)), C(f(X)), Z_{(\alpha,s)}(f(X)), \quad \text{if } d = -1. \tag{4.23}$$

Similarly as above, for each $(\alpha, s, \alpha', s', d) \in \delta_r$ we add:

$$Z'_{(\alpha',s')}(f(X)) \leftarrow R(X), C(X), Z_{(\alpha,s)}(X), \quad \text{if } d = +1, \tag{4.24}$$
$$Z'_{(\alpha',s')}(X) \leftarrow R(f(X)), C(f(X)), Z_{(\alpha,s)}(f(X)), \quad \text{if } d = -1. \tag{4.25}$$

Acceptance. We can now define rules to deal with acceptance. Recall that the initial state s_0 is assumed to be existential, and that an existential state always leads to a universal, and vice versa. To reflect this, we label each path in our computation tree using fresh predicates $Exists$ and $ForAll$ as follows. Starting with the constant c, we label nodes on the path with $Exists$. When a break point occurs, we switch to labeling using $ForAll$; after the next break point we switch back to $Exists$. Such alternation is repeated forever.

$$\begin{aligned}
Exists(c) &\leftarrow && (4.26)\\
Exists(f(X)) &\leftarrow Exists(X), && (4.27)\\
ForAll(f(X)) &\leftarrow ForAll(X), && (4.28)\\
Exists(l(X)) &\leftarrow ForAll(X), Break(l(X)), && (4.29)\\
Exists(r(X)) &\leftarrow ForAll(X), Break(r(X)), && (4.30)\\
ForAll(l(X)) &\leftarrow Exists(X), Break(l(X)), && (4.31)\\
ForAll(r(X)) &\leftarrow Exists(X), Break(r(X)). && (4.32)
\end{aligned}$$

Intuitively, if a break point is labeled with $Exists$ (resp., $ForAll$), then the configurations that follows the break point has an existential (resp., universal) state.

We use the following rules to check the existence of an accepting run:

$$\begin{aligned}
Accept(X) &\leftarrow Z_{\alpha,q_{accept}}(X), \qquad \text{for all } \alpha \in \Sigma, && (4.33)\\
Accept(X) &\leftarrow Accept(f(X)), && (4.34)\\
Accept(X) &\leftarrow Exists(X), Accept(l(X)), && (4.35)\\
Accept(X) &\leftarrow Exists(X), Accept(r(X)), && (4.36)\\
Accept(X) &\leftarrow ForAll(X), Accept(l(X)), Accept(r(X)). && (4.37)
\end{aligned}$$

The above mirrors the acceptance condition for alternating Turing machines (see Definition 2.15).

We are done with the first part of the encoding. It is easy to see that M accepts w if and only if there exists a minimal model I of the above program such that:

(i) $Accept(c) \in I$;

(ii) In each fragment of configuration terms, the "current" configuration (stored via the Z'_L predicates) coincides with the "previous" configuration (stored via the Z_L predicates) in the two successive fragments of configuration terms.

We next define additional rules to filter out the stable models violating (i) or (ii). For this we will use the *saturation* method.

Comparing configurations. To find errors in configurations, we employ new predicate names $A^b_1, \dots, A^b_m$, where $b \in \{1, 0\}$, and add for each $i \in \{0, \dots, m\}$ the following rules:

$$A^0_i(r(X)) \vee A^1_i(r(X)) \leftarrow \mathit{Break}(r(X)) \tag{4.38}$$
$$A^0_i(l(X)) \vee A^1_i(l(X)) \leftarrow \mathit{Break}(l(X)) \tag{4.39}$$

Via the above rules, at a break point that is inside the computation tree (i.e., c is ignored) we guess some address $addr$. Recall that each such break point is the end of a sequence of configuration terms, and is also the beginning of another sequence. Our aim now is to check whether in the two sequences the two terms sharing the address $addr$ do not violate (ii). First, the address is broadcast to the two sequences.

We broadcast the A^b_i labels to the follow-up configuration. For all $i \in \{1, \dots, m\}$ and $b \in \{1, 0\}$ we add the following rules:

$$A^b_i(f(X)) \leftarrow \mathit{Break}(X), A^b_i(X), C(f(X)) \tag{4.40}$$
$$A^b_i(f(X)) \leftarrow A^b_i(X), C(f(X)). \tag{4.41}$$

To broadcast the A^b_i labels to the predecessor configuration, we employ fresh predicates names $A^{b,l}_1, \dots, A^{b,l}_m$ and $A^{b,r}_1, \dots, A^{b,r}_m$, where $b \in \{1, 0\}$. We need two copies because each configuration has two successive break points, and we must know from which of them a propagated address originates. For each $i \in \{1, \dots, m\}$ and $b \in \{1, 0\}$, we add the following rules:

$$A^{b,l}_i(X) \leftarrow \mathit{Break}(l(X)), A^b_i(l(X)), C(X) \tag{4.42}$$
$$A^{b,l}_i(X) \leftarrow A^{b,l}_i(f(X)), C(X) \tag{4.43}$$
$$A^{b,r}_i(X) \leftarrow \mathit{Break}(r(X)), A^b_i(r(X)), C(X) \tag{4.44}$$
$$A^{b,r}_i(X) \leftarrow A^{b,r}_i(f(X)), C(X) \tag{4.45}$$

The next step is to identify the term in the successive configuration that has the guessed address:

$$EQ_i(X) \leftarrow A^b_i(X), B^b_i(X) \tag{4.46}$$
$$EQ(X) \leftarrow EQ_1(X), \dots, EQ_m(X) \tag{4.47}$$

We can similarly identify the term in the previous configuration:

$$EQ^l_i(X) \leftarrow A^{b,l}_i(X), B^b_i(X) \tag{4.48}$$
$$EQ^l(X) \leftarrow EQ^l_1(X), \dots, EQ^l_m(X) \tag{4.49}$$
$$EQ^r_i(X) \leftarrow A^{b,r}_i(X), B^b_i(X) \tag{4.50}$$
$$EQ^r(X) \leftarrow EQ^r_1(X), \dots, EQ^r_m(X) \tag{4.51}$$

The identified terms send back the required content to the break-point for comparison.

For the term in the successive sequence this is done using a fresh predicate SZ_L and the following rules for all $L \in \mathcal{Z}$:

$$\begin{aligned} SZ_L(X) &\leftarrow EQ(X), Z_L(X) && (4.52)\\ SZ_L(X) &\leftarrow SZ_L(f(X)), C(f(X)) && (4.53)\end{aligned}$$

For the term in the previous sequence, we use fresh predicates PZ^l_L and PZ^r_L for each $L \in \mathcal{Z}$. To propagate the Z'_L label of the identified term we add the following rules for all $L \in \mathcal{Z}$:

$$\begin{aligned} PZ^l_L(X) &\leftarrow EQ^l(X), Z'_L(X) && (4.54)\\ PZ^l_L(f(X)) &\leftarrow PZ^l_L(X), C(f(X)) && (4.55)\\ PZ^l_L(l(X)) &\leftarrow PZ^l_L(X), Break(l(X)) && (4.56)\\ PZ^r_L(X) &\leftarrow EQ^r(X), Z'_L(X) && (4.57)\\ PZ^r_L(f(X)) &\leftarrow PZ^r_L(X), C(f(X)) && (4.58)\\ PZ^r_L(r(X)) &\leftarrow PZ^r_L(X), Break(r(X)) && (4.59)\end{aligned}$$

We can now compare the received content. This is done via the following rules:

$$\begin{aligned} NoError(X) &\leftarrow PZ^l_L(X), SZ_L(X), Break(X) && (4.60)\\ NoError(X) &\leftarrow PZ^r_L(X), SZ_L(X), Break(X) && (4.61)\end{aligned}$$

We now employ the saturation trick to make sure that the comparison of labels is made for all possible addresses. For this we add the following rules:

$$\begin{aligned} A^b_i(X) &\leftarrow NoError(X) \qquad \text{for all } i \in \{1, \ldots, m\} && (4.62)\\ Good(X) &\leftarrow A^1_1(X), \ldots, A^1_m(X), A^0_1(X), \ldots, A^0_m(X) && (4.63)\\ Good(c) &\leftarrow . && (4.64)\end{aligned}$$

The above rules have the following effect. Suppose I is a minimal model of the program constructed so far. Suppose $Break(t) \in I$ and $t \neq c$, i.e., t is an inner break-point. It is easy to see that $Good(t) \in I$ iff the "current" configuration in the term sequence leading to t coincides with the "previous" configuration in the term sequence departing from t. This is because for $Good(t)$ to be true in I we must have

$$\{A^1_1(t), \ldots, A^1_m(t), A^0_1(t), \ldots, A^0_m(t)\} \subseteq I.$$

The latter may be true only in case any choice of an address in the rules (4.38)- (4.39) leads to a proof $NoError(t)$.

Recall that the constant c is a break-point that starts the initial configuration: we not need to do a comparison for it, and thus (4.64) is added.

Using constraints we can now require $\mathit{Accept}(c)$ to be proven and $\mathit{Good}(t)$ to be proven for all break-points t:

$$\leftarrow \mathit{Break}(X), \mathit{not}\ \mathit{Good}(X) \tag{4.65}$$
$$\leftarrow \mathit{not}\ \mathit{Accept}(c) \tag{4.66}$$

This ends the definition of $P_{M,w}$. The last constraints ensure that each stable model of $P_{M,w}$ satisfies the requirement (i) and (ii), and hence we have:

Proposition 4.39. *M accepts w iff $P_{M,w}$ is consistent.*

The reduction above is polynomial in the size of M and w, and thus we obtain the desired lower bound. Using the upper bound in Theorem 4.37, we obtain the following:

Theorem 4.40. *Deciding consistency of disjunctive core programs is* 2-EXPTIME-*complete.*

Observe that negation was used in the rules (4.65) and (4.66) only. We can actually simulate the effect of these rules using positive rules and a brave query. Consider the program $P'_{M,w}$ obtained from $P_{M,w}$ by replacing (4.65) and (4.66) with the following rules:

$$\mathit{AllGood}(X) \leftarrow Z'_{\alpha,q_{accept}}(X), \quad \text{for all } \alpha \in \Sigma \tag{4.67}$$
$$\mathit{AllGood}(X) \leftarrow Z'_{\alpha,q_{reject}}(X), \quad \text{for all } \alpha \in \Sigma \tag{4.68}$$
$$\mathit{AllGood}(X) \leftarrow \mathit{AllGood}(f(X)), \tag{4.69}$$
$$\mathit{AllGood}(X) \leftarrow \mathit{AllGood}(l(X)), \mathit{Good}(l(X)), \tag{4.70}$$
$$\mathit{AllGood}(X) \leftarrow \mathit{AllGood}(r(X)), \mathit{Good}(r(X)), \tag{4.71}$$
$$\mathit{Goal}(c) \leftarrow \mathit{Accept}(c), \mathit{AllGood}(c). \tag{4.72}$$

Intuitively, $\mathit{AllGood}(c)$ is proven iff in all paths to the accepting/rejecting configuration we have that $\mathit{Good}(t)$ is true for all encountered break-points t.[5]

Proposition 4.41. *M accepts w iff $P'_{M,w} \models_b \mathit{Goal}(c)$.*

The alternative reduction remains polynomial in the size of M and w, and thus using the upper bound in Theorem 4.37 we obtain the following completeness result.

[5]For simplicity of presentation, existential and universal states of M are not distinguished here; that is, (ii) is ensured in all paths that follow a configuration with an existential state, although this is not strictly necessary.

Theorem 4.42. *Brave entailment of ground queries over disjunctive core programs without negation is* 2-EXPTIME-*complete.*

Recall that we used three function symbols f, r, l for the encoding. The symbols r and l were used for branching into two alternative configurations. In case we have only one function symbol available, with minor modifications the above construction degenerates to an encoding of a *deterministic* Turing Machine with exponentially bounded space. This reduction together with the upper bound in Theorem 4.38 gives us the following result:

Theorem 4.43. *Deciding consistency of disjunctive core programs that allow for one function symbol only is* EXPSPACE-*complete.*

4.4 Fragments of Bidirectional Programs

As we saw in the previous section, the complexity of reasoning in full $\mathbb{BD}$ programs is rather high: standard reasoning is complete for EXPTIME and 2-EXPTIME for normal and disjunctive programs, respectively. We saw also that restricting to the case of one function symbol leads to a decrease in complexity, i.e., to completeness for PSPACE and EXPSPACE, respectively. In this section we develop an alternative restriction that reduces the complexity, allows an implementation exploiting existing ASP reasoners, and does not prohibit interesting applications.

We present here *function-safe* $\mathbb{BD}$ programs that allow only for a limited recursion over term-introducing rules. The programs have only finite models of limited size, but are still expressive enough to facilitate reasoning involving, e.g., non-recursive data structures.

Function-safe core programs are presented first; we generalize the notions and results to full $\mathbb{BD}$ programs in the end of the section.

Definition 4.44 (function-safe core programs)**.** *Let P be a core program, and let G be a graph over* $\mathsf{preds}(P)$ *such that there is an arc $B \rightarrow A$ iff there is a rule $r \in P$ where B occurs positively in the body of r and A occurs in the head of r; we say the arc is* unsafe *if $r = A(f(x)) \leftarrow B(x)$ for some function symbol f.*

A predicate $R \in$ $\mathsf{preds}(P)$ *is* function-unsafe *if*

(a) R occurs in a cycle involving an unsafe arc, and

(b) R is reachable from $F \in$ $\mathsf{preds}(P)$ *for some fact $F(c) \leftarrow$ in P.*

Otherwise, if R is not function-unsafe, R is function-safe. *The program P is* function-safe, *if for each rule $r \in P$, the body of r contains a positive occurrence of a function-safe predicate.*

Function-safe core programs can be used for generating and processing finite tree-shaped structures. One possible application is processing of HTML or XML documents, which can be seen as finite node-labeled trees, where labels correspond to elements, attributes, etc. Rule-based languages have already been deployed for this purpose: e.g., [GK04a, GK04b] use *monadic* DATALOG to query HTML documents in order to extract content on the Web. Suppose $\mathcal{T} = (T, \mathcal{L})$ is a finite labeled tree over Σ with branching bounded by k, i.e., $T \subseteq \{1, \ldots, k\}^*$. Then $\mathcal{T}$, which may correspond to an HTML or XML document, can be represented in a function-safe core program as follows. We can use unary predicates $Node_n$ for each $n \in T$, $Label_\sigma$ for each $\sigma \in \Sigma$, and function symbols f_i for each $i \in \{1, \ldots, k\}$. Then T can be reconstructed using the fact $Node_\epsilon(c) \leftarrow$ and the rule $Node_{n \cdot i}(f_i(X)) \leftarrow Node_n(X)$ for each $n \in T$ and $i \in \{1, \ldots, k\}$ with $n \cdot i \in T$. The labeling function $\mathcal{L}$ can be expressed using the rule $Label_\sigma(X) \leftarrow Node_n(X)$ for each node $n \in T$ with $\sigma \in \mathcal{L}(n)$. We can now use additional rules to express a query over $\mathcal{T}$. For instance,

$$\begin{array}{lcll} Q(X) & \leftarrow & Label_\sigma(X), & \\ Q(X) & \leftarrow & Q(f_i(X)), & \quad\quad \text{for each } i \in \{1, \ldots, k\}, \end{array}$$

collects all the nodes in $\mathcal{T}$ that are labeled with σ or have a descendant labeled with σ. All monadic DATALOG queries (see [GK04a]) can be emulated in function-safe core programs, and also extended with disjunction and negation under the stable model semantics. Interestingly, our language also allows to deal with trees that may not be completely specified. Intuitively, using functional terms an incomplete $\mathcal{T}$ can be non-deterministically augmented to a full tree (conforming, e.g., to an XML Schema). One can then employ cautious inference to obtain certain answers to a query.

The key feature of function-safe core programs leading to a decrease in complexity is that only Herbrand interpretations over polynomially deep terms have to be considered when computing the stable models (as opposed to unbounded depth in the general case). To see this more formally, assume a function-safe core program P, and for a term $t = f_n(\ldots f_1(c) \ldots)$ from $\mathcal{HU}_P$, let $\mathsf{depth}(t) = n + 1$ (note $\mathsf{depth}(c) = 1$). It follows immediately from function-safeness that if $A(t) \in \mathcal{HB}_P$ is an atom and $\mathsf{depth}(t) > |\mathsf{preds}(P)|$, then $A(t) \notin I$ for any $I \in SM(P)$.

In other words, we have:

Proposition 4.45. *If I is a stable model of a function-safe core program P, then* $\mathsf{depth}(t) \leq |\mathsf{preds}(P)|$ *for each atom $A(t) \in I$.*

Recall the complexity of consistency testing in propositional ASP: the problem is NP-complete for normal programs and Σ_2^P-complete for disjunctive programs (see e.g. [DEGV01]). It follows from the above proposition that to check consistency of a function-safe core program P, it suffices to consider the restriction of $\mathsf{Ground}(P)$ to rules where the term depth is bounded by $|\mathsf{preds}(P)|$. This restriction is of size exponential in the size of P. Therefore, consistency of normal and disjunctive function-safe core programs can be decided in NEXPTIME and NEXPTIME$^{\mathrm{NP}}$, respectively. We will see next that in both cases we can do better: the two problems are complete for PSPACE and NEXPTIME, respectively.

We remark here that in the case when only one function symbol is allowed in a function-safe core program, the complexity consistency testing drops down to NP-completeness and Σ_2^P-completeness since the relevant subset of the grounding is only of polynomial size. The lower bound follows immediately from the complexity of ASP in the propositional case.

Proposition 4.46. *For normal (resp., disjunctive) function-safe core programs with one function symbol only, deciding consistency is* NP*-complete (resp.,* Σ_2^P*-complete).*

We note also that checking function-safety of a core program P can be decided in polynomial time. Clearly, traversing the bodies of rules in P requires only linear time. We can also check in polynomial time whether a given predicate A is function-safe. This involves ensuring the nonexistence of a cycle over an unsafe arc. That this is feasible in polynomial time follows from well-known results: that checking cyclicity of a directed graph is NL-complete (cf. [Jon75]), the fact that $\mathrm{NL} = \mathrm{coNL}$ [Imm88, Sze88], and the inclusion $\mathrm{NL} \subseteq \mathrm{P}$.

4.4.1 Normal Function-Safe Core Programs

We show that checking consistency of a normal function-safe core program is in PSPACE. For this, we again develop a characterization of stable models via specially labeled trees. However, instead of using a tree automaton as we did for the general case, we use an alternating algorithm running in polynomial time. Since $\mathrm{AP} = \mathrm{PSPACE}$, this gives us the PSPACE upper bound (see Section 2.2 for more details). For the tight lower bound, we develop an encoding of quantified Boolean formulas (QBFs).

Recall that stable models of $\mathbb{BD}$ programs are tree-shaped. However, even using only polynomially deep terms we can build exponentially large tree-shaped interpretations, and thus for the PSPACE bound we need to be careful in what we store in memory.

As a stepping stone towards the algorithm, we will use a characterization of minimal models via strict partial orders (transitive irreflexive relations) over atoms. In particular, we use the following proposition which can be found in various forms in the literature (e.g., [MNR99]).

Proposition 4.47. *I is the least model of a ground positive disjunction-free program P iff I is a model of P and there is a strict partial order $\prec$ over I such that for each $p \in I$ there exists a rule $p \leftarrow q_1, \ldots, q_n$ in P such that $q_i \prec p$ for each $i \in \{1, \ldots, n\}$.*

We next describe stable models of normal function-safe core programs using labeled trees that store an interpretation and an additional information to prove modelhood and also to construct a strict partial to prove minimality.

The tree-shaped structures on which our algorithm operates are as follows:

Definition 4.48. *(Term trees) Given a normal function-safe core program P, a* term tree *for P is a tuple $\mathcal{V} = \langle T, L, O, R \rangle$ where:*

(i) $T \subseteq \mathcal{HU}^P$ *is such that* $f(t) \in T$ *implies* $t \in T$;

(ii) for each $t \in T$, $\mathsf{depth}(t) \leq |\mathsf{preds}(P)|$;

(iii) L *assigns to each* $t \in T$ *a set* $L(t) \subseteq \mathsf{preds}(P)$ *of predicate names;*

(iv) O *assigns to each* $t \in T$ *a strict partial order over* $L(t)$;

(v) R *assigns to each pair* (t, A)*, where* $t \in T$ *and* $A \in L(t)$*, a positive core rule* $R(t, A)$ *(over the signature of* P*) with head predicate* A.

For $\mathcal{V}$ *we define the* corresponding interpretation $\mathsf{int}(\mathcal{V}) = \{A(t) \mid t \in T \wedge A \in L(t)\}$. *For each* $t \in T$*, we let* $\prec_t^{\mathcal{V}} = O(t)$. *Finally, we define the relation* $\dot{\prec}^{\mathcal{V}} \subseteq \mathsf{int}(\mathcal{V}) \times \mathsf{int}(\mathcal{V})$ *as* $\dot{\prec}^{\mathcal{V}} = \dot{\prec}_1 \cup \dot{\prec}_2 \cup \dot{\prec}_3$*, where:*

(i) $\dot{\prec}_1 = \{\langle B(t), A(t) \rangle \mid t \in T \wedge B \prec_t^{\mathcal{V}} A\}$;

(ii) $\dot{\prec}_2 = \{\langle B(t), A(f(t)) \rangle \mid f(t) \in T \wedge R(f(t), A) = A(f(X)) \leftarrow B(X)\}$;

(iii) $\dot{\prec}_3 = \{\langle B(f(t)), A(t) \rangle \mid f(t) \in T \wedge R(t, A) = A(X) \leftarrow B(f(X))\}$.

A term tree can intuitively be viewed as a tree where nodes are terms and each node is assigned a set of predicate names ordered by a strict partial order. Furthermore, each predicate name A at a node t is associated with a rule $R(t, A)$ which, intuitively, is a candidate rule to justify the presence of A at t. Each term tree $\mathcal{V}$ for a program P induces the interpretation $I = \mathsf{int}(\mathcal{V})$ and the relation $\dot{\prec}^{\mathcal{V}}$ over I. Importantly, we can now define simple conditions for term trees under which

(I) the interpretation I is actually a model of P^I,

(II) $\dot{\prec}^{\mathcal{V}}$ is a strict partial order (note that this does *not* follow per se from (iv) above), and

(III) for each $p \in I$ there exists a rule $p \leftarrow q_1, \ldots, q_n$ in P^I with $q_i \prec p$ for all $i \in \{1, \ldots, n\}$.

Note that, by Proposition 4.47, (I-III) imply I is a stable model of P. We elaborate on the conditions next.

Definition 4.49. *(Well-aligned term trees) We say a term tree $\mathcal{V} = \langle T, L, O, R\rangle$ for a normal function-safe core program P is* well-aligned, *if each term $t \in T$ is* good *in $\mathcal{V}$. A term $t \in T$ is* good *in $\mathcal{V}$ if the following (consistency, supportedness and well-foundedness) conditions are satisfied:*

(C1) if $t = c$, then $A \in L(t)$ for each $A(c) \leftarrow$ of P;

(C2) for each f-forward rule $A(f(X)) \leftarrow B(X)$ in P, if $B \in L(t)$, then $f(t) \in T$ and $A \in L(f(t))$.

(C3) for each f-backward rule $A(X) \leftarrow B(f(X))$ in P, if $B \in L(t)$ and $t = f(v)$ for some v, then $A \in L(v)$;

(C4) for each local rule $A(X) \leftarrow B_1(X), \dots, B_n(X), \mathit{not}\ C_1(X), \dots, \mathit{not}\ C_m(X)$ in P, if $\{B_1, \dots, B_n\} \subseteq L(t)$ and $\{C_1, \dots, C_m\} \cap L(t) = \emptyset$, then $A \in L(t)$;

(S1) for each $A \in L(t)$, depending on the type of the rule $r = R(t, A)$, we have:

a) if $r = A(f(X)) \leftarrow B(X)$, then $r \in P$, $t = f(v)$ for some v and $B \in L(v)$;
b) if $r = A(X) \leftarrow B(f(X))$, then $r \in P$, $f(t) \in T$ and $B \in L(f(t))$;
c) if $r = A(X) \leftarrow B_1(X), \dots, B_n(X)$, then for some $\{C_1, \dots, C_m\} \cap L(t) = \emptyset$, the rule $A(X) \leftarrow B_1(X), \dots, B_n(X), \mathit{not}\ C_1(X), \dots, \mathit{not}\ C_m(X)$ is in P and $\{B_1, \dots, B_n\} \subseteq L(t)$.

(W1) For each $A \in L(t)$, if $R(t, A) = A(X) \leftarrow B_1(X), \dots, B_n(X)$, then $B_j \prec A$ for all $j \in \{1, \dots, n\}$.

(W2) If $A, B \in L(t)$ and $R(t, A) = A(X) \leftarrow C(f(X))$ for some C, and there is exists $D \in L(f(t))$ such that $R(f(t), D) = D(f(X)) \leftarrow B(X)$ and $D = C$ or $D \prec_{f(t)} C$, then $B \prec_t A$.

We note that (W2) is designed to avoid cycles in $\prec^{\mathcal{V}}$ spanning, intuitively, over several nodes in the term tree. In particular, the condition ensures the following property: if we have two atoms $B(t), A(t) \in \mathsf{int}(\mathcal{V})$ and there exists a sequence

$$B(t) \dot{\prec}^{\mathcal{V}} C_1(v_1) \dot{\prec}^{\mathcal{V}} \cdots \dot{\prec}^{\mathcal{V}} C_n(v_n) \dot{\prec}^{\mathcal{V}} A(t)$$

where each v_j is a superterm of t, then $B(t) \dot{\prec}^{\mathcal{V}} A(t)$. In other words, if we look at two predicate names B, A at some node t, and there is a path via $\dot{\prec}^{\mathcal{V}}$ that involves a descendant of t, then the existence of such path is already witnessed at the node t by the associated partial order $\prec^{\mathcal{V}}_t$.

We can now show the correspondence between well-aligned trees and stable models:

Theorem 4.50. *Given a normal function-safe core program P, we have:*

$$SM(P) = \{\mathsf{int}(\mathcal{V}) \mid \mathcal{V} \textit{ is a well-aligned term tree for } P\}.$$

Proof. Assume a well-aligned $\mathcal{V} = \langle T, L, O, R\rangle$ for a program P, and $I = \mathsf{int}(\mathcal{V})$. We argue that $I \in SM(P)$. Due to (C1-C4) of Definition 4.49, we have that I is a model of P^I. Due to (S1), (W1) and the definition $\dot{\prec}^{\mathcal{V}}$, for each $p \in I$ there exists a rule $p \leftarrow q_1, \dots, q_n$ in P^I with $q_i \dot{\prec}^{\mathcal{V}} p$ for all $i \in \{1, \dots, n\}$.

Given Proposition 4.47, it suffices to see that $(\dot{\prec}^{\mathcal{V}})^+$ (i.e., the transitive closure of $\dot{\prec}^{\mathcal{V}}$) is a strict partial order, or, equivalently, that $\dot{\prec}^{\mathcal{V}}$ has no cycle. To this end we chop $\dot{\prec}^{\mathcal{V}}$ into slices. For all $1 \le d \le |\mathsf{preds}(P)|$, let $\dot{\prec}_{|d}$ denote the restriction of $\dot{\prec}^{\mathcal{V}}$ to atoms $A(t) \in I$ with $\mathsf{depth}(t) \le d$. We simply show by induction on d that $\dot{\prec}_{|d}$ is acyclic for each $1 \le d \le |\mathsf{preds}(P)|$. Note that for $d = |\mathsf{preds}(P)| + 1$, $\dot{\prec}_{|d} = \dot{\prec}^{\mathcal{V}}$ by (ii) in Definition 4.48.

(i) Base case. For the case $d = 1$, the order $\dot{\prec}_{|d}$ is trivially acyclic because $\dot{\prec}_{|1} = \{\langle B(c), A(c)\rangle \mid B \prec_c^{\mathcal{V}} A\}$ and $\prec_c^{\mathcal{V}}$ is a strict partial order by Definition 4.48.

(ii) Inductive case. Assume a $d > 1$ and suppose $\dot{\prec}_{|i}$ is acyclic for all $1 \le i < d$. We show that $\dot{\prec}_{|d}$ is also acyclic. Suppose it is not the case, i.e., there exist a sequence of atoms $A_1(t_1) \dot{\prec}_{|d} A_2(t_2) \dot{\prec}_{|d} \dots \dot{\prec}_{|d} A_{k-1}(t_{k-1}) \dot{\prec}_{|d} A_k(t_k)$ with $A_1(t_1) = A_k(t_k)$. Due to the definition of $\dot{\prec}^{\mathcal{V}}$ and since $\prec_t^{\mathcal{V}}$ is a strict partial order for all $t \in T$, there must exist $j \in \{1, \dots, k\}$ such that $\mathsf{depth}(t_j) < d$, i.e., the cycle involves terms of levels $< d$. On the other hand, since $\prec_{|d-1}$ is acyclic by the induction hypothesis, for some term v with $\mathsf{depth}(v) = d - 1$, there must exist two atoms $A(v), B(v)$ and a function symbol f such that:

   a) $C(f(v)) \prec_{|d} A(v)$ and $B(v) \prec_{|d} D(f(v))$ for some C, D with $C = D$ or $D \prec_{f(v)} C$, and

   b) $B(v) \not\prec_{|d} A(v)$.

   We arrive at a contradiction, this cannot be the case by (W2).

For the other direction, assume $I \in SM(P)$. We build a well-aligned term tree $\mathcal{V}$ for P such that $\mathsf{int}(\mathcal{V}) = I$. To this end, take any strict partial order $\prec$ over I such that for each $p \in I$ there is a rule $p \leftarrow q_1, \dots, q_n$ in P^I with $q_i \prec p$ for all $i \in \{1, \dots, n\}$. For each $p \in I$, we choose and fix one rule r^p from above. Recall that the desired $\prec$ exists by Proposition 4.47. We can now define the following term tree $\mathcal{V} = \langle T, L, O, R\rangle$ where:

(i) $T = \{t \mid \exists A \in \mathsf{preds}(P) : A(t) \in I\}$, i.e., T is the set of term occurring in I (recall also that by Proposition 4.45, $\mathsf{depth}(t) \le |\mathsf{preds}(P)|$ for all $t \in T$);

(ii) L assigns to each $t \in T$ the set $L(t) = \{A \mid A(t) \in I\}$;

(iii) O assigns to each $t \in T$ the relation $O(t) = \{(A, B) | A(t) \prec B(t)\}$;

(iv) For each $t \in T$ and $A \in L(t)$, depending on the type of $r^{A(t)}$, we have:

  a) $R(t, A) = A(X) \leftarrow B_1(X), \ldots, B_n(X)$, in case $r^{A(t)} = A(t) \leftarrow B_1(t), \ldots, B_n(t)$;

  b) $R(t, A) = A(X) \leftarrow B(f(X))$, in case $r^{A(t)} = A(t) \leftarrow B(f(t))$;

  c) $R(t, A) = A(f(X)) \leftarrow B(X)$, in case $r^{A(t)} = A(f(v)) \leftarrow B(t)$ where $f(v) = t$.

It is easy to see that $\mathcal{V}$ is well-aligned. The properties (C1-C4) in Definition 4.49 hold since I is a model of P^I. Furthermore, (S1) and (W1) are satisfied due to the existence of $\prec$. Finally, the fact that $\prec$ is transitive also ensures that (W2) is satisfied. $\square$

Given the above characterization, consistency of a normal function-safe core program P can be reduced to checking if a well-aligned term tree for P exists. To understand the algorithm that we develop next, observe that goodness of a term $f(t)$ in a term tree $\mathcal{V}$ depends only on (the label of) $f(t)$, t, and the terms of the form $f_1(f(t)), \ldots, f_n(f(t))$ of $\mathcal{V}$. In other words, the conditions are *local*. Intuitively, this means that to check if a candidate term tree is well-aligned it suffices to traverse it on a path basis.

We make this more formal next.

Definition 4.51. *Given a term tree $\mathcal{V} = \langle T, L, O, R \rangle$ and a term $t \in T$, we let $\mathcal{V}_{|t} = \langle T', L', O', R' \rangle$ where:*

(i) $T' = T_1 \cup T_2$ with

  a) $T_1 = \{v \in T \mid v \text{ is a subterm of } t\}$ (note $t \in T_1$), and

  b) $T_2 = \{f(v) \in T \mid v \in T_1 \text{ and } f \text{ is a function from } P\}$.

(ii) L', O', R' are respective restriction of L, O, R to terms in T'.

In other words, $\mathcal{V}_{|t}$ is obtained by restricting $\mathcal{V}$ to the path from the root to the term t and the children of the nodes on the path. Then the following is an immediate consequence of the definition of well-aligned term trees.

Proposition 4.52. *Given a term tree $\mathcal{V} = \langle T, L, O, R \rangle$ for P, $\mathcal{V}$ is well-aligned iff for each $t \in T$, t is good in $\mathcal{V}_t$.*

```
Algorithm consistencyTest
Input: a function-safe normal $\mathbb{BD}$-program $P$
Output: true iff there exists a well-aligned tree $\mathcal{V}$ for $P$
  let $d := |\mathsf{preds}(P)|$
  guess a tree $\mathcal{V} = \langle T, L, O, R\rangle$ for $P$ with $T = \{c\}$
return test$(P, \mathcal{V}, c, d-1)$

function test$(P, \mathcal{V}, t, d)$
  if $(d \neq 1)$ then nondeterministically expand $\mathcal{V}$ with some children of $t$
  if $t$ is not good in $\mathcal{V}$, then return false
  if test$(P, \mathcal{V}, t', d-1) =$ true for all children $t'$ of $t$ in $\mathcal{V}$, then return true
return false
```

Figure 4.4: Alternating procedure for testing consistency of normal function-safe $\mathbb{BD}$ programs via existence of well-aligned trees.

We can now define a procedure for testing consistency of a normal function-safe core program P. The alternating algorithm that checks the existence of a well-aligned term tree for P is given in Figure 4.4. First, the procedure nondeterministically chooses a labeling for the root node, that is the constant c. Then it recursively expands the tree by adding child nodes and nondeterministically labeling them. For each node t, it tests whether t is good in the candidate term tree. Note that the full candidate is not kept in memory, in each computation path the algorithm operates only on the relevant fragment of the full tree (this is enabled by Proposition 4.52). Due to the polynomial bound on the depth of term trees for P, the given alternating procedure runs time polynomial in the size of P. Since AP = PSPACE [CKS81], this gives us the desired upper bound.

Lower bound. To show that the above procedure is worst-case optimal, we give a polynomial time reduction from the PSPACE-complete evaluation problem for quantified Boolean formulas (QBFs). We define next QBFs in prenex normal form; generic QBFs can be rewritten into this form in linear time while preserving the truth value.

Definition 4.53. *A* quantified Boolean formula (QBF) *is an expression of the form*

$$F = Q_1 x_1 Q_2 x_2 \ldots Q_n x_n . \varphi,$$

where $Q_i \in \{\forall, \exists\}$, $1 \leq i \leq n$, and φ is a Boolean formula over $\mathbf{true}$, $\mathbf{false}$ *and the propositional letters from $\{x_1, \ldots, x_n\}$. For a Boolean formula φ, let $\varphi_{x \leftarrow y}$ be the formula that results after replacing each occurrence of x in φ by y. The truth value $v(F) \in \{\mathbf{true}, \mathbf{false}\}$ of F is determined inductively:*

(i) in case $n = 0$ (i.e., F is quantifier-free), $v(F) = \mathbf{true}$ iff φ evaluates to $\mathbf{true}$ (note that φ is over $\{\mathbf{true}, \mathbf{false}\}$ in this case);

(ii) in case $n > 0$, depending on Q_1, we have:

(a) if $Q_1 = \exists$, then $v(F) = \mathbf{true}$ iff $v(Q_2x_2 \ldots Q_nx_n.\varphi_{x_1 \leftarrow p}) = \mathbf{true}$ for some $p \in \{\mathbf{true}, \mathbf{false}\}$;

(b) if $Q_1 = \forall$, then $v(F) = \mathbf{true}$ iff $v(Q_2x_2 \ldots Q_nx_n.\varphi_{x_1 \leftarrow p}) = \mathbf{true}$ for all $p \in \{\mathbf{true}, \mathbf{false}\}$.

For the reduction, assume a QBF $F = Q_1x_1Q_2x_2 \ldots Q_nx_n.\varphi$. We next show how to construct in polynomial time a function-safe normal $\mathbb{BD}$-program P_F such that $v(F) = \mathbf{true}$ iff P_F is inconsistent. W.l.o.g. we assume that $n \neq 0$ (i.e., F has at least one quantifier), and that φ is in negation normal form, i.e., negation occurs in front of propositions only.

Intuitively, P_F generates a tree with $n + 1$ nodes on each path in a way that each branch corresponds to an assignment of truth values to the propositional letters of F. The leaf nodes are used to evaluate φ under the generated assignments, and then the results are propagated back to the root while taking into account the quantifiers.

For the encoding we use:

- two function symbols f, g to simulate branching;
- predicate names $L_1, \ldots, L_{n+1}$ to indicate levels of the tree;
- predicate E_α for each $\alpha \in S \cup \{\neg x_1, \ldots, \neg x_n\}$, where S denotes the set of subformulas of φ.

The program P_F is as follows. The first 3 rules are to construct the required binary tree.

$$L_1(c) \leftarrow \;, \tag{4.73}$$
$$L_{i+1}(f(X)) \leftarrow L_i(X), \qquad \text{for } i \in \{1, \ldots, n\}, \tag{4.74}$$
$$L_{i+1}(g(X)) \leftarrow L_i(X), \qquad \text{for } i \in \{1, \ldots, n\}. \tag{4.75}$$

We now decorate nodes with possible values of the propositions in φ. For this, we add the following rules for each $i \in \{1, \ldots, n\}$:

$$E_{\neg x_i}(f(X)) \leftarrow L_i(X), \tag{4.76}$$
$$E_{x_i}(g(X)) \leftarrow L_i(X), \tag{4.77}$$
$$E_{x_i}(u(X)) \leftarrow E_{x_i}(X), L_{j+1}(u(X)), \qquad \text{for } u \in \{f, g\}, 1 \leq j \leq n, \tag{4.78}$$
$$E_{\neg x_i}(u(X)) \leftarrow E_{\neg x_i}(X), L_{j+1}(u(X)), \qquad \text{for } u \in \{f, g\}, 1 \leq j \leq n. \tag{4.79}$$

For each proposition x_i, where $1 \leq i \leq n$, the rules (4.76) and (4.77) give rise to the two different alternatives for value of x_i. The remaining rules (4.78) and (4.79) propagate the assignment to the leaves of the generated tree.

For evaluating φ at each leaf, we use the following rules.

(i) For each $\alpha_1 \vee \alpha_2 \in S$, we add:

$$E_{\alpha_1 \vee \alpha_2}(X) \leftarrow E_{\alpha_1}(X), L_{n+1}(X),$$

$$E_{\alpha_1 \vee \alpha_2}(X) \leftarrow E_{\alpha_2}(X), L_{n+1}(X).$$

(ii) For each $\alpha_1 \wedge \alpha_2 \in S$, we add the rule

$$E_{\alpha_1 \wedge \alpha_2}(X) \leftarrow E_{\alpha_1}(X), E_{\alpha_2}(X), L_{n+1}(X).$$

Intuitively, $E_\varphi(t)$ is true for a leaf node t if φ evaluates to true under the assignment given in the path to t. To check whether F evaluates to true, we propagate E_φ to the root.

For each $i \in \{1, \ldots, n\}$, we add the following rules:

$$E_\varphi(X) \leftarrow L_i(X), E_\varphi(f(x)), \qquad \text{in case } Q_i = \exists, \tag{4.80}$$
$$E_\varphi(X) \leftarrow L_i(X), E_\varphi(g(x)), \qquad \text{in case } Q_i = \exists, \tag{4.81}$$
$$E_\varphi(X) \leftarrow L_i(X), E_\varphi(f(x)), E_\varphi(g(x)), \qquad \text{in case } Q_i = \forall. \tag{4.82}$$

We can now add the constraint $\leftarrow E_\varphi(c)$ to obtain the final program P_F. By construction, $v(F) = \mathbf{true}$ iff P_F is inconsistent. The reduction is clearly polynomial in size of F. Note also that negation was not used in P_F, and thus PSPACE-hardness of inconsistency testing already applies for positive programs. Since completeness for PSPACE is closed under complementation of languages, we obtain the desired lower bound.

Theorem 4.54. *Deciding consistency of normal function-safe core programs is* PSPACE*-complete, even if negation is disallowed.*

4.4.2 Disjunctive Function-Safe Core Programs

We deal now with the disjunctive case, and show that checking consistency in disjunctive function-safe core programs is NEXPTIME-complete. Note that the problem is 2-EXPTIME-complete in the general case.

The upper bound can be shown in the guess-and-check manner. Recall that to compute the stable models of a function-safe core program P we can limit our attention to candidate interpretations with atoms $A(t)$ such that $\mathsf{depth}(t) \leq |\mathsf{preds}(P)|$. Each such interpretation I is finite and has at most exponential size in the size of P. Note also that we can check in single exponential time whether I is a *model* of P^I. Thus the single non-trivial task is to check in exponential time (in $|P|$) whether I is a *minimal* model of P^I, i.e., $I \in MM(P^I)$.

Let us assume for the rest of the section a disjunctive function-safe core program P and a model I of P^I such that $\mathsf{depth}(t) \leq |\mathsf{preds}(P)|$ for each term t occurring in I.

To test whether I is a minimal model of P^I, we resort to splits for disjunctive programs introduced in Section 4.3. In particular, we employ Theorem 4.29 which tells us the following: $I \notin MM(P^I)$ iff there exists a split P' of P^I w.r.t. I such that I is not the least model of P'. We develop next an algorithm to test the latter condition.

We must be careful: there can double exponentially many splits of P^I w.r.t. I, and hence dealing with one split at a time would lead to a double exponential time algorithm. Our solution to the above problem consists of two parts:

(i) We introduce the notion of a *witness to non-minimality*, which is a structure that witnesses the existence of a split P' of P^I w.r.t. I such that I is not the least model of P'. Intuitively, a witness is a pair of a split and a proof that shows some atoms in I to be *unfounded* w.r.t. the split. We also show that in case the above P' exists, there is always a witness for it. The original characterization of nonminimality via unfounded sets of atoms for nondisjunctive programs can be found in [GRS91]. Our approach is similar to the one in [LRS97] for disjunctive programs, but uses splits to deal with disjunction and allows us to obtain a self-contained proof of the upper bound.

(ii) We introduce a procedure to check the existence of a witness in time single exponential in the size of P. Combining this result with the observations in the beginning of the section, we obtain the desired NEXPTIME upper bound.

The notion of a witness to non-minimality is defined next. We w.l.o.g. assume that $A(f(t)) \in I$ implies $B(t) \in I$ for some B. The assumption allows us to view the set of terms occurring in T as a tree. Recall that, by Proposition 4.4, we can discard I if it violates the above condition.

Definition 4.55 (Witnesses to non-minimality)**.** *Let $T = \{t \mid \exists A : A(t) \in I\}$, i.e., T is the set of terms occurring as arguments in atoms of I.*

A set Q of rules is called a split of P^I w.r.t. $t \in T$ and I *if Q is a $\subseteq$-minimal set of rules satisfying the following condition: if r is a local rule in P^I, t is the argument of atoms in r, and $\mathsf{head}(r) \cap I \neq \emptyset$, then $h \leftarrow \mathsf{body}^+(r) \in Q$ for some $h \in \mathsf{head}(r) \cap I$.*

Assume a pair $(\mathcal{R}, \mathcal{U})$, where

- *$\mathcal{R}$ is a mapping that assigns to each term $t \in T$ a split of P w.r.t. t and I, and*

- *$\mathcal{U}$ assigns to each $t \in T$, a set $\mathcal{U}(t) \subseteq \mathsf{preds}(P)$.*

We call $(\mathcal{R},\mathcal{U})$ a witness to non-minimality of I w.r.t. P^I *if $\mathcal{U}$ satisfies the following conditions:*

(W1) For some $A(t) \in I$, $A \in \mathcal{U}(t)$.

(W2) For each fact $A(c) \in P$, $A \notin \mathcal{U}(c)$.

(W3) If $t \in T$ and $A \in \mathcal{U}(t)$, then for each rule $r \in \mathcal{R}(t)$ with head atom $A(t)$, there exists a body atom $B(t)$ such that $B(t) \notin I$ or $B \in \mathcal{U}(t)$.

(W4) If $f(t) \in T$ and $A \in \mathcal{U}(f(t))$, then for each f-forward rule $A(f(X)) \leftarrow B(X)$ of P we have $B(t) \notin I$ or $B \in \mathcal{U}(t)$.

(W5) If $t \in T$ and $A \in \mathcal{U}(t)$, then for each f-backward rule $A(X) \leftarrow B(f(X))$ of P we have $B(f(t)) \notin I$ or $B \in \mathcal{U}(f(t))$.

Let $(\mathcal{R},\mathcal{U})$ be a tuple as above. Intuitively, $\mathcal{R}$ corresponds to some split P' of P^I w.r.t. I, while $\mathcal{U}$ claims that some atoms in I are unfounded w.r.t. P', or, in other words, that I is not the least model of P'. In order for the claim to be justified, we require (W1-W5). The condition (W1) ensures that at least one atom is claimed to be unfounded, while by (W2) no atoms given as facts can be stated as unfounded. The conditions (W3-W5) capture the meaning of unfoundedness, which intuitively reads as follows: an atom R is unfounded in I if in each rule that can imply R some body atom is false or is itself unfounded.

We obtain the following correspondence:

Proposition 4.56. *I is not a minimal model of P^I iff there exists a witness for non-minimality of I w.r.t. P^I.*

Proof. Suppose I is not a minimal model of P^I. By Theorem 4.29, there exists a split P' of P^I w.r.t. I such that I is not the least model of P'. We build a witness $(\mathcal{R},\mathcal{U})$ for non-minimality of I w.r.t. P^I as follows.

For a term $t \in T$, we say $r \in P^I$ is a *t-rule* if r is not a constraint and all atoms in r have t as the argument, i.e., r stems from the grounding of some local rule in P using the term t. Then for each $t \in T$, $\mathcal{R}(t) = \{r \in P' \mid r \text{ is a } t\text{-rule}\}$. Since P' is a split of P w.r.t. I, it is easy to see that $\mathcal{R}(t)$ is a split of P w.r.t. t and I.

To define $\mathcal{U}$, let J be the least model of P'. For each $t \in T$, we let $\mathcal{U}(t) = \{A \mid A(t) \in I \setminus J\}$. Then (W1-W2) are satisfied trivially because $J \subset I$ and J is a model of P'. It is also easy to verify (W3-W5). Suppose (W3) is violated, i.e., there exists $t \in T$, $A \in \mathcal{U}(t)$ such that for some rule $A(t) \leftarrow B_1(t), \ldots, B_n(t)$ of $\mathcal{R}(t)$ we have $\{B_1(t), \ldots, B_n(t)\} \subseteq I$ and $\{B_1, \ldots, B_n\} \cap \mathcal{U}(t) = \emptyset$. Then due to the way we built $\mathcal{U}$, we have $B_1(t), \ldots, B_n(t) \in J$. Since J is a model of P', it must be the case that $A(t) \in J$. We arrive at a contradiction: by the construction of $\mathcal{U}$, $A \notin \mathcal{U}(t)$. The argument for (W4-W5) is analogous.

Assume a witness $(\mathcal{R},\mathcal{U})$ for non-minimality of I w.r.t. P^I. Take the program P' containing:

(a) all facts $r \in P^I$;

(b) for each $r \in P^I$ with $\mathsf{head}(r) \cap I = \emptyset$, the constraint $\leftarrow \mathsf{body}^+(r)$;

(c) for each f-forward or f-backward $r \in P^I$ with $\mathsf{head}(r) \in I$, the rule r;

(d) each $r \in \bigcup_{t \in T} \mathcal{R}(t)$.

It is easy to see that P' is a split of P^I w.r.t. I. It remains see that I is not the least model of P' (recall Theorem 4.29). Take the interpretation $J = I \setminus \{A(t) \mid t \in T \wedge A \in \mathcal{U}(t)\}$. Clearly, $J \subset I$. It is easy to see that J is a model of P'. Suppose it is not the case, i.e., there is a rule $r \in P'$ with $\mathsf{body}(r) \subseteq J$ and $\mathsf{head}(r) \cap I = \emptyset$. The rule r cannot be a fact due to (W2) and because I is a model of P^I. There are 3 remaining cases:

(a) $r = A(f(t)) \leftarrow B(t)$ for some f and t. Since $B(t) \in I$ and I is a model of P^I, we have $A(f(t)) \in I$. Since $A(f(t)) \notin J$, we have $A \in \mathcal{U}(f(t))$. Then by (W4) we get $B \in \mathcal{U}(t)$. Hence, $B(t) \notin J$. Contradiction.

(b) $r = A(t) \leftarrow B(f(t))$ for some f and t. As above, since $B(f(t)) \in I$ and I is a model of P^I, we get $A \in \mathcal{U}(t)$. By (W5) we get $B \in \mathcal{U}(f(t))$. Hence, $B(f(t)) \notin J$. Contradiction.

(c) $r = A(t) \leftarrow B_1(t), \ldots, B_n(t)$ for some $t \in T$, i.e., r is from $\mathcal{R}(t)$. Due to the definition of $\mathcal{R}(t)$, we have $A(t) \in I$. Since r is violated in J by assumption, it must be the case that $A \in \mathcal{U}(t)$. We know that the body of r is true in I, i.e., $\{B_1(t), \ldots, B_n(t)\} \subseteq I$. Hence, by (W3), there must exist $i \in \{1, \ldots, n\}$ such that $B_i \in \mathcal{U}(t)$. This implies $B_i(t) \notin J$. We arrive at a contradiction.

Thus $J \subset I$ is a model of P', and hence I is not the least model of P'. $\square$

Given the above characterization, it remains to see that the existence of a witness to non-minimality of I w.r.t. P^I can be decided in time exponential in the size of P. In Figure 4.5 we present a recursive procedure for this purpose. To check existence of a witness $(\mathcal{R},\mathcal{U})$, the procedure tries to label the tree-shaped T in a top-down fashion, i.e., after building $\mathcal{R}(t)$ and $\mathcal{U}(t)$ for a term $t \in T$, it recursively proceeds to building $\mathcal{R}(f(t))$ and $\mathcal{U}(f(t))$ for each term $f(t) \in T$. Ensuring the correctness of the labeling, i.e., the satisfaction of (W1-W5) in Definition 4.55, is straightforward. We note that the *findUnfounded* flag is used to ensure (W1), i.e., the existence of at least one unfounded atom.

For the desired upper-bound, it clearly suffices to see that *test*$(Q, U, c, \textit{true})$ for any Q and U can be computed in time exponential in the size of P. This can be

```
function unfoundednessTest
(returns true iff there exists a witness to non-minimality of $I$ w.r.t. $P^I$)
  guess a split $Q$ of $P^I$ w.r.t. $c$ and $I$
  guess a set $U \subseteq \mathsf{preds}(P) \setminus \{A \mid A(c) \leftarrow \text{ in } P\}$        (W2)
return test($Q, U, c,$ true)

function test($Q, U, t,$ findUnfounded)
  let $S = \{f(t) \mid \exists A \in \mathsf{preds}(P) : A(f(t)) \in I\}$
  if $S = \emptyset \wedge$ findUnfounded $=$ true $\wedge U = \emptyset$ then return false        (W1)
  if there exists $A(t) \leftarrow B_1(t), \ldots, B_n(t) \in Q$ such that:        (W3)
          $A \in U$, $\{B_1(t), \ldots, B_n(t)\} \subseteq I$ and $\{B_1, \ldots, B_n\} \cap U = \emptyset$
      then return false
  forall $v \in S$ do guess a pair $(Q_v, U_v)$, where
     (1) $Q_v$ is a split of $P^I$ w.r.t. $v$ and $I$, and
     (2) $U_v \subseteq \mathsf{preds}(P)$.
  if there exists $f(t) \in S$ such that:
    (1) $\exists A(f(X)) \leftarrow B(X) \in P$ s.t. $A \in U_{f(t)}$, $B(t) \in I$ and $B \notin U$, or        (W4)
    (2) $\exists A(X) \leftarrow B(f(X)) \in P$ s.t. $A \in U$, $B(f(t)) \in I$ and $B \notin U_{f(t)}$        (W5)
   then return false
  if findUnfounded $=$ true $\wedge U \neq \emptyset$        (W1)
      then let $C = \emptyset$
      else let $C = \{s\}$ for some $s \in S$
  if  for all $v \in S$, test($Q_v, U_v, v, v \in C$) $=$ true
      then return true
      else  return false
```

Figure 4.5: A procedure to decide the existence of a witness to non-minimality of I w.r.t. P^I.

seen by computing the values of *test* in the bottom-up fashion, i.e., the values for a term t are computed using the precomputed values for terms $f(t) \in T$. Observe that *test*$(Q, U, t,$ *findUnfounded*$)$ can be computed in polynomial time in case t has no successor terms $f(t) \in T$. Otherwise, *test*$(Q, U, t,$ *findUnfounded*$)$ can be computed by traversing exponentially many choices of a labeling for successors of $f(t) \in T$ and then checking the results of *test* for each such successor. Since the maximal depth of terms in T is bounded by $|\mathsf{preds}(P)|$, we get that computing time for *test*$(Q, U, c,$ *findUnfounded*$)$ is bounded by $2^{\mathcal{O}(|P| \cdot (|\mathsf{preds}(P)|))}$, i.e., single exponential in $|P|$.

We note that *unfoundednessTest* can be seen as an alternating procedure which runs in polynomial time in the size of P, but with access to an oracle that allows to query $A(f(t)) \in I$ for any input term t. Since AP $=$ PSPACE, it follows that the space required by *unfoundednessTest* to do the computation (disregarding the space required to store I, which can be exponential) is polynomially in the size of P.

Lower bound. The presented algorithm is worst-case optimal. To see this, we reduce the consistency problem for disjunctive DATALOG programs to checking consistency in (disjunctive) function-safe core programs. Recall that the problem is NEXPTIME-complete [EGM97].

Assume a disjunctive DATALOG program P. We construct a function-safe core program P' such that P is consistent iff P' is consistent. Recall that DATALOG rules allow for arbitrary predicate arities but disallow function symbols. Thus $\mathcal{HU}^P$ is the set of constants in P.

For the encoding, we w.l.o.g. assume the following:

- Only facts have constants as arguments;
- All predicate occurring in P have the same arity $ar > 0$.
- Each rule has variables only from $\{X_1, \ldots, X_{mv}\}$, where mv is the maximal number of variables in the rules of P.

We build P' as follows. The first step is to generate a tree where leaves correspond to the possible tuples $\langle c_1, \ldots, c_{ar}\rangle$ of constants in P. To this end, for each constant d of P, we use the function symbol g_d. We add the following rules to P':

$$L_0(c) \leftarrow \ , \tag{4.83}$$
$$L_i(g_d(X)) \leftarrow L_{i-1}(X), \qquad \text{for } i \in \{1, \ldots, ar\} \text{ and } d \in \mathcal{HU}^P \tag{4.84}$$
$$T(X) \leftarrow L_i(X), \qquad \text{for } i \in \{0, \ldots, ar\}. \tag{4.85}$$

The last rule above is to ensure function-safety latter on. The first two rules fire an atom $L_{ar}(t)$ for each term of the form $t = g_{c_{ar}}(\ldots g_{c_1}(c)\ldots)$, where $\langle c_1, \ldots, c_{ar}\rangle$ is a tuple of constants in P (we call such t a *leaf* term). For each such t, we will also enforce an atom $POS_i^d(t)$ to be true if $c_i = d$, i.e., if the ith constant in the encoded tuple is d. For this, we add the following rules:

$$POS_i^d(g_d(X)) \leftarrow L_{i-1}(X) \qquad \text{for } i \in \{1, \ldots, ar\} \text{ and } d \in \mathcal{HU}^P \tag{4.86}$$

$$POS_i^d(g_c(X)) \leftarrow POS_i^d(X) \qquad \text{for } i \in \{2, \ldots, ar\} \text{ and } c, d \in \mathcal{HU}^P. \tag{4.87}$$

We can now use disjunctive rules to generate different interpretations for P. We employ unary symbols $S, \bar{S}$ for each predicate name S in P, and use atoms $S(t)$ (resp.,

$\bar{S}(t)$) to indicate that the predicate is true (resp., false) for the tuple of constants encoded in t. For each relation symbol S of P we add to P' the following rule:

$$S(X) \vee \bar{S}(X) \leftarrow L_{ar}(X). \tag{4.88}$$

We note here that there is a one-to-one correspondence between the minimal models of P' constructed so far, and the Herbrand interpretations for P.

We can now turn to testing the satisfaction of the rules of P. Dealing with facts is easy: for each $R(c_1, \ldots, c_{ar}) \leftarrow$ of P we enforce $S(t)$ to be true for the term t corresponding to $\langle c_1, \ldots, c_{ar} \rangle$. This is achieved using the following rule:

$$S(X) \leftarrow L_{ar}(X), POS_1^{c_1}(X), \ldots, POS_{ar}^{c_{ar}}(X). \tag{4.89}$$

To deal with the rules containing variables, we employ saturation. Assume $\mathcal{HU}^P = \{c_1, \ldots, c_n\}$, and recall that all variables in P are from $\{X_1, \ldots, X_{mv}\}$.

We first add the following rule for each $i \in \{1, \ldots, mv\}$:

$$G_{X_i,c_1}(X) \vee \cdots \vee G_{X_i,c_n}(X) \leftarrow L_0(X) \tag{4.90}$$

Using the above rule, for each variable X_i of P we select one constant of P. In other words, $G_{X_i,d}(c)$ corresponds to the replacement of X_i by the constant d.

We next add the following rules:

$$G_{X_i,c_j}(X) \leftarrow OK(X) \qquad \text{for } i \in \{1, \ldots, mv\}, j \in \{1, \ldots, n\} \tag{4.91}$$

$$CON(X) \leftarrow \bigcup_{1 \leq i \leq mv,\ 1 \leq j \leq n} \{G_{X_i,c_j}(X)\} \tag{4.92}$$

The intuition behind the above rules is as follows. Suppose the predicate OK is defined via some Horn rules in such a way that $OK(c)$ is true iff under the variable assignment generated by (4.90) there is no rule of P that is violated. In other words, given an assignment, $OK(c)$ is not implied iff there is some violated rule in P. Then by the rules (4.91) and (4.92) we get the following: $CON(c)$ is forced to be true iff there is no assignment under which a rule of P is violated, i.e., if the encoded interpretation is a model of P. Indeed, if I is a minimal model of P' such that $CON(c) \notin I$, then there is $i \in \{1, \ldots, mv\}$ and $j \in \{1, \ldots, n\}$ such that $G_{X_i,c_j}(c) \notin I$, and hence $OK(c) \notin I$. The latter can only be the case if to satisfy (4.90) we can choose a variable assignment that does not imply $OK(c)$, i.e., indicates a violated rule in P. On the other hand, if I is a minimal of P' with $CON(c) \in I$, then $G_{X_i,c_j}(c) \in I$ for all $i \in \{1, \ldots, mv\}$ and $j \in \{1, \ldots, n\}$. Due to the minimality of I, this can only be the case if under all choices in (4.90) $OK(c)$ is forced to be true in I, i.e., I encodes a model of the original program P.

We are interested in minimal models of P' that encode models of P, and thus we add the following constraint:

$$\leftarrow not\ CON(c). \tag{4.93}$$

It remains to define the test predicate OK. First, we have to replace variables in rules with constants given by the assignment. We use unary predicate names $A^{r,R}_{i,X}$ to indicate that in the atom R of the rule r we have the variable X in the position i. Similarly, we use $A^{r,R}_{i,d}$ to state that in the atom R of r the variable in position i is replaced by the constant $d \in \mathcal{HU}^P$. We can implement the replacement of variables by constants as follows.

We add the following for each rule $r \in P$ and each atom R of r such that X is the variable in the position i of R:

$$A^{r,R}_{i,X}(X) \leftarrow L_0(X), \tag{4.94}$$

$$A^{r,R}_{i,d}(X) \leftarrow L_0(X), A^{r,R}_{i,X}(X), G_{X,d}(X), \qquad \text{for all } d \in \mathcal{HU}^P. \tag{4.95}$$

Intuitively, the assignment of constants is done at the root of the generated tree (notice $L_0(X)$ in the above rules). We now propagate this information to the leaves of the tree where the truth of atoms can be determine. For all rules $r \in P$ and atoms R of r, we add to P':

$$A^{r,R}_{i,e}(g_d(X)) \leftarrow T(g_d(X)), A^{r,R}_{i,e}(X). \qquad \text{for all } d, e \in \mathcal{HU}^P \text{ and } i \in \{1, \ldots, ar\}. \tag{4.96}$$

Intuitively, via the above rules, each leaf term is now "aware" of the replacement made at the root of the tree.

We now define the predicate $EQ^{r,R}$ to identify the leaf term that stores the truth value of the atom R of $r \in P$ under the generated assignment. This is done via the rules:

$$EQ^{r,R}_i(X) \leftarrow L_{ar}(X), A^{r,R}_{i,d}(X), POS^d_i(X) \qquad \text{for all } d \in \mathcal{HU}^P \text{ and } 1 \leq i \leq ar \tag{4.97}$$

$$EQ^{r,R}(X) \leftarrow L_{ar}(X), EQ^{r,R}_1(X), \ldots, EQ^{r,R}_{ar}(X). \tag{4.98}$$

We can now determine the truth values of the grounded atoms. To this end, we use $T^{r,R}_{true}$ (resp., $T^{r,R}_{false}$) to indicate that the ground version of the atom R in $r \in P$ is true (resp., false) in the interpretation encoded by the leaf nodes. Importantly, the truth values are propagated back to the root where the test for rule satisfaction can be performed.

For all rules $r \in P$ and all atoms R, where $R = S(\vec{x})$ for some S, we add the following:

$$T^{r,R}_{true}(X) \leftarrow EQ^{r,R}(X), S(X), \tag{4.99}$$

$$T^{r,R}_{false}(X) \leftarrow EQ^{r,R}(X), \bar{S}(X), \tag{4.100}$$

$$T^{r,R}_{false}(X) \leftarrow T^{r,R}_{false}(g_d(X)), \qquad \text{for all } d \in \mathcal{HU}^P, \tag{4.101}$$

$$T^{r,R}_{true}(X) \leftarrow T^{r,R}_{true}(g_d(X)), \qquad \text{for all } d \in \mathcal{HU}^P. \tag{4.102}$$

We can now test if the rules of P are satisfied (under the assignment induced by (4.90)). For each rule $r \in P$, we define $OK_r(c)$ to be true iff under the variable assignment some head atom is true or some body atom is false. For each rule $r \in P$, where $\{H_1, \ldots, H_m\}$ and $\{B_1, \ldots, B_k\}$ are, respectively, the body and the head atoms of r, we add the following:

$$OK^r(X) \leftarrow T^{r,B_i}_{false}(X), \quad 1 \le i \le k, \tag{4.103}$$

$$OK^r(X) \leftarrow T^{r,H_i}_{true}(X), \quad 1 \le i \le m. \tag{4.104}$$

Assume $P = \{r_1, \ldots, r_n\}$. Then the required OK predicate is defined as follows:

$$OK(X) \leftarrow OK^{r_1}(X), \ldots, OK^{r_n}(X). \tag{4.105}$$

This concludes the definition of P'. Note that the above program P' is a function-safe core program, and that the reduction is polynomial in the size of P. By construction we have P is consistent iff P' is consistent. Combining the reduction with the algorithm developed in the beginning of the section, we obtain the following:

Theorem 4.57. *Consistency of function-safe core programs is* NEXPTIME*-complete.*

We note here that negation occurs only in the rule (4.93) of P'. If we delete (4.93) from P, we obtain a positive program P''. Since P' is consistent iff $P'' \models_b CON(c)$, we have that NEXPTIME-completeness applies already for brave queries over positive function-safe core programs. On the other hand, consistency and cautious queries in positive function-safe core programs are PSPACE-complete. The lower bound for these tasks follows from Theorem 4.54. The upper bound is also due to Theorem 4.54, and the fact that consistency of a positive function-safe core program can be easily reduced in polynomial time to checking consistency of a normal function-safe core program.

4.4.3 Full Function-Safe $\mathbb{BD}$ programs

Recall that any $\mathbb{BD}$-program P can be rewritten into a core program core(P) (Definition 4.8) while preserving a one-to-one correspondence between stable models. Exploiting this fact, we define function-safety for full $\mathbb{BD}$ programs via the function-safety of resulting core programs.

Definition 4.58. *A (general) $\mathbb{BD}$-program P is* function-safe *if* core(P) *is function-safe.*

Under bounded number of variables, the core program $\mathsf{core}(P)$ is of size polynomial in the size of a given function-safe $\mathbb{BD}$ program P. Thus our upper bounds for function-safe core program carry over to function-safe $\mathbb{BD}$ programs assuming a bound on the number of variables in rules. The PSPACE and NEXPTIME lower bounds for core programs also apply immediately:

Theorem 4.59. *Under bounded number of variables, checking consistency of normal (resp., disjunctive) function-safe $\mathbb{BD}$ programs is* PSPACE*-complete (resp.,* NEXPTIME*-complete).*

In all the cases considered so far, checking if a given program satisfies the given syntactic restrictions was feasible in polynomial time. This is not true for function-safe $\mathbb{BD}$ programs.

Theorem 4.60. *Checking if a $\mathbb{BD}$-program P is function-safe is* PSPACE*-complete.*

Proof. For the upper-bound, first note that the rules in $\mathsf{core}(P)$ can be traversed in polynomial space, i.e., without building $\mathsf{core}(P)$ explicitly, which, in general, would require exponential space. For each rule in $\mathsf{core}(P)$ we have to find a positive occurrence of a function-safe predicate. Thus it suffices to see that given a predicate A we can decide in polynomial space whether A is function-unsafe in $\mathsf{core}(P)$. This is an easy consequence of NPSPACE $=$ PSPACE [Sav70]. Without explicitly building the dependency graph for $\mathsf{core}(P)$, we can nondeterministically check for the existence of a cycle that witnesses unsafety of A. The procedure requires only polynomial space: it needs to store A, the original program P, a pair of predicates from $\mathsf{core}(P)$ (this corresponds to an edge in the dependency graph), and a counter of linear size to count up to $|\mathsf{preds}(\mathsf{core}(P))|$.

For the lower bound, we encode the word problem for a deterministic Turing machine $M = (Q, \Sigma, q_0, \delta)$ with polynomially bounded space. By assumption, there exists a polynomial $p(\cdot)$ such that for any input word I, M uses at most $p(|I|)$ tape cells. We also w.l.o.g. assume that M terminates on every input. Assume an input word I and let $m = p(|I|)$. Let us also assume w.l.o.g. that $\Sigma = \{\bar{0}, \bar{1}, \llcorner\!\lrcorner\}$, and let $n = |Q|$.

We build a $\mathbb{BD}$-program P such that M accepts I iff P is not function-safe. We use one function symbol f and one predicate S with arity $2m+n+5$. First, we modify M in such a way that after it moves into an accepting state, it restores the input configuration, i.e., it restarts the computation on the original word. This can be done in polynomial time.

Assume and fix and enumeration $q_1, \ldots, q_n$ of the state set S. We use constants c_{q_i} for each q_i, and also $c_{\bar{0}}, c_{\bar{1}}, c_{\llcorner\!\lrcorner}$ for the content of tape cells. Assume a ground atom $S(\vec{t})$ bellow:

$$S(s, \underbrace{t_1, \ldots, t_m}_{m}, \underbrace{t'_1, \ldots, t'_m}_{m}, u, \underbrace{u_1, \ldots, u_n, v_0, v_1, v_{\llcorner\!\lrcorner}}_{n+3})$$

The argument structure can be explained as follows:

- The first term s encodes the time instant.
- The tuple $\langle t_1, \ldots, t_m, t'_1, \ldots, t'_m \rangle$ encodes the content of the tape. The RW head is over the symbol t'_1.
- The term u stores the current state of the machine, i.e., $u = c_{q_i}$ for some $i \in \{1, \ldots, n\}$.
- The values of the last $n + 3$ terms are fixed, in other words, the content in these positions will remain the same when applying the rules. The term $u_1, \ldots, u_n$ enumerate the states, i.e., $u_i = c_{q_i}$, where $1 \leq i \leq n$. Furthermore, we have $v_0 = c_{\bar{0}}$, $v_1 = c_{\bar{1}}$, $v_{\sqcup} = c_{\sqcup}$.

Then P can be constructed as follows:

(i) For the input word $I = i_1, \ldots, i_k$, where $k \leq m$, we add the fact:

$$S(c, \underbrace{c_{\sqcup}, \ldots, c_{\sqcup}}_{m}, c_{i_1}, \ldots, c_{i_k}, \underbrace{c_{\sqcup}, \ldots, c_{\sqcup}}_{m-k}, c_{q_0}, c_{q_1}, \ldots, c_{q_n}, c_{\bar{0}}, c_{\bar{1}}, c_{\sqcup}) \leftarrow$$

(ii) for each state $q_i \in S$, where $1 \leq i \leq n$, and each symbol $b \in \{\bar{0}, \bar{1}, \sqcup\}$ with $\delta(q_i, b) = (q_j, b', +1)$, where $1 \leq j \leq n$, we add the rule $H \leftarrow B$ where:

(a) $B = S(X, \underbrace{x_1, \ldots, x_m}_{m}, \underbrace{z_b, x'_2, \ldots, x'_m}_{m}, y_{q_i}, y_{q_1}, \ldots, y_{q_n}, z_{\bar{0}}, z_{\bar{1}}, z_{\sqcup})$

(b) $H = S(f(X), \underbrace{x_2, \ldots, x_m, z_{b'}}_{m}, \underbrace{x'_2, \ldots, x'_m, z_{\sqcup}}_{m} y_{q_j}, y_{q_1}, \ldots, y_{q_n}, z_{\bar{0}}, z_{\bar{1}}, z_{\sqcup})$

(iii) The rules for transitions with $d = -1$ and $d = 0$ are analogous.

We finally note that the reduction is clearly polynomial in the size of M and I. $\square$

We note that above reduction is similar to the one in [GP03] given in the context of linear recursive *single rule programs* (*sirups*).

We note also that under bounded number variables, checking whether a $\mathbb{BD}$-program P is function-safe is feasible in polynomial time because $\mathsf{core}(P)$ is of polynomial size and can be computed in polynomial time; recall that for core programs a test for function-safety can be performed in polynomial time.

$\mathbb{BD}$ programs	Full	One function symbol	Function-safe	Function-safe with one function symbol
Disjunctive	2-ExpTime	ExpSpace	NExpTime	Σ_2^P
Normal	ExpTime	PSpace	PSpace	NP

Table 4.2: Checking consistency of $\mathbb{BD}$ programs and fragments (Completeness results under bounded number of variables)

4.5 Discussion

In this chapter we defined the class of $\mathbb{BD}$ programs and some of its fragments. $\mathbb{BD}$ programs circumvent some limitations of $\mathbb{FDNC}$ and finitely recursive programs by allowing atoms to be inferred from structurally more complex atoms. In the context of temporal reasoning or planning, this enables reasoning about the past. One possible applications is updating the values of fluents (in the past) based on a current observation. For instance, in (an extension of) the Yale shooting example it might be useful to state the following: *if in the current time instant the target is intact, then it was intact in the previous time instant.* This can be easily expressed via a rule in the syntax of $\mathbb{BD}$ programs. The fragment also allows to elegantly require finiteness of stable models (see the finiteness filter in Example 4.6), which cannot be imposed in $\mathbb{FDNC}$ programs. On the other hand, $\mathbb{BD}$ programs are computationally more expensive than $\mathbb{FDNC}$.

Since $\mathbb{BD}$ programs are not finitely recursive, the reasoning methods of this chapter are significantly different from the model construction via knots in the previous chapter, or the model building method for finitely recursive programs [Bon04]. In $\mathbb{BD}$ programs, an atom can be proven using structurally more complex atoms, and thus we had to develop a mechanism to ensure finiteness of proofs, i.e., to ensure that atoms are not added unfoundedly.

The complexity of reasoning in $\mathbb{BD}$ programs and the considered fragments is summarized in Table 4.2. In terms of expressivity, $\mathbb{BD}$ programs subsume $\mathbb{FDNC}$, but standard reasoning is harder by an exponential. It is interesting to note that $\mathbb{FDNC}$ and normal $\mathbb{BD}$ programs have the same complexity but are orthogonal in expressivity. In particular, normal $\mathbb{BD}$ programs allow to enforce finiteness of stable model, while $\mathbb{FDNC}$ allows for disjunction (which cannot be succinctly simulated in $\mathbb{BD}$ programs).

As in the case of $\mathbb{FDNC}$, the class of $\mathbb{BD}$ programs is defined using syntactic restrictions, which modularly apply on the rules, and can checked in polynomial time. The same applies for function-safe core programs, except the full function-safe $\mathbb{BD}$ programs. For the latter, the problem is PSpace-complete, but is not harder than standard reasoning in the fragment.

For $\mathbb{BD}$ programs with unbounded number of variables, we obtain an exponential increase in complexity (to completeness for 3-ExpTime in full $\mathbb{BD}$-programs, and

2-ExpTime in the non-disjunctive case). Intuitively, $\mathbb{BD}$ programs are exponentially more succinct than core programs, and hence the reduction to a core program (see Definition 4.9) is exponential in general. The 2-ExpTime-hardness of normal $\mathbb{BD}$-programs can be shown by encoding an alternating Turing machine operating in exponential space. As already discussed in the context of higher-arity $\mathbb{FDNC}$, in case of unbounded arities, the nodes in tree-shaped stable models of $\mathbb{BD}$ programs can be viewed as ordinary databases storing exponentially long configurations of the machine. With the availability of disjunction and unbounded number of variables, our 2-ExpTime-hardness result in Section 4.3.1 can be lifted to a proof of 3-ExpTime-hardness. The only tricky part is to replace the original counter consisting of polynomially many bits by a counter with exponentially many bits. This can be done by simply storing it as a database of exponential size (this is exploited, e.g., in [DEGV01] for proving ExpTime-hardness of reasoning in Datalog).

The high expressivity of $\mathbb{BD}$ programs makes them a possible host for encoding problems with matching complexity into ASP with function symbols. Examples are 2-ExpTime-complete planning problems (e.g., conditional planning, cf. [Rin04]) and reasoning tasks in Description Logics (e.g., answering conjunctive queries in $\mathcal{SHIQ}$ [GLHS08, CEO07] and satisfiability testing in $\mathcal{SRIQ}$ [Kaz08, CEO09]) that can be encoded in core $\mathbb{BD}$ programs. Full $\mathbb{BD}$ programs can, e.g., accomodate conjunctive query answering in description logics $\mathcal{SRIQ}$, $\mathcal{SROQ}$ and $\mathcal{SROI}$, which is feasible in 3-ExpTime [CEO09]. To our knowledge, no ASP classes, as simple as $\mathbb{BD}$ programs, with similar capacity were identified before.

Chapter 5

Related Work

$\mathbb{FDNC}$ programs and $\mathbb{BD}$ programs enlarge the range of decidable ASP programs with function symbols. We compare next our work with other related approaches.

5.1 Finitely Recursive and Finitary Programs

Our class $\mathbb{FN}$, which results from $\mathbb{FDNC}$ by disallowing constraints and disjunction, is in essence (modulo elimination of rules (R2) and (R4)) a decidable subclass of the finitely recursive programs (FRPs) in [Bon04, BBC09]. In this formalism, inconsistency checking is R.E.-complete and brave entailment ground atoms is co-R.E.-complete in general [BBC09]. For $\mathbb{FN}$ and our full class $\mathbb{FDNC}$, which implicitly obeys the restrictions of FRPs, these problems are EXPTIME-complete. On the other hand, $\mathbb{FN}$ is not a subclass of the *finitary programs (FPs)* [Bon04], which are defined as finitely recursive programs with only finitely many atoms occurring 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 $\mathbb{FN}$ and all other fragments of $\mathbb{FDNC}$ allows effective recognition of their programs. Recognition of FRPs and FPs, instead, suffers from undecidability.

We recall that, due to rules of the form $A(X) \leftarrow A(f(X))$, $\mathbb{BD}$ programs are not finitely recursive. Naturally, since the aforementioned reasoning tasks are decidable for $\mathbb{BD}$ programs, there are numerous problems that can be encoded in finitely recursive programs, but not in $\mathbb{BD}$ programs.

5.2 Finitely Ground Programs

In [CCIL08a] the authors introduced *finitely ground* ($\mathcal{FG}$) and *finite domain* ($\mathcal{FD}$) programs that allow for function symbols and negation under the stable model semantics. The main idea is to consider an *intelligent instantiation* of a program, which, intuitively, corresponds to a subset of the grounding that is relevant for computing the stable models of a program. If the intelligent instantiation is finite, the stable models of the program can be computed using standard methods (see [CCIL08b] for the implementation based on DLV). The class $\mathcal{FG}$ is defined in terms of programs for which a finite intelligent instantiation can be obtained. $\mathcal{FG}$ programs are decidable for the standard reasoning tasks, and, in fact, are expressive enough to capture all computable functions. For this reason, recognizing $\mathcal{FG}$ programs is only semi-decidable. Consistent $\mathcal{FG}$ programs only have finitely many stable models and they are all finite. Recall that $\mathbb{FDNC}$ and $\mathbb{BD}$ programs can have infinite stable models. Thus, even though $\mathcal{FG}$ programs capture all computable functions, $\mathbb{FDNC}$ and $\mathbb{BD}$ programs are not subsumed by $\mathcal{FG}$ programs. On the other hand, function-safe core programs and full function-safe $\mathbb{BD}$ programs are fragments of $\mathcal{FG}$ programs, and thus stand out as subclasses with effectively recognizable syntax.

The class $\mathcal{FD}$ is defined in terms of effectively recognizable syntactic restrictions that ensure the programs are finitely ground. The restrictions are similar to function-safety and are designed to limit recursion. Strictly speaking, function-safe core programs are not subsumed by $\mathcal{FD}$ programs. For instance, $P_1 = \{A(f(X)) \leftarrow A(X); B(c) \leftarrow\}$ is a function-safe core program; indeed, the predicate A is function-safe because it is not reachable from B in the dependency graph. However, P_1 is not in $\mathcal{FD}$. If we drop (b) in Definition 4.44, we obtain a fragment of function-safe core programs that is subsumed by $\mathcal{FD}$. However, even without (b), full function-safe $\mathbb{BD}$ programs are not subsumed by $\mathcal{FD}$. This is witnessed by the following function-safe $\mathbb{BD}$-program $P_2 = \{A(f(X), z_1, z_2, z_2) \leftarrow A(X, z_1, z_1, z_2); A(c, a, a, b) \leftarrow\}$. P_2 is function-safe because the dependency graph of $\mathsf{core}(P_2)$ does not have a cycle.

We remark here that a relaxation of the conditions for $\mathcal{FD}$ programs was introduced in [LL09]. Full function-safe $\mathbb{BD}$ programs are not subsumed by the introduced class of programs, even if (b) in Definition 4.44 is not required; the above program P_2 again provides a counter-example.

5.3 ω-restricted Logic Programs

For logic programs with negation under stable model semantics, ω-restricted logic programs have been presented in [Syr01] and have been implemented in the SMODELS system [SNS02]. These are normal logic programs with function symbols of arbitrary arities and an unbounded number of variables, but have restricted syntax to ensure that all answer sets of a program are finite. 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 do not exclude cyclic dependencies, and they lack 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. We note that function-safe core programs are ω-restricted in case (b) in Definition 4.44 is deleted.

5.4 λ-restricted Logic Programs

λ-programs where introduced in [GST07]. They are a relaxation of ω-restricted programs, and strictly subsume them. However, the distinguishing features are again that λ-restricted programs have finitely many finite stable models. As noted previously, $\mathbb{FDNC}$ and $\mathbb{BD}$ programs allow to enforce infinitely many possibly infinite stable models. Full function-safe $\mathbb{BD}$ programs are orthogonal to λ-restricted programs. Function-safe core programs become a fragment of λ-programs if (b) in Definition 4.44 is deleted.

5.5 Local Extended Conceptual Logic Programs

Another formalism related to our languages, and especially to $\mathbb{FDNC}$ programs, are *Local Extended Conceptual Logic Programs (LECLPs)* [HNV05] which evolved from [Hey06] and extend *Conceptual Logic Programs (CLPs)* with ground rules. Such programs are function-free but have answer sets over *open domains*, i.e., answer sets 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 rewrite P into a program P' under the standard answer set semantics with a double exponential blow-up in the size of the program, and then use a standard ASP solver. The consistency problem for $\mathbb{FDNC}$ and $\mathbb{BD}$ programs is ExpTime-complete and 2-ExpTime-complete, respectively, and thus less complex.

Comparing the expressiveness of LECLPs with that of $\mathbb{FDNC}$ and $\mathbb{BD}$ programs is intricate due to the different settings. At least, all three formalisms can encode certain description logics (e.g., $\mathcal{ALC}$). However, 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, $\mathbb{BD}$ programs facilitate reasoning in DLs with inverse roles (e.g., in $\mathcal{ALCI}$), which were not considered in [HNV05].

While LECLPs and CLPs have desirable features for certain applications (e.g., for ontological reasoning), these languages deviate from the general intuition behind the minimal model semantics of logic programs. Modeling in them requires the use of the so-called *free rules* of the form $p(x) \vee \mathit{not}\ p(x) \leftarrow$; to unfoundedly add atoms into an answer set. $\mathbb{FDNC}$ and $\mathbb{BD}$, instead, do not allow for free rules, and each atom in a stable model of P must be justified from the facts of P.

We note that for reasoning in CLPs, [Hey06] presents a similar automata construction as we do in Section 4.2.2 for $\mathbb{BD}$ programs. However, CLPs lack disjunction (in the usual sense) and, in this respect, are easier to handle.

5.6 DATALOG$_{nS}$

A close relative of $\mathbb{FDNC}$ is DATALOG$_{nS}$ [CI93, Cho95], which provides an extension of DATALOG 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 [CI93] presented 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 [CI93] does not extend to $\mathbb{FDNC}$ programs, which can have multiple (even infinitely many) stable models. The knot technique, which uses knots as building blocks for stable models, handles multiplicity of models by knot sharing, i.e., the same knot may be used in several stable models.

Notably, both ordinary and higher-arity $\mathbb{F}$ have lower complexity than DATALOG$_{nS}$, at least regarding data complexity (which was considered in [CI93]). 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, a 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 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 [Cho95], 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.

Via a minor (polynomial) rewriting of programs, normal positive $\mathbb{BD}$ programs are exactly *normalized* DATALOG$_{nS}$ programs (see [CI93] for normalization, which does not alter any results on full DATALOG$_{nS}$). Using our results, we can extend DATALOG$_{nS}$ with disjunction and/or negation under the stable model semantics; let us denote the three resulting languages DATALOG$^{\vee}_{nS}$, DATALOG$^{\neg}_{nS}$ and DATALOG$^{\neg,\vee}_{nS}$. Our complexity results on $\mathbb{BD}$ programs carry over to the three extensions. For DATALOG$^{\neg}_{nS}$, standard reasoning problems considered here (consistency, brave/cautious entailment of ground/existentially quantified queries) are 2-EXPTIME-complete. For DATALOG$^{\vee}_{nS}$ and DATALOG$^{\neg,\vee}_{nS}$, the same problems are 3-EXPTIME-complete. We can also infer the data-complexity of $\mathbb{BD}$ programs and thus of the above extensions of DATALOG$_{nS}$. In case of $\mathbb{BD}$ programs, if the set of rules is fixed and the data varies, the translation into a core program (see Section 4.1) is polynomial in the size of ground facts. For this reason, our upper bounds for the case of bounded number of variables correspond to upper bounds for data complexity of $\mathbb{BD}$ programs. In particular, w.r.t. data-complexity, the above reasoning tasks are in EXPTIME and in 2-EXPTIME for normal $\mathbb{BD}$ programs and for full $\mathbb{BD}$ programs, respectively. It is also not hard to see that these bounds are tight. The EXPTIME lower bound follows from the data-complexity in DATALOG$_{nS}$. The 2-EXPTIME lower bound in the disjunctive case can be obtained by modifying the reduction in Section 4.3.1, using the well-known notion of a *meta-interpreter*. That is, we encode all the details of the Turing machine and its input into facts (not necessarily unary), in a way that the rest of the program does not refer to a particular machine or input, but instead it 'interprets' the content of the facts.[1] The lower bounds on data complexity in disjunctive $\mathbb{BD}$ programs also apply for disjunctive DATALOG$_{nS}$ programs.

5.7 Reductions of Description Logics to ASP

Reductions of description logics to ASP have been considered e.g. in [AB01, Bar02, Swi04, HMS04, HV03, HNV05]. Alsaç and Baral [AB01, Bar02] 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 [Swi04] 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. [HV03, HNV05] reduced $\mathcal{SHIQ}$ (which subsumes $\mathcal{ALCQI}$) to their Conceptual Logic Programs and extensions; however, they used answer sets over open domains.

Most relevant for this thesis is the work in [HMS04, Mot06]. The authors reduced reasoning in a $\mathcal{SHIQ}$ knowledge base to the evaluation of a positive disjunctive DAT-

[1]The single tricky part here is dealing with m-bit addresses which were encoded using (input dependent) unary predicate names $B^1_1, \ldots, B^1_m, B^0_1, \ldots, B^0_m$. We can get rid of these predicates by establishing and maintaining a linear order over a set D of m designated constants. In particular, an enumeration $d_1, \ldots, d_m$ of D can established using facts $FIRST(c, d_1)$, $LAST(c, d_n)$ and $NEXT(c, d_i, d_{i+1})$, where $1 \leq i < m$. All functional terms can be made aware of the order using rules of the form $A(f(X), y_1, \ldots, y_n) \leftarrow A(X, y_1, \ldots, y_n)$. At a term t, a value of the ith bit can be encoded in an atom $VAL(t, d_i, v)$ where v is one of the two constants $1, 0$ designated for truth and falsity. The address counter in the reduction can be easily modified to operate on this structure.

ALOG program. The program is generated in three steps. First, the knowledge base is translated into first-order logic in the standard way. After that, resolution and superposition techniques are applied to saturate a clausal form of the transformation. Finally, functional terms are removed using new constant symbols.

The reduction of $\mathcal{ALC}$ to $\mathbb{FDC}$ in Section 3.3.2 has some similarities to the one of Hustadt et al. described above. The main differences are with respect to their second step, where our method uses knots for compilation, and that our method aims at model building while the one in [HMS04] is geared towards *instance checking*. Notably, the disjunctive DATALOG program constructed in [HMS04] is generally exponential in the size of the initial DL knowledge base (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 is rewritten (very efficiently) on the DL syntax side into a normal form, rather than on the first-order logic side after the mapping. Second, a transformation into $\mathbb{FDC}$ opens the possibility to use any dedicated evaluation algorithm for such programs, beyond a specific method (like the one in this thesis).

5.8 Reasoning about Actions and Planning

As already discussed in the previous sections, the use of nonmonotonic logic programs under answer set semantics as a tool for solving problems in reasoning about actions has been considered in many papers, including [DNK97, Lif99, Bar02, EFL$^+$04, TSB07, SBTM06, STGM05, MTS07]. The work presented in this thesis adds to these other works by providing an underpinning of the computational properties of nonmonotonic logic programs with functions symbols that naturally emerge in this context, and, importantly, capture indefinitely long action sequences. Our programs may help in assessing the complexity of particular planning problems and may be useful to show that tractability can be achieved in some cases. Furthermore, our algorithms may also be exploited in this area.

5.9 Mosaics and Types

The knot technique can be seen as an instance of other reasoning methods that have been used for modal and description logics, and other related fragments of first-order logic. In particular, knots are a special instance of the *mosaic* technique [Ném86] that is well known in the context of modal logics. The basic idea underlying the technique is that models can be decomposed into a finite collection of small model parts called *mosaics*, and that if a finite set of mosaics is suitably *linked*, its elements can be combined into a model. Mosaics were first introduced in [Ném86] and since then they have been used for several modal logics, especially for logics with multidimensional features. For an excellent exposition of the mosaic technique and a comprehensive list of references, we refer to [MV07, BdRV01]. Mosaics are usually applied to show that the formula satisfiability problem for a given logic is decidable and, in fewer cases, also for deriving tight complexity upper bounds. The precise notion of mosaic, the local conditions, and the definition of the links between mosaics are always tailored for the specific logic under consideration.

Knots and mosaics are closely related to *types*. Roughly, a type is a small mosaic with only one element, and in compensation for the simplicity of the mosaics, more involved global conditions may be required. In fact, the elimination algorithm in Section 3.3.1 is a variant of the famous *type elimination* algorithm proposed by Pratt for Propositional Dynamic Logic [Pra79]. This kind of type elimination algorithms have been applied to a wide range of logics including, for example, various modal and description logics [PSV06, HM92, LWZ08], the guarded fragment and 2-variable fragments of first order logic [ANvB98, GKV97].

We applied knots to build and reason about the stable models of a program with default negation, which is complicated because it requires minimization of models. Thus, in addition to ensuring the satisfaction of rules, we had to define special conditions to ensure that knots can be assembled into stable models. To the best of our knowledge, mosaic-like techniques had not been applied to minimal model reasoning before, at least not in the setting of Logic Programming.

Chapter 6

Conclusion

The goal of this thesis was to identify fragments of ASP with function symbols that are expressive enough to allow for common-sense reasoning in applications with potentially infinite domains, and at the same time are still decidable and have good computational properties. We remind that our goal is nontrivial because coupling ASP with function symbols easily leads to high undecidability. Our research was motivated by the fact that current decidable ASP languages practically do not support function symbols for a generic representation of problems involving infinite processes, recursive data structures, and other problems that require an unbounded number of domain objects.

6.1 Our Results

Our chief contributions are two decidable languages, $\mathbb{FDNC}$ and $\mathbb{BD}$ programs, that are expressive fragments of ASP with function symbols facilitating reasoning over infinite domains. The languages push the frontier of decidable, yet easy to recognize programs. Indeed, unlike most of other relevant fragments, our languages are defined by syntactic restrictions that modularly apply on the rules and can be easily checked (in polynomial time for all fragments except for full function-safe $\mathbb{BD}$ programs). Apart from the decidability results, the thesis provides a detailed characterization of the computational complexity of several reasoning problems in $\mathbb{FDNC}$ programs, $\mathbb{BD}$ programs, and many restricted subfragments that are obtained by disallowing or limiting the use of various constructs. As a side results, we obtain also complexity results for extensions of $\text{D}\textsc{atalog}_{nS}$ [CI93, Cho95] with disjunction and negation under the answer set semantics (see Section 5.6). Table 6.1 gives a comprehensive overview of the complexity of reasoning in the developed fragments (see also Table 2.1 for the relevant existing results).

The restrictions developed in this thesis are, in fact, liberal enough to allow an encoding of some relevant problems. In particular, $\mathbb{FDNC}$ allows to encode transition-based planning problems, and also to simulate some expressive description logics. $\mathbb{BD}$ programs effectively extend $\mathbb{FDNC}$ with additional expressiveness. This is witnessed, for example, in temporal domains, where $\mathbb{BD}$ programs may refer to the *future* and the *past*, while $\mathbb{FDNC}$ is limited to one modality only. Thus $\mathbb{BD}$ programs have expressive means to change historic values of fluents and to deal with surprises (see, e.g., [SZ95] for a

Languages	Consistency	$P \models_b A(\vec{t})$	$P \models_b \exists\vec{x}.A(\vec{x})$	$P \models_c A(\vec{t})$	$P \models_c \exists\vec{x}.A(\vec{x})$
$\mathbb{F}$	trivial	P	PSPACE	P	PSPACE
$\mathbb{FD}$	trivial	Σ_2^P	PSPACE	co-NP	EXPTIME
$\mathbb{FC}$	PSPACE	PSPACE	PSPACE	PSPACE	PSPACE
$\mathbb{FDC}$, $\mathbb{FN}$, $\mathbb{FNC}$, $\mathbb{FDNC}$	EXPTIME	EXPTIME	EXPTIME	EXPTIME	EXPTIME
disjunctive $\mathbb{BD}$ programs	2-EXPTIME	2-EXPTIME	2-EXPTIME	2-EXPTIME	2-EXPTIME
disjunctive $\mathbb{BD}$ programs with one function symbol	EXPSPACE	EXPSPACE	EXPSPACE	EXPSPACE	EXPSPACE
normal $\mathbb{BD}$ programs	EXPTIME	EXPTIME	EXPTIME	EXPTIME	EXPTIME
normal $\mathbb{BD}$ programs with one function symbol	PSPACE	PSPACE	PSPACE	PSPACE	PSPACE
function-safe disjunctive $\mathbb{BD}$ programs	NEXPTIME	NEXPTIME	NEXPTIME	CO-NEXPTIME	CO-NEXPTIME
function-safe disjunctive $\mathbb{BD}$ programs with one function symbol	Σ_2^P	Σ_2^P	Σ_2^P	Π_2^P	Π_2^P
function-safe normal $\mathbb{BD}$ programs	PSPACE	PSPACE	PSPACE	PSPACE	PSPACE
function-safe normal $\mathbb{BD}$ programs with one function symbol	NP	NP	NP	co-NP	co-NP

Table 6.1: Summary of complexity results for $\mathbb{FDNC}$ programs, $\mathbb{BD}$ programs, and their fragments (completeness results). The results for $\mathbb{BD}$ programs assume bounded number of variables. See also Table 3.1 for open queries in $\mathbb{FDNC}$, which were not considered for $\mathbb{BD}$ programs.

discussion of surprise handling in planning, which aims at dealing with (unexpected) observations by recomputing or updating a plan). On the other hand, $\mathbb{BD}$ programs can simulate more expressive description logics, in particular the ones supporting inverse roles (e.g., $\mathcal{ALCI}$). They also provide power tools to manipulate tree-shaped structures, e.g., HTML or XML documents, with the support for common-sense reasoning via default negation.

The main technical challenge in our quest was dealing with minimality. Recall that, unlike the classical semantics of first-order logic, the stable model semantics requires testing minimality of candidate models. In the presence of infinite domains, and even more if disjunction is allowed, this becomes a nontrivial task. Indeed, for an infinite Herbrand interpretation I, there are uncountably many smaller interpretations $J \subset I$.

Since $\mathbb{FDNC}$ and $\mathbb{BD}$ programs can have infinitely large stable models, we had to deal with this problem. Intuitively, minimality is a *global* (second-order) condition on interpretations. Thus even though our syntactic restrictions are inspired in description and modal logics, the results from these fields do not carry over easily to our setting. In fact, the absence of global conditions is attributed as a main factor for decidability of description and modal logics (see Chapter 7 in [GKL$^+$07]).

To show decidability and worst-case optimal complexity results for $\mathbb{FDNC}$ programs, we have developed the *knot technique* to finitely represent the stable models of a program. In particular, we have shown that for any $\mathbb{FDNC}$ program there exists a finite set of building blocks, or *knots*, such that each stable model of the program can be reconstructed by gluing them together. The method—related to the mosaic technique known from modal logics (see Section 5)—allowed us to infer a variety of worst-case optimal upper bounds for $\mathbb{FDNC}$ and a wide range of its fragments. These complexity results are summarized in Table 6.1 (see also Table 3.1 for open queries and references to the specific proofs).

The syntax of $\mathbb{FDNC}$ is quite complicated, but it ensures two good features. Firstly, the restrictions ensure that the stable models of a program have the shape of a forest, i.e., a collection of tree-shaped structures, which allows us to decompose them into knots. Secondly, they ensure finite recursiveness, which in turn means that stable models can be built in stages and minimality testing can be done without explicitly quantifying over infinite interpretations. These restrictions put the complexity of reasoning in $\mathbb{FDNC}$ in line with the complexity of reasoning in related fragments of first-order logic. In other words, even though the stable model semantics in $\mathbb{FDNC}$ requires minimality testing, the overall complexity is not higher than that of reasoning under the classical first-order semantics in standard description logics, like $\mathcal{ALC}$.

The syntax of $\mathbb{BD}$ programs is much simpler, but this also leads to the loss of the positive impact of finite recursiveness and brings us to another level of expressiveness and complexity. Our main complexity results for $\mathbb{BD}$ programs are also summarized in Table 6.1. We recall that in case unbounded number of variables is allowed in the rules, the complexity of reasoning in $\mathbb{BD}$ jumps by an exponential (see Section 4.5). Importantly, in $\mathbb{BD}$ programs we may require finiteness of stable models, and can write a program that has infinitely many stable models where each of them is finite. Clearly, finiteness is a global condition that cannot be verified by 'looking' at finite parts of a candidate interpretation. For this reason, the knot technique, in its current form, does not extend to $\mathbb{BD}$ programs. One possible way to deal with this seems to be by storing additional non-logical information (e.g., counters) in knots. Automata over infinite trees appear to be more suitable for reasoning in $\mathbb{BD}$ programs. As we have seen, for a given $\mathbb{BD}$ program we can build a tree automaton that accepts (or, equivalently, generates) exactly the stable models of the program. The automaton can be viewed as a finite representation of the stable models, although less constructive than the knot-based representation. An important component in our construction was the characterization of minimal models of disjunctive programs in terms of *split programs* (Theorem 4.29), which allows to reduce the minimality test of a model for a disjunctive program to a set of minimality tests for nondisjunctive programs.

Tree automata can in principle be applied for $\mathbb{FDNC}$ programs as well, but formally showing that the stable models of $\mathbb{FDNC}$ programs can be build in stages, similarly as with knots, seems inevitable. We finally note that decidability of $\mathbb{FDNC}$ and $\mathbb{BD}$ programs can also be shown by an encoding into monadic second-order logic over trees (SkS) [Rab69], however this does not give optimal complexity bounds. In general, SkS is non-elementary, and we are not aware of complexity characterizations for (prefix) fragments of SkS that would be applicable in our setting.

6.2 Future Outlook

Some limitations and possible extensions of $\mathbb{FDNC}$ and $\mathbb{BD}$ programs can also be identified. Firstly, our languages provide only a limited support for atoms with unbounded number of arguments. This can be partially solved by considering various *guardedness* restrictions (cf. [ANvB98, Var96, Grä99, CGK08]), which ensure the (generalized) tree-shape model property. We believe that our programs can be viewed as *principle* languages in the sense that the methods and techniques applied for our languages can be generalized to guarded rules, in the same way as algorithms for the various guarded fragments of first-order logics are derived from the ones for the corresponding modal logics.

Important work on guarded rules was done in [CGK08], where the authors consider conjunctive query answering under expressive database constraints. In particular, they work on *guarded tuple generating dependencies*, which in our setting can be viewed as rules that have function symbols but adhere to some guardedness restrictions. The restrictions require certain rule variables to occur together in a body atom. The query answering algorithm that was developed in [CGK08] is sophisticated because conjunctive queries cannot be stated as constraints without violating the guardedness restrictions. We believe that a very expressive fragment of ASP with function symbols can be built by relaxing the guardedness restrictions in [CGK08] and in this way allowing for conjunctive queries and their generalizations as part of the language. As a first step in this direction, we have developed $\mathbb{GT}$ *programs* (*graph-tree programs*), which capture $\mathbb{FDNC}$ and $\mathbb{BD}$ programs, and support a generalization of conjunctive queries to recursive queries as part of the language. More precisely, we employ a condition that we call *head-guardedness*, which requires all variables in the head of the rule to appear in some body atom. In this way, e.g., $\mathbb{GT}$ programs allow to extend the DL $\mathcal{ALCI}$ with recursive rules that generalize conjunctive queries. Our preliminary work on $\mathbb{GT}$ programs is presented in Appendix C, where we define the language and present a 3-EXPTIME upper bound for consistency testing (only a 2-ExpTime lower bound is known). A precise characterization of the complexity of reasoning in $\mathbb{GT}$ programs remains for future work.

The languages presented in this thesis are rule languages that allow to simulate existential quantification in description logics via function symbols. We believe that $\mathbb{FDNC}$, $\mathbb{BD}$ and $\mathbb{GT}$ programs are important for the future development of formalisms that integrate rules and description logics. As it was noted in the introduction, such languages are of interest in Knowledge Representation as they are ought to provide the expressive features of two largely orthogonal paradigms. The more immediate applications of such languages are in the Semantic Web, where declarative rule-based access to description logic ontologies is desirable.

Another possible direction for future research is to provide a more flexible support for function symbols with higher arities. Recall that (using a suitable rewriting) function symbols in our programs can always be viewed as unary. Certainly, allowing to construct more complex terms using nonunary function symbols is of interest, especially in the context of recursive data structures. However, ensuring decidability in this setting is a largely unexplored area.

An implementation of $\mathbb{FDNC}$ and $\mathbb{BD}$ programs is also a subject of future work. As we have noted already, for $\mathbb{FDNC}$ an implementation of our knot-based algorithms seems viable. Since stable knots—which are basic model building pieces for stable models of $\mathbb{FDNC}$ programs—are defined in terms of stable models of finite propositional programs, exploiting highly optimized answer set solvers to do part of the reasoning is feasible. Before implementing a reasoner for $\mathbb{BD}$ programs, however, we need to obtain algorithms that are more direct than the automata-based approach described in this thesis. Indeed, automata encodings are a powerful tool for reasoning in expressive formalisms, but they also lead to a significant loss of the problem structure, which can otherwise be exploited for optimization purposes.

Appendix A

Auxiliary Results

A.1 Auxiliary Lemma

Lemma A.1. *(Lemma 3.37 on page 48) Let C be a complexity class in Table 3.1, and let $\mathcal{L}$ be from the $\mathbb{F}$ family. Then:*

(i) If deciding program consistency for $\mathcal{L}$ is C-hard, then deciding brave entailment of queries (ground or existential, unary or binary) is also C-hard for $\mathcal{L}$.

(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 $\mathcal{L}$.

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

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 and v,t are ground.

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. $\square$

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 A.1: Rules for Rewriting into Normal Form

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

Proposition A.2. *We can 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).*

Proof. 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 A.1. 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 the rewriting in each of the phases. Following the standard assumption in description logics, we assume that each of the atomic concepts in **C** is of constant size, i.e., the length of the binary string representing an atomic concept does not depend on the particular knowledge base. The size $|\mathcal{K}|$ of a knowledge base $\mathcal{K}$ amounts then to the number of symbols in the string representing the axioms of $\mathcal{K}$. Without loss of generality, we assume that $\mathcal{K}$ contains only one axiom α (note that, in general, each axiom in a knowledge base can be rewritten independently).

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 a 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 (cf. [BCM$^+$03]), 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 $\mathcal{K}$. $\square$

Appendix B

Open Queries in $\mathbb{FDNC}$: Lower Bound

In Section 3.3.4 we have shown that checking cautious entailment of open queries in $\mathbb{FD}$, $\mathbb{FN}$, $\mathbb{FNC}$, $\mathbb{FDC}$ and $\mathbb{FDNC}$ is feasible in exponential space. We prove here that the algorithm is worst-case optimal.

Lemma B.1. *Cautious entailment of open queries in* $\mathbb{FD}$, $\mathbb{FN}$, $\mathbb{FNC}$, $\mathbb{FDC}$ *and* $\mathbb{FDNC}$ *programs is* EXPSPACE*-hard.*

Proof. Consider a language L over an alphabet Σ in EXPSPACE. Then there is a deterministic Turing machine $M = (Q, \Sigma, q_0, \delta)$ as in Definition 2.11 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 $\lambda x.A(x)$ under cautious entailment. By I_k we denote the kth symbol in the input string $I = I_0, \ldots, I_{|I|-1}$.

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

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

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

$$rwoequ(x) \leftarrow equ_0(x), \ldots, equ_{as}(x). \tag{B.2}$$

The inequality is also easily expressed by the rules

$$nonequ(x) \leftarrow opos_i^b(x), rwpos_i^{\bar{b}}(x) \tag{B.3}$$

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

$$rwpos_i^0(st) \;\leftarrow\; , \tag{B.4}$$
$$state_{q_0}(st) \;\leftarrow\; , \tag{B.5}$$
$$opos_i^1(st) \vee opos_i^0(st) \;\leftarrow\; . \tag{B.6}$$

Intuitively, (B.4) sets the position of the r/w head to the left most cell and (B.5) set the machine into the start state, while (B.6) 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

$$symbol_\alpha(st) \leftarrow opos_0^{b_0}(st), \ldots, opos_{as}^{b_{as}}(st). \tag{B.7}$$

For all other positions, the symbols are blank. Assuming that $n = b_0^* \cdots b_{as}^*$ in binary, we express this with rules

$$symbol_b(st) \leftarrow opos_1^{b_1^*}(x), \ldots, opos_{j-1}^{b_{j-1}^*}(x), opos_j^1(x), \tag{B.8}$$

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:

$$next(x, \bar{t}(x)) \leftarrow rwoequ(x), state_s(x), symbol_\alpha(x), \tag{B.9}$$
$$next(x, \bar{t}(x)) \leftarrow nonequ(x), state_s(x), \tag{B.10}$$
$$state_{s'}(\bar{t}(x)) \leftarrow next(x, \bar{t}(x)), \tag{B.11}$$
$$symbol_{\alpha'}(\bar{t}(x)) \leftarrow rwoequ(x), next(x, \bar{t}(x)), \tag{B.12}$$
$$move_D(\bar{t}(x)) \leftarrow next(x, \bar{t}(x)). \tag{B.13}$$

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 (B.9). If the r/w head is not at the position of the observed cell, the transition is made blindly (B.10). 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 (B.11) sets the new state, while (B.12) sets the new symbol of the observed cell. The rule (B.13) 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

$$shift^1_{as}(x) \leftarrow move_{+1}(x), \tag{B.14}$$
$$shift^1_{as}(x) \leftarrow move_{-1}(x), \tag{B.15}$$
$$shift^0_{as}(x) \leftarrow move_0(x). \tag{B.16}$$

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

$$rwpos^{\bar{b}}_j(y) \leftarrow shift^1_j(y), rwpos^b_j(x), next(x, y), \tag{B.17}$$
$$rwpos^b_j(y) \leftarrow shift^0_j(y), rwpos^b_j(x), next(x, y), \tag{B.18}$$
$$shift^b_{j'-1}(y) \leftarrow move_{+1}(y), shift^1_{j'}(y), rwpos^b_{j'}(x), next(x, y), \tag{B.19}$$
$$shift^{\bar{b}}_{j'-1}(y) \leftarrow move_{-1}(y), shift^1_{j'}(y), rwpos^b_{j'}(x), next(x, y), \tag{B.20}$$
$$shift^0_j(x) \leftarrow move_0(x). \tag{B.21}$$

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

$$opos^b_i(y) \leftarrow opos^b_i(x), next(x, y), \tag{B.22}$$

for each $i \in \{0, \dots, as\}$ 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 $\alpha \in \Sigma$:

$$symbol_\alpha(y) \leftarrow nonequ(x), symbol_\alpha(x), next(x, y). \tag{B.23}$$

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 q_{accept}. We establish the following lemmas.

Lemma B.2. *If the machine M accepts the input I on the run $t_1, \ldots, t_n$, $n \geq 1$, then $P(M, I) \models_c state_{q_{accept}}(u)$, 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 $state_{q_{accept}}(u) \in R^w$. 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) $state_{q_j}(u_j) \in R^w$, 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 $s^n = q_{accept}$.

We proceed by induction on $j \geq 0$. The base case $j = 0$ is clear by the encoding of the initial word (rules (B.7) and (B.8)), the initial r/w head position (facts (B.4)) and the initial state (fact (B.5)).

For the inductive case, assume the claim holds for $0 \leq j < n$ and consider $j + 1$. By the induction hypothesis, $symbol_\alpha(u_i) \in R^w$, $state_{q_j}(u_j) \in R^w$, and pos^j is described by the atoms $rwpos^b_i(u_j) \in R^w$. There are now, by the rules (B.1) – (B.3) 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 (B.10); by the rule (B.23), 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 (B.9); by the rule (B.12), 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 (B.11) we have $state_{q_{i+1}}(u_{i+1}) \in R^w$, and thus (ii) holds for $j + 1$. Finally, the rules (B.13) and (B.14) – (B.21) effect that atoms $pos^b_i(u_{j+1})$ which correctly represent pos^{j+1} are derived. Hence, (iii) holds for $j + 1$. ☐

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

Proof. Suppose $P(M, I) \models_c state_{q_{accept}}(u)$. By assumption, the initial state is not q_{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, $state_{q_{accept}}(u_n)$ 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 $s^n = q_{accept}$, 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, $s^0 = q_0$, and $pos^0 = 0$ by the facts and rules (B.4) – (B.8). 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 (B.11); 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 (B.1) – (B.3), (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 (B.12), and in case (b) $symbol_\alpha(u_{i+1}) \in R^w$ by rule (B.23), 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 (B.13); by the induction hypothesis and the rules (B.14) – (B.21), 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 $\lambda x.state_{q_{accept}}(x)$ are constructible in polynomial time from M and I, from Lemmas B.2 and B.3 the claimed EXPSPACE-hardness result follows for $\mathbb{FD}$, $\mathbb{FDN}$, and $\mathbb{FDNC}$; by replacing the disjunctive guessing rules (B.6) 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 $\mathbb{FNC}$. □

Appendix C

An Upper-Bound for $\mathbb{GT}$ Programs

We define here $\mathbb{GT}$ programs (*graph-tree programs*), and provide an upper-bound for testing their consistency. As it was noted in Section 6.2 (page 146), $\mathbb{GT}$ programs are an expressive fragment that captures the other fragments we have defined, and also allows for recursive rules with arbitrary number of variables. To achieve this expressiveness without compromising decidability, we employ a condition that we call *head-guardedness*.

$\mathbb{GT}$ programs are defined as follows:

Definition C.1. *A program P is called a $\mathbb{GT}$ program if the following conditions are satisfied:*

1) All relations in P are unary or binary.

2) All ground rules are facts of the form $A(c) \leftarrow$ and $R(c,d) \leftarrow$, where c, d are constants.

3) Constant occur in facts only.

4) The rules with variables have the following properties:

a) Binary atoms are of the form $R(x,y)$, $R(x,f(x))$ or $R(f(x),x)$, where $x \neq y$;

b) Unary atoms are of the form $A(x)$ or $A(f(x))$;

c) Rules are safe, i.e., each variable occurs in some positive body atom;

d) (Head guardedness) If H is an atom in the head of a rule, then there is a positive body atom that contains all the variables in H;

Both $\mathbb{FDNC}$ and core $\mathbb{BD}$ programs are subsumed by $\mathbb{GT}$ programs; full $\mathbb{BD}$ programs can be encoded via the translation into core $\mathbb{BD}$ programs (see Section 4.1). Observe also that the body of a rule in a $\mathbb{GT}$ program can be seen as an arbitrary labeled graph over variables (this explains '$\mathbb{G}$'). However, the head-guardedness condition ensures that using such rules we can only create tree-shaped structures (thus '$\mathbb{T}$').

Proposition C.2. *If I is a stable model of a $\mathbb{GT}$ program P, then each binary atom in I has form $R(c,d)$, $R(t,f(t))$ or $R(f(t),t)$, where c, d are constants and t is a term.*

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

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

In the remainder of this section we show how consistency and other standard reasoning tasks for $\mathbb{GT}$ programs can be decided by employing tree automata. We note that, similarly as for $\mathbb{FDNC}$ and $\mathbb{BD}$ programs, decidability of $\mathbb{GT}$ programs can be inferred from the decidability of SkS. Nevertheless, we provide a direct a automata-based algorithm, that allows us to obtain a 3ExpTime upper bound. We build on the method used in [CEO07] for answering (extensions of) conjunctive queries over some description logics, but adapt it to handle arbitrary head-guarded rules, and to additionally test the minimality condition in the definition of stable models.

To provide an algorithm, we first need to represent Herbrand interpretations of $\mathbb{GT}$ programs as trees.

Definition C.3. *Let P be a $\mathbb{GT}$ program. Let $a_1, \dots, a_n, f_{n+1}, \dots, f_m$ be an enumeration of the constants and function symbols occurring in P, where each a_i is a constant and each f_j is a function symbol. We define*

$$\begin{aligned} \mathbf{C} = &\ \{1, \dots, n\}, \textit{ and} \\ \mathbf{F} = &\ \{n+1, \dots, m\}. \end{aligned}$$

A word $w \in \mathbf{C} \times \mathbf{F}^*$ is called a term node. *Each term node $w = i \cdot j_1 \cdots j_k$ encodes the term* $\mathsf{term}(w) = f_{j_k}(\dots f_{j_1}(a_i)\dots)$.

Let L^P be the set of unary predicate names that contains:

(T1) each unary A occurring in P;

(T2) fresh unary predicates R_f and R_f^- for each binary R and each function symbol f occurring in P;

(T3) a fresh unary predicate $R_{c,d}$ for each binary R and each pair of constants c, d occurring in P.

Intuitively, R_f and R_f^- are used to encode atoms of the form $R(t, f(t))$ and $R(f(t), t)$, respectively. The unary $R_{c,d}$ will be used to encode the atom $R(c, d)$.

We define the alphabet $\Sigma_P = 2^{L^P}$, and we call a tree $\mathcal{T} = (T, \mathcal{L})$ over Σ_P proper, *if the following are true for every $n \in T$:*

(P1) if $\mathcal{L}(n)$ contains some predicate of type (T3), then $n = \epsilon$;

(P2) if $\mathcal{L}(n)$ contains some predicate of type (T1) or (T2), then n is a term node;

(P3) if $\mathcal{L}(n) \neq \emptyset$, then $n = \epsilon$ or n is a term node.

A proper tree $\mathcal{T} = (T, \mathcal{L})$ over Σ_P is a representation of a Herbrand interpretation for P. Indeed, the root ϵ of $\mathcal{T}$ stores the binary atoms of the form $R(c, d)$. The children of ϵ correspond to constants of P, and the $\mathbf{F}^+$ descendants of constants correspond to functional terms. The labeling of term nodes provides the predicates that are satisfied in an interpretation. More formally, we have:

Definition C.4. *A proper tree $\mathcal{T} = (T, \mathcal{L})$ over Σ_P* encodes *the interpretation* $\mathsf{int}(\mathcal{T})$ *consisting of:*

(i) $R(c, d)$, for each $R_{c,d} \in \mathcal{L}(\epsilon)$;

(ii) $A(\mathsf{term}(w))$, for each term node $w \in T$ such that A is a unary predicate in P and $A \in \mathcal{L}(w)$;

(iii) $R(\mathsf{term}(w), f(\mathsf{term}(w)))$ for each term node $w \in T$ with $R_f \in \mathcal{L}(w)$;

(iv) $R(f(\mathsf{term}(w)), \mathsf{term}(w))$ for each term node $w \in T$ with $R^-_f \in \mathcal{L}(w)$.

We say that an automaton with alphabet Σ_P is proper *if every tree it accepts is proper, and we say that an automaton A* accepts an interpretation *I if there is a proper $\mathcal{T}$ such that* $\mathsf{int}(\mathcal{T}) = I$ *and A accepts $\mathcal{T}$.*

Observe that any interpretation I with binary atoms only of the form $R(c, d)$, $R(t, f(t))$ or $R(f(t), t)$ can be represented as a proper tree, i.e., there exists proper $\mathcal{T}$ with $\mathsf{int}(\mathcal{T}) = I$. Then, by Proposition C.2, we get:

Proposition C.5. *Let P be a $\mathbb{GT}$ program. For any stable model I of P, there exists a proper tree $\mathcal{T}$ with* $\mathsf{int}(\mathcal{T}) = I$.

Thus to test consistency of P, it suffices to build an automaton A_P^{sm} that accepts exactly the proper trees $\mathcal{T}$ such that $\mathsf{int}(\mathcal{T})$ is a stable model of P.

We use another kind of trees that represent a *pair* of Herbrand interpretations for a given $\mathbb{GT}$ program.

Definition C.6. *Let P be a $\mathbb{GT}$ program, and let $\mathcal{T} = (T, \mathcal{L})$ be a tree over $\Sigma_P \times \Sigma_P$. We denote by $\mathcal{T}|_1 = (T, \mathcal{L}_1)$ (resp., $\mathcal{T}|_2 = (T, \mathcal{L}_2)$) the tree over Σ_P such that, for each $n \in T$, $\mathcal{L}_1(n)$ (resp., $\mathcal{L}_2(n)$) is the first (resp., second) component of $\mathcal{L}(n)$.*

We say that $\mathcal{T}$ is proper *if $\mathcal{T}|_1$ and $\mathcal{T}|_2$ are proper. If $\mathcal{T}$ is proper, then it* encodes *the pair of interpretations (I_1, I_2), where $I_1 =$* $\mathsf{int}(\mathcal{T}|_1)$ *and $I_2 =$* $\mathsf{int}(\mathcal{T}|_2)$. *We say that an* *automaton A with alphabet $\Sigma_P \times \Sigma_P$ is* proper *if every tree it accepts is proper, and we say that an automaton A accepts (I_1, I_2) if there is a proper $\mathcal{T}$ over $\Sigma_P \times \Sigma_P$ such that $\mathcal{T}$ encodes (I_1, I_2) and A accepts $\mathcal{T}$.*

Our construction also requires an automaton $A|$ with alphabet Σ_P that recognizes the second component $\mathcal{T}|_2$ of each tree $\mathcal{T}$ accepted by an automaton A with alphabet $\Sigma_P \times \Sigma_P$. Such a projection automaton can be easily constructed.

Definition C.7. *Let A be a 1NTA with alphabet $\Sigma_P \times \Sigma_P$, and let A' be an automaton with alphabet a Σ_P. We say that A' is a* projection automaton *for A if the following two conditions are satisfied:*

- *If A accepts a tree $\mathcal{T}$, then A' accepts the tree $\mathcal{T}|_2$ (i.e., the tree obtained taking only the second component of the labels of $\mathcal{T}$).*
- *If A' accepts a tree $\mathcal{T}'$ over Σ_P, then there is a tree $\mathcal{T}$ over $\Sigma_P \times \Sigma_P$ such that $\mathcal{T}|_2 = \mathcal{T}'$ and A accepts $\mathcal{T}$.*

For a 1NTA $A = (\Sigma_P \times \Sigma_P, Q, \delta, q_0, F)$, we define

$$A| = (\Sigma_P, Q, \delta', q_0, F)$$

where, for each $N' \in \Sigma_P$ and each state $q \in Q$,

$$\delta'(N', q) = \bigvee_{N \in \Sigma_P} \delta\big((N, N'), q\big).$$

The following is easy to check:

Proposition C.8. *For every nondeterministic 1-way tree automaton A with alphabet $\Sigma_P \times \Sigma_P$, $A|$ is a projection automaton for A.*

Now we are ready to explain how to build an automaton A_P^{sm} that accepts the stable models of a $\mathbb{GT}$ program P, by combining the following nondeterministic 1-way tree automata. We will show later how these automata can be constructed.

Proposition C.9. *Let P be a given $\mathbb{GT}$ program. Then the following proper 1NTA can be constructed:*

(a) (Counter-example automaton) A_P^{ce} that accepts exactly the pairs (I, I') such that $I \not\models P^{I'}$. The automaton has exponential alphabet and exponentially many states, and the index of the parity condition is fixed.

(b) $A_P^{\subseteq}$ that accepts exactly the pairs (I, I') such that $I \subset I'$. The automaton has exponential alphabet, but a fixed number of states states and a fixed parity condition.

(c) $A_P^{=}$ *that accepts exactly the pairs* (I, I') *such that* $I = I'$*. As* $A_P^{\subsetneq}$*, the automaton has exponential alphabet, but a fixed number of states states and a fixed parity condition.*

The automaton A_P^{sm} can be built by transforming and combining the above automata as follows:

- Let $A_1 = \overline{A_P^{ce}} \cap A_P^{=}$, i.e., A_1 is the intersection automaton for the complement of A_P^{ce}, and the automaton $A_P^{=}$. Then A_1 accepts pairs of interpretations (I, I') where $I = I'$ and $I \models P^{I'}$.

- We project away the first interpretation in the language of A_1 and keep only the second: simply let $A_P^{mods} = A_1|$ be the projection automaton of A_1. Then A_P^{mods} accepts an interpretation I' iff $I' \models P^{I'}$.

- The next step is to device an automaton that verifies whether a model I' of the reduct $P^{I'}$ is minimal. To this aim, we first let $A_2 = \overline{A_P^{ce}} \cap A_P^{\subsetneq}$, be the intersection automaton for the complement of A_P^{ce}, and the automaton $A_P^{\subsetneq}$. Then A_2 accepts a pair (I, I') iff $I \models P^{I'}$ and $I \subset I'$.

- We take the projection automaton $A_2|$, which accepts I' iff there is some $I \subset I'$ with $I \models P^{I'}$. That is, $A_2|$ accepts I' if there is some I witnessing that I' is not a minimal model of its reduct $P^{I'}$.

- We take the complement $A_P^{min} = \overline{(A_2|)}$ of $A_2|$, which accepts I' if there is no $I \models P^{I'}$ with $I \subset I'$. That is, A_P^{min} accepts an interpretation I' only if no smaller interpretation is a model of the reduct. Note that A_P^{min} does not ensure that I' is a model of $P^{I'}$. It only ensures that if it is a model, then it is minimal.

- Finally, we intersect A_P^{min} with the automaton A_P^{mods}, which accepts I' iff it is a model of $P^{I'}$. That is, the desired automaton $A_P^{sm} = A_P^{min} \cap A_P^{mods}$ accepts an interpretation I' iff $I' \models P^{I'}$ and there is no $I \subset I'$ with $I \models P^{I'}$.

Thus consistency of P can be decided by checking non-emptiness of A_P^{sm}.

Theorem C.10. *A* $\mathbb{GT}$ *program* P *is consistent iff the language of* A_P^{sm} *is non-empty.*

For the complexity of reasoning, we note that due to the complementation step when constructing A_P^{mods}, the automaton may have double exponentially many states and an exponential parity condition in the size of P (see, e.g., [MS95] for the complexity of complementing 1NTAs). On the other hand, due to the double complementation for A_P^{min}, the automaton may have a triple exponential number of states and a parity condition of double exponential size. Thus testing nonemptiness of the resulting automaton A_P^{sm} is feasible in triple exponential time in the size of P.

Theorem C.11. *Checking consistency of* $\mathbb{GT}$ *programs is in* 3EXPTIME.

The precise complexity of consistency testing remains open, although we believe the problem is solvable in 2-EXPTIME by applying the methods developed for $\mathbb{BD}$-programs. It is not hard to show that the problem is 2-EXPTIME-hard, using a straightforward reduction of the *conjunctive query entailment problem* in the DL $\mathcal{ALCI}$, which was shown to be 2-EXPTIME-hard in [Lut07]. Indeed, an $\mathcal{ALCI}$ knowledge base $\mathcal{K}$ can be encoded into a positive $\mathbb{GT}$ program $P_{\mathcal{K}}$ (see Table 3.2 for an encoding of $\mathcal{ALC}$, which can be easily lifted to support inverses). By adding a given (Boolean) conjunctive query q as a constraint, we obtain a program that is satisfiable iff $\mathcal{K}$ does not entail q.

In case we are interested only in the classical models of a $\mathbb{GT}$ program P, i.e., if we do not require the stability condition, we can use the automaton A_P^{mods} to decide consistency of P in double exponential time in the size of P. For this we note that the forest-model property of $\mathbb{GT}$ programs described in Proposition C.2 does not refer to stability also holds for the classical models of P. Since the 2-EXPTIME-hardness of consistency testing already applies to positive $\mathbb{GT}$ programs, the resulting 2-EXPTIME upper bound is tight.

Automata Constructions

In the rest of this section we show Proposition C.9, i.e., that the automata A_P^{ce}, $A_P^{\subsetneq}$, and $A_P^{=}$ can be constructed.

Ensuring properness. The automata $A_P^{\subsetneq}$, and $A_P^{=}$ are obtained by first constructing a 2ATA, then transforming it into a 2ATA that is proper, and then transforming it into a 1NTA. For the latter transformation we rely on Theorem 2.23. Before presenting the constructions, we show how to do the first transformation that ensures properness of a given 2ATA.

Lemma C.12. *The following hold:*

- *For every 2ATA A with alphabet Σ_P, there exists some A' that accepts a tree $\mathcal{T}$ iff A accepts $\mathcal{T}$ and $\mathcal{T}$ is proper.*
- *For every 2ATA A with alphabet $\Sigma_P \times \Sigma_P$, there exists some A' that accepts a tree $\mathcal{T}$ iff A accepts $\mathcal{T}$ and $\mathcal{T}$ is proper.*

Furthermore, in both cases, the number of states and the size of the acceptance condition of A' are linearly bounded in the number of states and the size of the acceptance condition of A, respectively.

Proof. We only need to show that there exists an automaton A_P^{prop} that accepts a tree $\mathcal{T}$ over Σ_P if and only if it is proper. The lemma follows easily from this: for the first item, we simply intersect A_P^{prop} with the given A. That is, $A' = A \cap A_P^{\mathit{prop}}$ is the desired automaton. For the second item, we obtain an automaton $(A_P^{\mathit{prop}})^2$ that tests properness of trees over $\Sigma_P \times \Sigma_P$ by taking the standard *product* automaton $(A_P^{\mathit{prop}})^2 = A_P^{\mathit{prop}} \times A_P^{\mathit{prop}}$, and intersecting $(A_P^{\mathit{prop}})^2$ it with A to obtain the desired A'.

Now we define the automaton A_P^{prop} for testing properness of trees over Σ_P. We define:

- L_r is the set of all predicate names of type (T3) in L^P, i.e., all predicates $R_{c,d}$ where c, d are constants in P.
- $L_d = L^P \setminus L_1$, i.e., L_2 contains the predicate names of type (T1) and (T2).

The automaton is then defined as $A_P^{\mathit{prop}} = (\Sigma_P, Q, \delta, q_0, F)$, where

- The set of states Q consists of (i) the initial state q_0; (ii) the state q_r to ensure that predicates from L_r only occur at the root of the tree; (iii) the state q_t to ensure that only term nodes are labeled with predicates in L_d; (iv) the state $q_\emptyset$ to ensure that a node is labeled with $\emptyset$.
- The transition function δ is as follows. In the initial state the automaton checks that the root node has no labels from L_d, and then switches to states q_r, q_t, and $q_\emptyset$ to ensure that the descendants of the root do not have labels from L_r and that the nodes that do not correspond to term nodes are labeled with $\emptyset$. This is implemented using the following transition for each $\sigma \in \Sigma_P$:
$$\delta(\sigma, q_0) = [\sigma \cap L_d = \emptyset] \wedge \bigwedge_{i \in \mathbf{C} \cup \mathbf{F}} (i, q_r) \wedge \bigwedge_{i \in \mathbf{C}} (i, q_t) \wedge \bigwedge_{i \in \mathbf{F}} (i, q_\emptyset).$$
For each $\sigma \in \Sigma_P$, the transitions for q_r, q_t, and $q_\emptyset$ are defined in the following way:
$$\begin{aligned} \delta(\sigma, q_r) &= \begin{cases} \bigwedge_{i \in \mathbf{C} \cup \mathbf{F}} (i, q_r) & \text{if } \sigma \cap L_r = \emptyset \\ \mathbf{false} & \text{if } \sigma \cap L_r \neq \emptyset \end{cases} \\ \delta(\sigma, q_t) &= \bigwedge_{i \in \mathbf{F}} (i, q_t) \wedge \bigwedge_{i \in \mathbf{C}} (i, q_\emptyset) \\ \delta(\sigma, q_\emptyset) &= \begin{cases} \bigwedge_{i \in \mathbf{C} \cup \mathbf{F}} (i, q_\emptyset) & \text{if } \sigma = \emptyset \\ \mathbf{false} & \text{if } \sigma \neq \emptyset \end{cases} \end{aligned}$$
- The (parity) acceptance condition is $F = (\emptyset, Q)$, i.e., all states are allowed to occur infinitely often.

This finishes the construction of the automaton A_P^{prop} that accepts a tree over Σ_P iff it is proper. The last part of the claim can be easily inferred using the fact that A_P^{prop} has a fixed number of states and a parity condition of fixed size. $\square$

Comparing interpretations: the automata $A_P^{\subsetneq}$ and $A_P^{=}$

Now we proceed with the construction of the automata $A_P^{\subsetneq}$ and $A_P^{=}$ that test for strict containment and equality of interpretations. We start by constructing two alternating automata $A_0^{\subsetneq}$ and $A_0^{=}$, and then we transform them into the desired 1NTAs.

- $A_0^{=} = (\Sigma_P \times \Sigma_P, \{q^{=}\}, \delta, q^{=}, F)$ is defined as follows:
  - For each $(N, N') \in \Sigma_P \times \Sigma_P$, the transition is as follows:
  $$\delta((N, N'), q^{=}) = [N = N'] \wedge \bigwedge_{i \in \mathbf{C} \cup \mathbf{F}} (i, q^{=}).$$
  - The acceptance condition is simply $F = (\emptyset, Q)$.
- $A_0^{\subsetneq} = (\Sigma_P \times \Sigma_P, \{q_0, q^{\subseteq}, q^{\neq}\}, \delta, q^{\subseteq}, F)$ is defined similarly, but replacing $[N = N']$ above with $[N \subseteq N']$, and adding additional states $q_0, q^{\neq}$ and defining transitions to make sure that the containment is strict. More precisely, we have:
  - For each $(N, N') \in \Sigma_P \times \Sigma_P$, there are transitions:
  $$\begin{aligned} \delta((N, N'), q_0) &= (0, q^{\subseteq}) \wedge (0, q^{\neq}) \\ \delta((N, N'), q^{\subseteq}) &= [N \subseteq N'] \ \wedge \textstyle\bigwedge_{i \in \mathbf{C} \cup \mathbf{F}} (i, q^{\subseteq}) \\ \delta((N, N'), q^{\neq}) &= [N \neq N'] \vee \textstyle\bigvee_{i \in \mathbf{C} \cup \mathbf{F}} (i, q^{\neq}) \end{aligned}$$
  - The acceptance condition is simply $F = (\emptyset, \{q^{\subseteq}\}, Q)$, that is, the state $q^{\neq}$ is not allowed to occur infinitely often. This ensures that, in some branch of the tree, a node is eventually reached for which N and N' are different.
- The automata $A_P^{\subsetneq}$ and $A_P^{=}$ are obtained by transforming $A_0^{\subsetneq}$ and $A_0^{=}$, respectively, into proper 2ATAs (i.e., intersecting them with $(A_P^{\mathit{prop}})^2$) and then into 1NTAs (in fact, it is not hard to see that 2-wayness and alternation are not really needed in these automata). Both automata $A_P^{\subsetneq}$ and $A_P^{\neq}$ have boundedly many states and a bounded acceptance condition.

Testing the satisfaction of the reduct: the automaton A_P^{ce}

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

1. The automaton $A_P^X = (2^X \times \Sigma_P \times \Sigma_P, Q, \delta, q_0, F)$, which ensures that in a tree $\mathcal{T} = (T, \mathcal{L})$ over $2^X \times \Sigma_P \times \Sigma_P$ every variable is assigned to exactly one node, is defined as follows.

    - The state set Q of A_P^X consists of an initial state q_0 and the states q_x, q'_x, $q_x^{\in}$ and $q_x^{\notin}$ for each variable x of P. Intuitively, the automaton uses q_x to verify that some node is labeled with x, and uses the state q'_x to verify that x is neither in the labeling of the current symbol, nor in the labeling of any descendant. The states $q_x^{\in}$ are $q_x^{\notin}$ to verify the presence or absence of the variable x is in the labeling of the current node, respectively.
    - The transition function δ is as follows. From the initial state the automaton switches to states q_x for each variable $x \in X$, i.e., for each $\sigma \in 2^X \times \Sigma_P \times \Sigma_P$, we have:
    $$\delta(\sigma, q_0) = \bigwedge_{x \in X} (0, q_x).$$
    When in state q_x, the automaton either decides to place the variable in the current node, or chooses a branch where it will be placed. After placing the variable, it enters the state q'_x to ensure that a variable does not occur more than once. This is implemented by the following transition for each $\sigma \in 2^X \times \Sigma_P \times \Sigma_P$ and variable $x \in X$:
    $$\delta(\sigma, q_x) = \Big((0, q_x^{\in}) \wedge \bigwedge_{i \in \mathbf{C} \cup \mathbf{F}} (i, q'_x)\Big) \vee \Big(\bigvee_{i \in \mathbf{C} \cup \mathbf{F}} \big((i, q_x) \wedge \bigwedge_{j \in \mathbf{C} \cup \mathbf{F}, j \neq i} (j, q'_x)\big)\Big),$$
    $$\delta(\sigma, q'_x) = \Big((0, q_x^{\notin}) \wedge \bigwedge_{i \in \mathbf{C} \cup \mathbf{F}} (i, q'_x)\Big).$$
    The transitions for $q_x^{\in}$ and $q_x^{\notin}$ are simple. For each $\sigma = (V, N, N')$ in $2^X \times \Sigma_P \times \Sigma_P$ and variable $x \in X$ we have:
    $$\delta(\sigma, q_x^{\in}) = [x \in V],$$
    $$\delta(\sigma, q_x^{\notin}) = [x \notin V].$$

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

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

   The automaton $A = (2^X \times \Sigma_P \times \Sigma_P, Q, \delta, q_0, F)$ is defined as follows.

   - We define the state set Q first.

$$\begin{aligned} Q \;=\; & \{q^t_W, q^f_W, q^{f'}_W, q^{t,\downarrow}_W, q^{f,\downarrow}_W, q^{f',\downarrow}_W \mid W \text{ is an atom occurring in } P\} \cup \\ & \{q^t_A, q^f_A, , q^{f'}_A \mid A \text{ is a unary predicate name occurring in } P\} \cup \\ & \{q^t_{(R,x)}, q^t_{(R,x,y)}, q^f_{(R,x)}, q^f_{(R,x,y)}, q^{f'}_{(R,x)}, q^{f'}_{(R,x,y)} \mid R(x,y) \text{ is an atom} \\ & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{occurring in } P\} \cup \\ & \{q_x \mid x \text{ is a variable occurring in } P\} \end{aligned}$$

   - Next we define and explain the transition function, and explain also the states in Q.

     - First, the state set Q contains q^t_W, q^f_W and $q^{f'}_W$ for each atom W occurring in P. Intuitively, A moves to q^t_W, q^f_W or $q^{f'}_W$, to verify that under the assigment π the atom W is true in I, false in I, or false in I', respectively. From the initial state q_0, the automaton nondeterministically chooses a rule $r \in P$ and verifies that it is violated, by moving to q^t_W for each positive body atom W, to $q^{f'}_W$ for each negative body atom W, and to q^f_W for each head atom W. Hence, for each $\sigma \in 2^X \times \Sigma_P \times \Sigma_P$, we have:

$$\delta(\sigma, q_0) = \bigvee_{r \in P} \Big(\bigwedge_{W \in \mathit{body}^+(r)} (0, q^t_W) \wedge \bigwedge_{W \in \mathit{body}^-(r)} (0, q^{f'}_W) \wedge \bigwedge_{W \in \mathit{head}(r)} (0, q^f_W) \Big).$$

       It only remains to implement the transitions for q^t_W, q^f_W and $q^{f'}_W$.

     - The transitions for q^t_W use the states $q^{t,\downarrow}_W$ to check that, at the current position in the tree, the atom W is satisfied.
       The transitions from the state q^t_W depend on the form of the atom W. For ground atoms they are simple. Recall that we store binary ground atoms

$R_{c,d}$ in the label of the root, and that unary atoms $A(c)$ are represented by the symbol A in the label of the term node i with $c = a_i$. Hence, to verify the satisfaction of $R(c, d)$ we simply look for the corresponding symbol at the root. If the atom is unary, we use the auxiliary state q^t_A to check that the labeling of the corresponding term node contains A. For non-ground atoms the automaton non-deterministically navigates to some node of the tree. Then it uses the state $q^{t,\downarrow}_W$ to test there the satisfaction of W.
First, depending on the type of W, we let for each $\sigma = (V, N, N')$ in $2^X \times \Sigma_P \times \Sigma_P$:

$$\delta(\sigma, q^t_W) = \begin{cases} (0, q^{t,\downarrow}_W) \vee \bigvee_{i \in \mathbf{C} \cup \mathbf{F}} (i, q^t_W) & \text{if } W \text{ is not ground,} \\ [R_{c,d} \in N] & \text{if } W = R(c, d), \\ (i, q^t_A) & \text{if } W = A(c) \text{ and } c = a_i, \end{cases}$$

and for all $(V, N, N') \in 2^X \times \Sigma_P \times \Sigma_P$ and unary A of P, we let

$$\delta(\sigma, q^t_A) = [A \in N].$$

For the case where W is not ground, we also define transitions from the state $q^{t,\downarrow}_W$, which again depend on the form of the atom W. In case W is unary, for each $\sigma = (V, N, N')$ in $2^X \times \Sigma_P \times \Sigma_P$, we let:

$$\delta(\sigma, q^{t,\downarrow}_W) = \begin{cases} [A \in N \text{ and } x \in V] & \text{if } W = A(x), \\ [x \in V] \wedge (i, q^t_A) & \text{if } W = A(f(x)) \text{ and } f = f_i. \end{cases}$$

If W is binary with a function symbol (i.e., if $W = R(x, f(x))$ or $W = R(f(x), x)$), we define, for each $\sigma = (V, N, N')$ in $2^X \times \Sigma_P \times \Sigma_P$:

$$\delta(\sigma, q^{t,\downarrow}_W) = \begin{cases} [R_f \in N \text{ and } x \in V] & \text{if } W = R(x, f(x)) \\ [R^-_f \in N \text{ and } x \in V] & \text{if } W = R(f(x), x). \end{cases}$$

For atoms $R(x, y)$ it is a bit more complicated. For all $(V, N, N') \in 2^X \times \Sigma_P \times \Sigma_P$ and $W = R(x, y)$, we have:

$$\begin{aligned} \delta(\sigma, q^{t,\downarrow}_W) \;=\; & (0, q^t_{(R,x,y)}) \vee \\ & \Big([x \in V] \wedge \big(\bigvee_{i \in \mathbf{F}} ([R_{f_i} \in N] \wedge (i, q_y))\big)\Big) \vee \\ & \Big([x \in V] \wedge (-1, q_y) \wedge (-1, q^t_{(R,x)})\Big) \end{aligned}$$

Intuitively, the three disjuncts verify the three possible ways in which an atom $R(x, y)$ can be satisfied: (i) x and y are assigned to constants, (ii) y is mapped to a functional successor of $\pi(x)$, and (iii) x is mapped to a functional successor of $\pi(y)$.

In the first disjunct, the automaton moves to the auxiliary state $q^t_{(R,x,y)}$ to verify whether there is a pair of constants witnessing the satisfaction of the atom $R(x, y)$, i.e., whether there is a pair c, d such that x is assigned to c, y is assigned to d, and $R(c, d)$ holds; recall that the latter is stored at the label of the root. Hence we have, for each $\sigma = (V, N, N')$ in $2^X \times \Sigma_P \times \Sigma_P$:

$$\delta(\sigma, q^t_{(R,x,y)}) \;=\; \bigvee_{\{i,j\}\subseteq \mathbf{C}} \big([R_{a_i,a_j} \in N] \wedge (i, q_x) \wedge (j, q_y)\big)$$

Finally, for the auxiliary states q_x and $q^t_{(R,x)}$ we have, for each $\sigma = (V, N, N')$ in $2^X \times \Sigma_P \times \Sigma_P$

$$\begin{aligned} \delta(\sigma, q_x) &= [x \in V], \qquad\qquad \text{and} \\ \delta(\sigma, q^t_{(R,x)}) &= \textstyle\bigvee_{i\in\mathbf{F}} [R^-_{f_i} \notin N] \wedge (i, q_x). \end{aligned}$$

- The transitions for q^f_R are analogous, but each test $[s \in N]$ for a symbol $s \in L_P$, is replaced by the test $[s \notin N]$, and we use the states superindexed with f instead of their t counterparts (q^f_W instead of q^t_W, $q^{f,\downarrow}_W$ instead of $q^{t,\downarrow}_W$, etc.).
- Similarly, in the transitions for $q^{f'}_R$ we test for $[s \notin N']$ and use the states superindexed with f'.

• In the acceptance condition, we only need to prohibit the states q^t_W, q^f_W and $q^{f'}_W$, which can postpone the tests for the truth or falsity of atoms, from occurring infinitely often. Hence we set

$$F = (\{q^t_W, q^f_W, q^{f'}_W \mid W \text{ is an atom in } P\}, Q).$$

3. We can now finalize the construction of A^{ce}_P. First we let $B = (2^X \times \Sigma_P \times \Sigma_P, Q, \delta, q_0, F)$ be the result of translating the intersection automaton $A \cap A^X_P$ into a 1NTA. The state set of B is exponential in P, and its parity index is fixed.

   To obtain A^{ce}_P, we first obtain B' by projecting away the variable assignment in the first component of the labels, in a similar way as we projected the second component from A in Definition C.7. That is, $B' = (\Sigma_P \times \Sigma_P, Q, \delta', q_0, F)$ where for each $(N, N') \in \Sigma^{P^2}$ and each state $q \in Q$,

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

   The automaton B' accepts a tree $\mathcal{T}$ over $\Sigma_P \times \Sigma_P$ iff $\mathcal{T}$ can be decorated with variables in a way that the resulting tree $\mathcal{T}'$ over $2^X \times \Sigma_P \times \Sigma_P$ is accepted by B. Finally, the automaton A^{ce}_P is obtained by transforming B' into a proper automata, by intersecting it with the 1NTA version of $(A^{prop}_P)^2$. The automaton A^{ce}_P accepts exactly the pairs (I, I') such that $I \not\models P^{I'}$. These states have only a linear impact in the size of B, hence the state set of A_P^{ce} remains exponential and the parity index fixed.

Bibliography

[AB01] Güray Alsaç and Chitta Baral. Reasoning in description logics using declarative logic programming. Tech. rep., Dep. Computer Science and Engineering, Arizona State University, 2001.

[ADG$^+$05] Grigoris Antoniou, Carlos Viegas Damasio, Benjamin Grosof, Ian Horrocks, Michael Kifer, Jan Maluszynski, and Peter F. Patel-Schneider. Combining rules and ontologies. A survey. Technical Report Deliverable I3-D3, REWERSE Project, February 2005.

[AN78] Hajnal Andréka and István Németi. The generalised completeness of Horn predicate logics as programming language. *Acta Cybernetica*, 4(1):3–10, 1978.

[ANvB98] Hajnal Andréka, István Németi, and Johan van Benthem. Modal languages and bounded fragments of predicate logic. *Journal of Philosophical Logic*, 27(3):217–274, 1998.

[Bar02] Chitta Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2002.

[BBC09] Sabrina Baselice, Piero A. Bonatti, and Giovanni Criscuolo. On finitely recursive programs. *Theory and Practice of Logic Programming*, 9(2):213–238, 2009.

[BBL05] Franz Baader, Sebastian Brandt, and Carsten Lutz. Pushing the EL envelope. In Leslie Pack Kaelbling and Alessandro Saffiotti, editors, *IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK, July 30-August 5, 2005*, pages 364–369. Professional Book Center, 2005.

[BCM$^+$03] Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press, 2003.

[BdRV01] Patrick Blackburn, Maarten de Rijke, and Yde Venema. *Modal Logic*, volume 53 of *Cambridge Tracts in Theoretical Computer Sc.* Cambridge University Press, Cambridge, 2001.

[BLMV08] Piero Bonatti, Carsten Lutz, Aniello Murano, and Moshe Y. Vardi. The complexity of enriched μ-calculi. *Logical Methods in Computer Science*, 4(3:11):1–27, 2008.

[Bon04] Piero A. Bonatti. Reasoning with infinite stable models. *Artificial Intelligence*, 156(1):75–111, 2004.

[Bre91] Gerhard Brewka. *Nonmonotonic reasoning: logical foundations of common sense*. Cambridge University Press, New York, NY, USA, 1991.

[Büc60] J. Richard Büchi. Weak second-order arithmetic and finite automata. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 6(1-6):66–92, 1960.

[CCIL08a] Francesco Calimeri, Susanna Cozza, Giovambattista Ianni, and Nicola Leone. Computable functions in ASP: Theory and implementation. In M.G. de La Banda and E. Pontelli, editors, *Proceedings 24th International Conference on Logic Programming (ICLP 2008)*, number 5366 in LNCS, pages 407–424. Springer, 2008.

[CCIL08b] Francesco Calimeri, Susanna Cozza, Giovambattista Ianni, and Nicola Leone. DLV-Complex homepage, (since 2008). `http://www.mat.unical.it/dlv-complex`.

[CD97] Marco Cadoli and Francesco M. Donini. A survey on knowledge compilation. *AI Communications*, 10(3-4):137–150, 1997.

[CDG03] Diego Calvanese and Giuseppe De Giacomo. Expressive description logics. In Baader et al. [BCM$^+$03], chapter 5, pages 178–218.

[CEO07] Diego Calvanese, Thomas Eiter, and Magdalena Ortiz. Answering regular path queries in expressive description logics: An automata-theoretic approach. In *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence, July 22-26, 2007, Vancouver, British Columbia, Canada*, pages 391–396. AAAI Press, 2007.

[CEO09] Diego Calvanese, Thomas Eiter, and Magdalena Ortiz. Regular path queries in expressive description logics with nominals. In Craig Boutilier, editor, *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*, pages 714–720, 2009.

[CGK08] Andrea Calì, Georg Gottlob, and Michael Kifer. Taming the infinite chase: Query answering under expressive relational constraints. In Gerhard Brewka and Jérôme Lang, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Eleventh International Conference, KR 2008, Sydney, Australia, September 16-19, 2008*, pages 70–80. AAAI Press, 2008.

[CGL09] Andrea Calì, Georg Gottlob, and Thomas Lukasiewicz. Datalog$\pm$: a unified approach to ontologies and integrity constraints. In *ICDT '09: Proceedings of the 12th International Conference on Database Theory*, pages 14–30, New York, NY, USA, 2009. ACM.

[CHM$^+$08] Bernardo Cuenca Grau, Ian Horrocks, Boris Motik, Bijan Parsia, Peter Patel-Schneider, and Ulrike Sattler. OWL 2: The next step for OWL. *J. of Web Semantics*, 6(4):309–322, November 2008.

[Cho95] Jan Chomicki. Depth-bounded bottom-up evaluation of logic programs. *Journal of Logic Programming*, 25(1):1–31, 1995.

[CI93] Jan Chomicki and Tomasz Imielinski. Finite representation of infinite query answers. *ACM Transactions on Database Systems*, 18(2):181–223, 1993.

[CKS81] Ashok K. Chandra, Dexter C. Kozen, and Larry J. Stockmeyer. Alternation. *J. ACM*, 28(1):114–133, 1981.

[dBEPT06] Jos de Bruijn, Thomas Eiter, Axel Polleres, and Hans Tompits. On representational issues about combinations of classical theories with nonmonotonic rules. In Jérôme Lang, Fangzhen Lin, and Ju Wang, editors, *Knowledge Science, Engineering and Management, First International Conference, KSEM 2006, Guilin, China, August 5-8, 2006, Proceedings*, volume 4092 of *Lecture Notes in Computer Science*, pages 1–22. Springer, 2006.

[DEGV01] Evgeny Dantsin, Thomas Eiter, Georg Gottlob, and Andrei Voronkov. Complexity and expressive power of logic programming. *ACM Computing Surveys*, 33(3):374–425, 2001.

[DM02] Adnan Darwiche and Pierre Marquis. A knowledge compilation map. *Journal of Artificial Intelligence Research*, 17:229–264, 2002.

[DNK97] Yannis Dimopoulos, Bernhard Nebel, and Jana Koehler. Encoding planning problems in nonmonotonic logic programs. In *Proc. European Conference on Planning 1997 (ECP-97)*, volume 1348 of *Lecture Notes in Computer Science*, pages 169–181. Springer, 1997.

[EFL$^+$03] Thomas Eiter, Wolfgang Faber, Nicola Leone, Gerald Pfeifer, and Axel Polleres. A logic programming approach to knowledge-state planning, II: The $DLV^{\mathcal{K}}$ system. *Artificial Intelligence*, 144(1-2):157–211, 2003.

[EFL$^+$04] Thomas Eiter, Wolfgang Faber, Nicola Leone, Gerald Pfeifer, and Axel Polleres. A logic programming approach to knowledge-state planning: Semantics and complexity. *ACM Transactions on Computational Logic*, 5(2):206–263, 2004.

[EG97] Thomas Eiter and Georg Gottlob. Expressiveness of stable model semantics for disjunctive logic programs with functions. *Journal of Logic Programming*, 33(2):167–178, 1997.

[EGM97] Thomas Eiter, Georg Gottlob, and Heikki Mannila. Disjunctive datalog. *ACM Transactions on Database Systems*, 22(3):364–418, 1997.

[EGOŠ08] Thomas Eiter, Georg Gottlob, Magdalena Ortiz, and Mantas Šimkus. Query answering in the description logic Horn-SHIQ. In Steffen Hölldobler, Carsten Lutz, and Heinrich Wansing, editors, *Logics in Artificial Intelligence, 11th European Conference, JELIA 2008, Dresden, Germany, September 28 - October 1, 2008. Proceedings*, volume 5293 of *Lecture Notes in Computer Science*, pages 166–179. Springer, 2008.

[EIK09] Thomas Eiter, Giovambattista Ianni, and Thomas Krennwallner. Answer set programming: A primer. In Sergio Tessaris, Enrico Franconi, Thomas Eiter, Claudio Gutierrez, Siegfried Handschuh, Marie-Christine Rousset, and Renate A. Schmidt, editors, *Reasoning Web*, volume 5689 of *Lecture Notes in Computer Science*, pages 40–110. Springer, 2009.

[EIP$^+$06] Thomas Eiter, Giovambattista Ianni, Axel Florian Polleres, Roman Schindlauer, and Hans Tompits. Reasoning with Rules and Ontologies. In Pedro Barahona, Francois Bry, Enrico Franconi, Nicola Henze, and Ulrike Sattler, editors, *Lecture Notes in Computer Science. Reasoning Web*, pages 93–127, 4126, 2006. Lecture Notes in Computer Science. Springer.

[EIST05] Thomas Eiter, Giovambattista Ianni, Roman Schindlauer, and Hans Tompits. A uniform integration of higher-order reasoning and external evaluations in answer set programming. In Leslie Pack Kaelbling and Alessandro Saffiotti, editors, *IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK, July 30-August 5, 2005*, pages 90–96. Professional Book Center, 2005.

[Eit07] Thomas Eiter. Answer set programming for the semantic web (tutorial). In Ilkka Niemelä and Veronika Dahl, editors, *Proceedings 23th International*

Conference on Logic Programming (ICLP 2007), number 4670 in Lecture Notes in Computer Science, pages 23–26. Springer, 2007.

[EJ88] E. Allen Emerson and Charanjit S. Jutla. The complexity of tree automata and logics of programs (extended abstract). In *Proceedings of the 29th Annual Symposium on Foundations of Computer Science, 24-26 October 1988, White Plains, New York, USA*, pages 328–337. IEEE, 1988.

[EJ91] E. Allen Emerson and Charanjit S. Jutla. Tree automata, mu-calculus and determinacy. In *Proceedings of the 32nd Annual Symposium on the Foundations of Computer Science (FOCS'91)*, pages 368–377, 1991.

[ELM$^+$97] Thomas Eiter, Nicola Leone, Cristinel Mateis, Gerald Pfeifer, and Francesco Scarcello. A deductive system for non-monotonic reasoning. In Jürgen Dix, Ulrich Furbach, and Anil Nerode, editors, *Proc. 4th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'97)*, volume 1265 of *Lecture Notes in Computer Science*, pages 364–375. Springer, 1997.

[ELOŠ09] Thomas Eiter, Carsten Lutz, Magdalena Ortiz, and Mantas Šimkus. Query answering in description logics: The knots approach. In Hiroakira Ono, Makoto Kanazawa, and Ruy J. G. B. de Queiroz, editors, *Logic, Language, Information and Computation, 16th International Workshop, WoLLIC 2009, Tokyo, Japan, June 21-24, 2009. Proceedings*, volume 5514 of *Lecture Notes in Computer Science*, pages 26–36. Springer, 2009.

[EOŠ08] Thomas Eiter, Magdalena Ortiz, and Mantas Šimkus. Reasoning using knots. In Iliano Cervesato, Helmut Veith, and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 15th International Conference, LPAR 2008, Doha, Qatar, November 22-27, 2008. Proceedings*, volume 5330 of *Lecture Notes in Computer Science*, pages 377–390. Springer, 2008.

[EŠ09] Thomas Eiter and Mantas Šimkus. Bidirectional answer set programs with function symbols. In Craig Boutilier, editor, *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*, pages 765–771, 2009.

[EŠ10] Thomas Eiter and Mantas Šimkus. FDNC: Decidable nonmonotonic disjunctive logic programs with function symbols. *ACM Trans. Comput. Logic*, 11(2):1–50, 2010.

[Fag94] François Fages. Consistency of clark's completion and existence of stable models. *Methods of Logic in Computer Science*, (1):51–60, 1994.

[Fit96] Melvin Fitting. *First-order logic and automated theorem proving (2nd ed.)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996.

[GK04a] Georg Gottlob and Christoph Koch. Logic-based web information extraction. *SIGMOD Rec.*, 33(2):87–94, 2004.

[GK04b] Georg Gottlob and Christoph Koch. Monadic datalog and the expressive power of languages for web information extraction. *J. ACM*, 51(1):74–113, 2004.

[GKK$^+$08] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, Torsten Schaub, and Sven Thiele. Engineering an incremental asp solver. In Maria Garcia de la Banda and Enrico Pontelli, editors, *Logic Programming, 24th International Conference, ICLP 2008, Udine, Italy, December 9-13 2008, Proceedings*, volume 5366 of *Lecture Notes in Computer Science*, pages 190–205. Springer, 2008.

[GKL$^+$07] Erich Grädel, Phokion G. Kolaitis, Leonid Libkin, Maarten Marx, Joel Spencer, Moshe Y. Vardi, Yde Venema, and Scott Weinstein. *Finite Model Theory and Its Applications (Texts in Theoretical Computer Science. An EATCS Series)*. Springer, June 2007.

[GKNS07] Martin Gebser, Benjamin Kaufmann, André Neumann, and Torsten Schaub. *Clasp* : A conflict-driven answer set solver. In *Logic Programming and Nonmonotonic Reasoning, 9th International Conference, LPNMR 2007, Tempe, AZ, USA, May 15-17, 2007, Proceedings*, volume 4483 of *Lecture Notes in Computer Science*, pages 260–265. Springer, 2007.

[GKV97] Erich Grädel, Phokion G. Kolaitis, and Moshe Y. Vardi. On the decision problem for two-variable first-order logic. *Bulletin of Symbolic Logic*, 3(1):53–69, 1997.

[GL91] Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9(3/4):365–386, 1991.

[GL92] Michael Gelfond and Vladimir Lifschitz. Representing actions in extended logic programming. In *Proc. Joint International Conference and Symposium on Logic Programming (JICSLP-92)*, pages 559–573. MIT Press, 1992.

[GL98] Enrico Giunchiglia and Vladimir Lifschitz. An action language based on causal explanation: Preliminary report. In *Proc. 15th National Conference on Artificial Intelligence (AAAI-98)*, pages 623–630. AAAI Press, 1998.

[GLHS08] Birte Glimm, Carsten Lutz, Ian Horrocks, and Ulrike Sattler. Conjunctive query answering for the description logic shiq. *J. Artif. Intell. Res. (JAIR)*, 31:157–204, 2008.

[GP03] Georg Gottlob and Christos H. Papadimitriou. On the complexity of single-rule datalog queries. *Inf. Comput.*, 183(1):104–122, 2003.

[Grä99] Erich Grädel. On the restraining power of guards. *J. Symb. Log.*, 64(4):1719–1742, 1999.

[GRS91] Allen Van Gelder, Kenneth A. Ross, and John S. Schlipf. The well-founded semantics for general logic programs. *J. ACM*, 38(3):620–650, 1991.

[GST07] Martin Gebser, Torsten Schaub, and Sven Thiele. GrinGo : A new grounder for answer set programming. In *Logic Programming and Nonmonotonic Reasoning, 9th International Conference, LPNMR 2007, Tempe, AZ, USA, May 15-17, 2007, Proceedings*, volume 4483 of *Lecture Notes in Computer Science*, pages 266–271. Springer, 2007.

[Her71] Jacques Herbrand. *Logical Writings*. Harvard University Press, 1971. Edited by Warren D. Goldfarb.

[Hey06] Stijn Heymans. *Decidable Open Answer Set Programming*. PhD thesis, Theoretical Computer Science Lab, Department of Computer Science, Vrije Universiteit Brussel, 2006.

[HJ99] Patrik Haslum and Peter Jonsson. Some results on the complexity of planning with incomplete information. In Susanne Biundo and Maria Fox, editors, *Proc. 5th European Conference on Planning (ECP-99)*, volume 1809 of *Lecture Notes in Computer Science*, pages 308–318. Springer, 1999.

[HM87] Steve Hanks and Drew V. McDermott. Nonmonotonic logic and temporal projection. *Artificial Intelligence*, 33(3):379–412, 1987.

[HM92] Joseph Y. Halpern and Yoram Moses. A guide to completeness and complexity for modal logics of knowledge and belief. *Artif. Intell.*, 54(3):319–379, 1992.

[HMS04] Ullrich Hustadt, Boris Motik, and Ulrike Sattler. Reducing SHIQ-description logic to disjunctive datalog programs. In *Proceedings KR-2004*, pages 152–162. AAAI Press, 2004.

[HNV05] Stijn Heymans, Davy Van Nieuwenborgh, and Dirk Vermeir. Nonmonotonic ontological and rule-based reasoning with extended conceptual logic programs. In Asunción Gómez-Pérez and Jérôme Euzenat, editors, *Proc. 2nd European Semantic Web Conference (ESWC-05)*, volume 3532 of *Lecture Notes in Computer Science*, pages 392–407. Springer, 2005.

[HSG04] Ullrich Hustadt, Renate A. Schmidt, and Lilia Georgieva. A survey of decidable first-order fragments and description logics. *Journal of Relational Methods in Computer Science*, 1:251–276, 2004.

[HV03] Stijn Heymans and Dirk Vermeir. Integrating semantic web reasoning and answer set programming. In Marina de Vos and Alessandro Provetti, editors, *Proc. Workshop on Answer Set Programming (ASP-2003)*, volume 78 of *CEUR Workshop Proc.*, pages 194–208. CEUR-WS.org, 2003.

[Imm88] Neil Immerman. Nondeterministic space is closed under complementation. *SIAM J. Comput.*, 17(5):935–938, 1988.

[Jon75] Neil D. Jones. Space-bounded reducibility among combinatorial problems. *J. Comput. Syst. Sci.*, 11(1):68–85, 1975.

[Kaz08] Yevgeny Kazakov. Riq and sroiq are harder than shoiq. In Gerhard Brewka and Jérôme Lang, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Eleventh International Conference, KR 2008, Sydney, Australia, September 16-19, 2008*, pages 274–284. AAAI Press, 2008.

[KPV01] Orna Kupferman, Nir Piterman, and Moshe Y. Vardi. Extended temporal logic revisited. In Kim Guldstrand Larsen and Mogens Nielsen, editors, *CONCUR 2001 - Concurrency Theory, 12th International Conference, Aalborg, Denmark, August 20-25, 2001, Proceedings*, volume 2154 of *Lecture Notes in Computer Science*, pages 519–535. Springer, 2001.

[KSV02] Orna Kupferman, Ulrike Sattler, and Moshe Y. Vardi. The complexity of the graded μ-calculus. In Andrei Voronkov, editor, *Proc. of the 18th Int. Conf. on Automated Deduction (CADE 2002)*, volume 2392 of *Lecture Notes in Computer Science*, pages 423–437. Springer, 2002.

[KV98] Orna Kupferman and Moshe Y. Vardi. Weak alternating automata and tree automata emptiness. In *Proceedings of the Thirtieth Annual ACM Symposium on the Theory of Computing*, pages 224–233. ACM, 1998.

[Lif99] Vladimir Lifschitz. Answer set planning. In Danny De Schreye, editor, *Proc. 16th International Conference on Logic Programming (ICLP-99)*, pages 23–37. The MIT Press, 1999.

[Lif02] Vladimir Lifschitz. Answer set programming and plan generation. *Artificial Intelligence*, 138:39–54, 2002.

[LL09] Yuliya Lierler and Vladimir Lifschitz. One more decidable class of finitely ground programs. In Patricia M. Hill and David Scott Warren, editors, *Logic Programming, 25th International Conference, ICLP 2009, Pasadena, CA, USA, July 14-17, 2009. Proceedings*, volume 5649 of *Lecture Notes in Computer Science*, pages 489–493. Springer, 2009.

[LPR98] Hector J. Levesque, Fiora Pirri, and Raymond Reiter. Foundations for the situation calculus. *Electronic Transactions on Artificial Intelligence*, 2:159–178, 1998.

[LRS97] Nicola Leone, Pasquale Rullo, and Francesco Scarcello. Disjunctive stable models: Unfounded sets, fixpoint semantics, and computation. *Inf. Comput.*, 135(2):69–112, 1997.

[LT94] Vladimir Lifschitz and Hudson Turner. Splitting a logic program. In Pascal Van Hentenryck, editor, *Proc. 11th International Conference on Logic Programming (ICLP-94)*, pages 23–37. The MIT Press, 1994.

[Lut07] Carsten Lutz. Inverse roles make conjunctive queries hard. In Diego Calvanese, Enrico Franconi, Volker Haarslev, Domenico Lembo, Boris Motik, Anni-Yasmin Turhan, and Sergio Tessaris, editors, *Proceedings of the 2007 International Workshop on Description Logics (DL2007), Brixen-Bressanone, near Bozen-Bolzano, Italy, 8-10 June, 2007*, volume 250 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2007.

[LWZ08] Carsten Lutz, Frank Wolter, and Michael Zakharyaschev. Temporal description logics: A survey. In Stéphane Demri and Christian S. Jensen, editors, *15th International Symposium on Temporal Representation and Reasoning, TIME 2008, Université du Québec à Monteéal, Canada, 16-18 June 2008*, pages 3–14. IEEE Computer Society, 2008.

[MHS07] Boris Motik, Ian Horrocks, and Ulrike Sattler. Bridging the gap between OWL and relational databases. In Carey L. Williamson, Mary Ellen Zurko, Peter F. Patel-Schneider, and Prashant J. Shenoy, editors, *Proc. 16th International Conference on World Wide Web (WWW-07)*, pages 807–816. ACM, 2007.

[Min88] Jack Minker, editor. *Foundations of Deductive Databases and Logic Programming*. Morgan Kaufmann, 1988.

[MNR92] V. Wiktor Marek, Anil Nerode, and Jeffrey B. Remmel. How complicated is the set of stable models of a recursive logic program? *Ann. Pure Appl. Logic*, 56(1-3):119–135, 1992.

[MNR94] V. Wiktor Marek, Anil Nerode, and Jeffrey B. Remmel. The stable models of a predicate logic program. *J. Log. Program.*, 21(3):129–153, 1994.

[MNR99] V. Wiktor Marek, Anil Nerode, and Jeffrey B. Remmel. Logic programs, well-orderings, and forward chaining. *Ann. Pure Appl. Logic*, 96(1-3):231–276, 1999.

[Mot06] Boris Motik. *Reasoning in Description Logics using Resolution and Deductive Databases*. PhD thesis, Univesität Karlsruhe (TH), Karlsruhe, Germany, January 2006.

[MR03] V. Wiktor Marek and Jeffrey B. Remmel. On the expressibility of stable logic programming. *Theory and Practice of Logic Programming*, 3(4-5):551–567, 2003.

[MS95] David E. Muller and Paul E. Schupp. Simulating alternating tree automata by nondeterministic automata: New results and new proofs of the theorems of rabin, mcnaughton and safra. *Theor. Comput. Sci.*, 141(1&2):69–107, 1995.

[MT99] Victor W. Marek and Mirosław 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.

[MTS07] A. Ricardo Morales, Phan Huy Tu, and Tran Cao Son. An extension to conformant planning using logic programming. In Manuela M. Veloso, editor, *Proc. 20th International Joint Conference on Artificial Intelligence (IJCAI-07)*, pages 1991–1996. AAAI Press/IJCAI, 2007.

[MV07] Maarten Marx and Yde Venema. Local variations on a loose theme: Modal logic and decidability. In *Finite Model Theory and Its Applications*, chapter 7, pages 371–429. Springer, June 2007.

[MvH04] Deborah L. Mcguinness and Frank van Harmelen. OWL web ontology language overview. W3C recommendation, W3C, February 2004.

[Ném86] István Németi. Free algebras and decidability in algebraic logic. DSc. thesis, Mathematical Institute of The Hungarian Academy of Sciences, Budapest, 1986.

[Nie99] Ilkka Niemelä. Logic programming with stable model semantics as constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3–4):241–273, 1999.

[NS97] Ilkka Niemelä and Patrik Simons. Smodels - an implementation of the stable model and well-founded semantics for normal lp. In Jürgen Dix, Ulrich Furbach, and Anil Nerode, editors, *Proc. 4th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'97)*, volume 1265 of *Lecture Notes in Computer Science*, pages 421–430. Springer, 1997.

[OŠE08a] Magdalena Ortiz, Mantas Šimkus, and Thomas Eiter. Conjunctive query answering in SH using knots. In Franz Baader, Carsten Lutz, and Boris Motik, editors, *Description Logics*, volume 353 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2008.

[OŠE08b] Magdalena Ortiz, Mantas Šimkus, and Thomas Eiter. Worst-case optimal conjunctive query answering for an expressive description logic without inverses. In Dieter Fox and Carla P. Gomes, editors, *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008, Chicago, Illinois, USA, July 13-17, 2008*, pages 504–510. AAAI Press, 2008.

[Pap94] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[PFT$^+$04] Jeff Z. Pan, Enrico Franconi, Sergio Tessaris, Giorgos Stamou, Vassilis Tzouvaras, Luciano Serafini, Ian Horrocks, and Birte Glimm. Specification of coordination of rule and ontology languages. Technical report, The Knowledge Web project, 2004.

[Pra79] Vaughan R. Pratt. Models of program logics. In *20th Annual Symposium on Foundations of Computer Science, 29-31 October 1979, San Juan, Puerto Rico*, pages 115–122. IEEE, 1979.

[PSHH04] Peter Patel-Schneider, Patrick Hayes, and Ian Horrocks. OWL Web Ontology Language semantics and abstract syntax – W3C recommendation. Technical report, World Wide Web Consortium, February 2004. Available at `http://www.w3.org/TR/owl-semantics/`.

[PSV06] Guoqiang Pan, Ulrike Sattler, and Moshe Y. Vardi. Bdd-based decision procedures for the modal logic K. *Journal of Applied Non-Classical Logics*, 16(1-2):169–208, 2006.

[Rab69] Michael O. Rabin. Decidability of second-order theories and automata on infinite trees. *Transactions of the American Mathematical Society*, 141:1–35, 1969.

[Rin04] Jussi Rintanen. Complexity of planning with partial observability. In Shlomo Zilberstein, Jana Koehler, and Sven Koenig, editors, *Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling (ICAPS 2004), June 3-7 2004, Whistler, British Columbia, Canada*, pages 345–354. AAAI, 2004.

[Ros06] Riccardo Rosati. Integrating ontologies and rules: Semantic and computational issues. In Pedro Barahona, François Bry, Enrico Franconi, Nicola Henze, and Ulrike Sattler, editors, *Reasoning Web*, volume 4126 of *Lecture Notes in Computer Science*, pages 128–151. Springer, 2006.

[Sav70] Walter J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *Journal of Computer and System Sciences*, 4(2):177–192, 1970. ISSN 1439-2275.

[SBTM06] Tran Cao Son, Chitta Baral, Nam Tran, and Sheila A. McIlraith. Domain-dependent knowledge in answer set planning. *ACM Transactions on Computational Logic*, 7(4):613–657, 2006.

[Sch91] Klaus Schild. A correspondence theory for terminological logics: Preliminary report. In *Proc. 12th International Joint Conference on Artificial Intelligence (IJCAI-91)*, pages 466–471. Morgan Kaufmann, 1991.

[ŠE07] Mantas Šimkus and Thomas Eiter. FDNC: Decidable non-monotonic disjunctive logic programs with function symbols. In Nachum Dershowitz and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 14th International Conference, LPAR 2007, Yerevan, Armenia, October 15-19, 2007, Proceedings*, volume 4790 of *Lecture Notes in Computer Science*, pages 514–530. Springer, 2007.

[Ser06] Olivier Serre. Parity games played on transition graphs of one-counter processes. In Luca Aceto and Anna Ingólfsdóttir, editors, *Foundations of Software Science and Computation Structures, 9th International Conference, FOSSACS 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 25-31, 2006, Proceedings*, volume 3921 of *Lecture Notes in Computer Science*, pages 337–351. Springer, 2006.

[SNS02] Patrik Simons, Ilkka Niemelä, and Timo Soininen. Extending and implementing the stable model semantics. *Artif. Intell.*, 138(1-2):181–234, 2002.

[STGM05] Tran Cao Son, Phan Huy Tu, Michael Gelfond, and A. Ricardo Morales. Conformant planning for domains with constraints-a new approach. In Manuela M. Veloso and Subbarao Kambhampati, editors, *Proc. 20th National Conference on Artificial Intelligence (AAAI-05)*, pages 1211–1216. AAAI Press/MIT Press, 2005.

[Swi04] Terrance Swift. Deduction in ontologies via ASP. In Vladimir Lifschitz and Ilkka Niemelä, editors, *Proc. 7th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR-04)*, volume 2923 of *LNCS/LNAI*, pages 275–288. Springer, 2004.

[Syr01] Tommi Syrjänen. Omega-restricted logic programs. In Thomas Eiter, Wolfgang Faber, and Miroslaw Truszczynski, editors, *Proc. 6th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR-01)*, volume 2173 of *Lecture Notes in Computer Science*, pages 267–279. Springer, 2001.

[SZ95] V. S. Subrahmanian and Carlo Zaniolo. Relating stable models and AI planning domains. In *Proc. ICLP-95*, pages 233–247. MIT Press, 1995.

[Sze88] Róbert Szelepcsényi. The method of forced enumeration for nondeterministic automata. *Acta Inf.*, 26(3):279–284, 1988.

[Tho90] Wolfgang Thomas. Automata on infinite objects. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 133–192. Elsevier, 1990.

[TSB07] Phan Huy Tu, Tran Cao Son, and Chitta Baral. Reasoning and planning with sensing actions, incomplete information, and static causal laws using answer set programming. *Theory and Practice of Logic Programming*, 7(4):377–450, 2007.

[Var96] Moshe Y. Vardi. Why is modal logic so robustly decidable? In Neil Immerman and Phokion G. Kolaitis, editors, *Descriptive Complexity and Finite Models*, volume 31 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 149–184. American Mathematical Society, 1996.

[Var98] Moshe Y. Vardi. Reasoning about the past with two-way automata. In Kim Guldstrand Larsen, Sven Skyum, and Glynn Winskel, editors, *Automata, Languages and Programming, 25th International Colloquium, ICALP'98, Aalborg, Denmark, July 13-17, 1998, Proceedings*, volume 1443 of *Lecture Notes in Computer Science*, pages 628–641. Springer, 1998.

[VW86] Moshe Y. Vardi and Pierre Wolper. Automata-theoretic techniques for modal logics of programs. *Journal of Computer and System Sciences*, 32:183–221, 1986.

[VW94] Moshe Y. Vardi and Pierre Wolper. Reasoning about infinite computations. *Inf. Comput.*, 115(1):1–37, 1994.

[Wol05] Stefan Woltran. Answer set programming: Model applications and proofs-of-concept. Technical Report WP5, Working Group on Answer Set Programming (WASP, IST-FET-2001-37004), 2005. Available at `www.kr.tuwien.ac.at/research/projects/WASP/report.html`.