FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Answer Set Programming with External Sources:

Algorithms and Efficient Evaluation

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktor der technischen Wissenschaften

by

Dipl.-Ing. Dipl.-Ing. Christoph Redl
Registration Number 0525250

to the Faculty of Informatics
at the Vienna University of Technology

Advisors: O. Univ.-Prof. Dipl.-Ing. Dr. techn. Thomas Eiter
Associate Prof. Dipl.-Ing. Dr. techn. Stefan Woltran

The dissertation has been reviewed by:

(O. Univ.-Prof. Dipl.-Ing. Dr. techn. Thomas Eiter)

(Prof. Dr. Giovambattista Ianni)

Vienna, 24th of April, 2014

(Dipl.-Ing. Dipl.-Ing. Christoph Redl)

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.ac.at

Erklärung zur Verfassung der Arbeit

DI DI Christoph Redl
Kieslingstraße 9
3500 Krems

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 24.04.2014

(Christoph Redl)

Dedicated to my parents,
Anita and Karl.

Gewidmet meinen Eltern,
Anita und Karl.

Acknowledgements

First of all I thank my PhD supervisor Prof. Thomas Eiter for encouraging me to do a PhD already while I worked on my master's theses under his supervision. Finishing this PhD thesis within the planned time frame would not have been possible without the continuous support and advices he gave me in the last years. Although he is a top researcher with a very dense schedule he always finds time for his students. I appreciated that he gave me much freedom concerning timing and setting focuses, but also ensured by regular meetings a good progress of my work. My second supervisor Prof. Stefan Woltran not only gave important feedback while I was writing my PhD proposal and my thesis, but also proved to be very patient by answering my pressing organizational questions about our doctoral program, even before it was officially started.

Next, I want to thank Michael Fink who gave some excellent courses which I enjoyed during my master studies. They attracted my interest and supported my decision for doing my theses in the area of answer set programming. As a co-author of most papers in context of this thesis and our project, he made many valuable suggestions for improvements and helped to present the results in a convincing way. The work on my PhD would have been much harder without the continuous help of Thomas Krennwallner, an expert in almost everything. He always has an open ear for questions of any kind, including publication and programming issues and technical problems with our benchmark systems. Peter Schüller is one of my predecessors in the DLVHEX project. He was a great help at the beginning of my PhD project by supporting me when I needed to get into the sourcecode of the system. Moreover, the good structure of his implementation eased my work a lot. He also developed some benchmarks which were used for evaluating the algorithms presented in this thesis.

Many thanks go to the *Fonds zur Förderung der wissenschaftlichen Forschung* (http://www.fwf.ac.at) for the financial support of the projects *Reasoning in Hybrid Knowledge Bases* (FWF P20840) and *Evaluation of Answer Set Programs with External Source Access* (FWF P24090), which gave me the opportunity to work on this interesting topic. Special thanks go to our secretary Eva Nedoma, who was a great help for the organization and accounting of scientific trips to conferences and project partners.

My colleagues from the doctoral programme, Annu, Dasha, Friedrich, Giselle, Johannes, Julia, Martin and Paolo, inspired me in the last years and motivated me each semester to work on new topics which could be presented in our PhD workshop. Their feedback and questions guided me to the right direction. In particular, I want to thank Martin and Johannes for organizing a great student retreat where we could exchange ideas in a very familiar atmosphere, and Friedrich, who is my office mate and has a wonderful dry humor.

Last but not least, I want to thank my family. My greatest thanks go to my beloved parents Karl and Anita for making the education of my choice possible and for continuously supporting me over many years of education and research, both morally and financially. They had an open ear for problems of any kind and always supported my scientific goals, although they sometimes said I should have spent more time on other activities. I owe them everything.

Danksagungen

Mein Dank gilt vor allem meinem Betreuer Prof. Thomas Eiter, der mich bereits zu einem Doktoratsstudium ermutigt hat als ich noch an meinen Diplomarbeiten schrieb. Der zeitgerechte Abschluss wäre ohne seine Unterstützung niemals möglich gewesen, die er mir in meiner Zeit als Dissertant zukommen ließ. Ich schätzte es, bei Zeiteinteilung und Schwerpunktsetzung viele Freiheiten zu haben, und dennoch durch regelmäßige Meetings ein gutes Vorankommen sicherstellen zu können. Mein Zweitbetreuer Prof. Stefan Woltran gab nicht nur wertvolles Feedback zu meiner Dissertation, sondern war auch sehr bemüht meine manchmal drängelnden Fragen zu unserem Doktoratskolleg zu beantworten, selbst bevor dieses offiziell begonnen hatte.

Weiters möchte ich mich bei Michael Fink bedanken, bei dem ich während meiner Studienzeit einige hervorragende Lehrveranstaltungen genießen durfte, die mein Interesse auf den Bereich des Answer Set Programming gezogen haben. Als Co-Author der meisten Publikationen im Umfeld unseres Projektes half er durch ständige Verbesserungsvorschläge, die Ergebnisse in einer überzeugenden Form zu präsentieren. Die Arbeit der letzten Jahre wäre ohne die ständige Hilfe von Thomas Krennwallner viel beschwerlicher gewesen. Er ist unser Experte für fast alles und hatte für Fragen jeglicher Art immer ein offenes Ohr, ganz gleich ob es nun um Publikationen, Implementierungsfragen oder unseren Benchmark-Server ging. Peter Schüller ist einer meiner Vorgänger im DLVHEX-Projekt und unterstützte mich besonders am Beginn meines Doktoratsstudium, als ich mich in den Quellcode einarbeitete. Auch die gute Strukturierung seiner Implementierung vereinfachte meine Erweiterungen enorm. Er entwickelte außerdem einige Benchmarks die zur Evaluierung der Techniken aus dieser Arbeit eingesetzt wurden.

Dem *Fonds zur Förderung der wissenschaftlichen Forschung* (http://www.fwf.ac.at) danke ich für die freundliche finanzielle Unterstützung der Projekte *Reasoning in Hybrid Knowledge Bases* (FWF P20840) und *Evaluation of Answer Set Programs with External Source Access* (FWF P24090), in denen ich während meines Doktoratsstudiums arbeiten durfte. Ein besonderer Dank geht auch an Eva Nedoma von unserem Sekretariat. Sie war bei der Organisation und Abrechnung von Dienstreisen eine große Hilfe.

Meinen Kollegen aus dem Doktoratskolleg, Annu, Dasha, Friedrich, Giselle, Johannes, Julia, Martin und Paolo, danke ich dafür, dass sie mich in den letzten Jahren inspiriert und motiviert haben, in jedem Semester neue Ideen zu erarbeiten die ich in unserem Student-Workshop präsentieren konnte. Ich bedanke mich vor allem für ihr Feedback und ihre Fragen, die mich in die richtige Richtung lenkten. Besonders beigetragen haben Martin und Johannes durch die Organisation eines Workshops, bei dem Ideen in einer familiären Atmosphäre diskutiert werden konnten, sowie mein Bürokollege Friedrich, dessen trockener Humor für viele Lacher sorgte.

Mein größter Dank geht jedoch an meine Familie und vor allem meinen geliebten Eltern Karl und Anita, die mir das Studium meiner Wahl ermöglicht haben und mich über die zahlreichen Jahre meiner Ausbildung und Forschung sowohl moralisch als auch finanziell unterstützt haben. Sie hatten immer ein offenes Ohr für Probleme jeglicher Art, sowie Verständnis für meine beruflichen Ziele und Vorhaben, auch wenn sie der Meinung waren, dass ich manchmal auch mehr Zeit für andere Tätigkeiten hätte abzweigen sollen. All das habe ich ihnen zu verdanken und wäre ohne ihre Hilfe wäre niemals möglich gewesen.

Abstract

Answer set programming (ASP) is a declarative programming approach which has gained increasing attention in the last years. It is useful for many tasks in artificial intelligence, and many language extensions have advanced the paradigm into a strong modeling language.

While the ASP programming paradigm has proved to be fruitful for a range of applications, current trends in distributed systems and the World Wide Web, for instance, revealed the need for access to external sources in a program, ranging from light-weight data access (e.g., XML, RDF, or relational data bases) to knowledge-intensive formalisms (e.g., description logics). To this end, HEX-programs are an extension and generalization of answer set programs by external sources which can be tightly coupled to the reasoner. This is realized by *external atoms*, whose truth value is not determined within the logic program, but by a background theory, which is technically realized as a plugin to the reasoner.

The traditional evaluation algorithm for HEX-programs uses a translation approach which rewrites them to ordinary ASP programs. The fundamental idea is to replace external atoms by ordinary ones whose truth values are guessed. The resulting program is then evaluated by a state-of-the-art ASP solver. The resulting model candidates are subsequently checked for compliance with the external sources, and are discarded if the guesses value differs from the real truth value. While this approach is intuitive and natural, it turned out to be a bottleneck in advanced applications. It does not scale well, as the number of candidate answer sets grows exponentially with the number of external atoms in the program. Moreover, the traditional algorithms also impose very strong syntactic safety conditions on the input program, which restricts the language. This motivates the development of novel evaluation algorithms for HEX-programs, which treat external atoms as first-class citizens and build models from first principles; it is expected that this increases scalability and expressiveness. The thesis consists of two major parts.

In the first part, we present new algorithms for ground HEX-programs, i.e., programs without variables. Conflict-driven learning techniques will be an important basis for our algorithms,

but need to be extended from ordinary ASP solving to HEX-programs. Moreover, minimality checking for model candidates of HEX-programs turned out to be an interesting topic because it causes the major part of the computational costs. Hence, new minimality checking methods will be developed and integrated into the overall evaluation algorithms.

The second part is concerned with HEX-programs with variables in general, and with *value invention* in particular, i.e., the introduction of new constants by external sources, which do not show up in the input program. Traditionally, value invention is restricted by syntactic conditions such that grounding algorithms for ASP programs without external sources are applicable. However, this restricts the expressiveness of the language. Thus the syntactic restrictions shall be relaxed whenever possible, which also requires the development of a new grounding algorithm.

The practical part of this thesis deals with the implementation of the new methods and algorithms in our prototype system DLVHEX. We will analyze and evaluate our work by empirical experiments, and show that the new algorithms provide a much better scalability and richer modeling language, which helps establishing HEX as a practical knowledge representation formalism. We then take a look at some practical applications and extensions of HEX-programs, with focus on those domains which newly emerged or have been significantly extended during the work on this thesis.

Kurzfassung

Antwortmengenprogrammierung (*answer set programming*, kurz *ASP*) ist ein deklarativer Programmieransatz der in den letzten Jahren stark an Popularität gewonnen hat. Sie ist für zahlreiche Probleme im Bereich der künstlichen Intelligenz gut geeignet, und hat sich dank zahlreicher Spracherweiterungen zu einer reichen Modellierungssprache weiterentwickelt.

Während ASP-Systeme bereits für eine Vielzahl von Applikationen im Einsatz sind, erfordern neue Trends, beispielsweise im Bereich der verteilten Systeme und dem World Wide Web, den Zugriff aus einem ASP-Programm auf externe Quellen, wie etwa XML- oder RDF-Dokumente, relationale Datenbanken, oder Formalismen aus dem Bereich der Wissensrepräsentation und -verarbeitung, beispielsweise Beschreibungslogiken (*description logics*). Zu diesem Zweck wurden HEX-Programme entwickelt, die sich als Generalisierung und Erweiterung von ASP verstehen, und die die Anknüpfung von externen Quellen an das ASP-System erlauben. Dies wird über sogenannte *externe Atome* (*external atoms*) erreicht, deren Wahrheitswert nicht im logischen Programm bestimmt, sondern von einer Hintergrundtheorie eingespeist wird, die als Plugin in das ASP-System eingehängt wird.

Der ursprüngliche Ansatz zur Auswertung von HEX-Programmen verwendet eine Übersetzung in gewöhnliche ASP-Programme. Externe Atome werden dabei durch gewöhnliche Atome ersetzt, deren Wahrheitswert nichtdeterministisch geraten wird. Das entstehende Programm kann von herkömmlichen ASP-Systemen ausgewertet werden. Anschließend wird jeder so gewonnene Modellkandidat auf seine Kompatibilität mit der Semantik der externen Quellen getestet und gegebenenfalls verworfen. Zwar ist dieser Ansatz elegant und natürlich, er skaliert aber schlecht für mittelgroße und größere Anwendungen. Außerdem erfordert diese Vorgehensweise die Einhaltung von sehr restriktiven syntaktischen Bedingungen, durch die die Ausdrucksstärke der Sprache eingeschränkt wird. Daher ist das Hauptziel dieser Dissertation die Entwicklung von neuen Evaluierungsalgorithmen für HEX-Programme, die externe Atome von Anfang an als solche behandeln und in die Berechnungen miteinbeziehen. Dadurch soll sowohl die Skalierbarkeit als auch die Ausdruckssärke erhöht werden. Die Arbeit setzt sich aus zwei Hauptteilen zusam-

men.

Im ersten Teil beschäftigen wir uns mit Algorithmen für variablenfreie HEX-Programme. Konflikt-getriebene Techniken (*conflict-driven techniques*) sind ein wichtiger Grundstein für unsere Algorithmen, müssen dazu aber von ASP auf HEX-Programme erweitert werden und externe Atome berücksichtigen. Es hat sich dabei auch herausgestellt, dass der Minimalitätscheck für Modellkandidaten eine wesentliche Rolle spielt, da er einen Großteil des gesamten Rechenaufwands verursacht. Deswegen werden wir uns in einem weiteren Schritt auch damit beschäftigen und neuartige Algorithmen zur Sicherung der Minimalität von Modellen präsentieren.

Der zweite Teil der Arbeit befasst sich mit HEX-Programmen mit Variablen, und insbesondere mit *Domänenerweiterung* durch externe Quellen (*value invention*). Darunter versteht man das Hinzufügen von neuen Konstanten durch externe Quellen, die im ursprünglichen Programm nicht vorkommen. Im bisherigen Ansatz wird dies durch starke syntaktische Einschränkungen so weit beschränkt, dass das Programm mit den in ASP üblichen Methoden in ein variablenfreies Programm übersetzt werden kann. Da dies jedoch auch den Freiraum bei der Modellierung einschränkt, sollen die syntaktischen Einschränkungen gelockert werden, wenn immer das möglich ist.

Der praktische Teil der Arbeit beschäftigt sich mit der Implementierung der neuen Methoden und Algorithmen in unserem Prototypsystem DLVHEX. Damit werden wir unsere Algorithmen auch empirischen Experimenten unterziehen, die zeigen, dass damit eine deutlich bessere Skalierbarkeit erreicht wird, und dass die Modellierungssprache nun deutlich weniger Einschränkungen unterliegt. Dies soll dazu beitragen, HEX zu einem praktisch nutzbaren Formalismus weiterzuentwickeln. Abschließend betrachten wir einige Anwendungen und Erweiterungen von HEX-Programmen, wobei der Fokus auf jenen Anwendungen liegt, die im Zuge dieser Arbeit neu entstanden sind oder wesentlich erweitert wurden.

Contents

Chapter 1

Introduction

Answer Set Programming (ASP) is a declarative programming paradigm which proved to be useful for many problems in artificial intelligence and gained attention as a knowledge representation and reasoning formalism in the last years [Niemelä, 1999; Marek and Truszczyński, 1999; Lifschitz, 2002]. Unlike traditional programming languages, the programmer specifies a description of the desired solution to some search problem rather than an algorithm which computes it. The problem at hand is encoded as logic program such that its solutions can be computed as models using an ASP solver. This approach is based on model finding methods for logic theories and is in spirit of the Satisfiability Solving (SAT) approach [Biere et al., 2009], but is more convenient for the user and has a richer expressiveness for many applications (e.g., transitive closure and programs with variables). Moreover, numerous extensions like aggregates [Pelov et al., 2007; Lee and Meng, 2009; Ferraris, 2011; Faber et al., 2011] and weak constraints [Buccafurri et al., 1997] exist.

The predominant notions of models in this context are *stable models* for normal logic programs [Gelfond and Lifschitz, 1988] and the generalized notion of *answer sets* for (possible disjunctive) logic programs [Gelfond and Lifschitz, 1991]. With both notions, the set of logical consequences from all stable models (resp. all answer sets) does, in general, not necessarily grow with increasing information. This is due to the use of negation-as-failure and is called *nonmonotonicity*. As a simple example, consider the computation of the symmetric difference of two sets. Then the output of the operation does in general not grow monotonically with the two sets. This kind of reasoning is also in spirit of human thinking, where it is very common to make assumptions about the world, which may need to be withdrawn if the available knowledge grows. This is referred to as *default reasoning* or *commonsense reasoning* and was formalized by Reiter (1980) as *default logic*. This work is one of the theoretical foundations of modern answer set programming systems.

While formalisms like Prolog have strong procedural elements, both the stable model and the answer set semantics are *fully declarative*. That is, neither the order of rules nor the or-

der of the literals in rules affect the result and program termination. The answer set semantics is an extension of the stable model semantics. While programs under the latter semantics are called *normal logic programs (NLPs)* and feature only negation-as-failure, the class of *extended logic programs (ELPs)* under the answer set semantics supports in addition also *strong negation* (also often called *classical negation*) in program rules, and programs with disjunctions in rule heads. The answer set semantics has then been further extended and generalized, e.g., to *nested logic programs* [Lifschitz et al., 1999], to programs with aggregates [Pelov et al., 2007; Lee and Meng, 2009; Ferraris, 2011; Faber et al., 2011], or to whole arbitrary propositional theories [Ferraris, 2005].

Answer set programming is well-suited for applications with incomplete and inconsistent information, and for expressing nondeterministic features. The popularity of ASP has especially increased since sophisticated solvers for the respective languages have become available, including DLV [Leone et al., 2006; DLV Website, 2014], SMODELS [Simons et al., 2002; SMODELS Website, 2014], and the CLASP system in the Potassco suite [Gebser et al., 2007a; Gebser et al., 2011b; CLASP Website, 2014]; (see Asparagus Website (2014) for more solvers). Besides many applications in artificial intelligence [Eiter et al., 2011a; Brewka et al., 2011] and data management [Antoniou et al., 2007; Halevy et al., 2003], ASP systems are also increasingly applied in other sciences [Erdogan et al., 2010; Hoehndorf et al., 2007] and commercial applications [Brewka et al., 2011]. Because of its expressiveness, ASP is also a suitable host language for capturing advanced tasks in automated reasoning, like planning, scheduling, or diagnosis. For this purpose, numerous front-ends to ASP solvers are available (the DLV system [Leone et al., 2006], e.g., has several in its distribution).

While the ASP programming paradigm has turned out to be fruitful for a range of applications, current trends in distributed systems and the World Wide Web, for instance, revealed the need for access to external sources in a program, ranging from light-weight data access (e.g., XML, RDF, or relational data bases) to knowledge-intensive formalisms (e.g., description logics).

Although modular aspects of ASP have been considered, e.g., by Janhunen et al. (2009), Dao-Tran et al. (2009a), Järvisalo et al. (2009), and Analyti et al. (2011), those frameworks are limited to logic programs or related formalisms. Also multi-context systems can be seen as modular logic programs. They allow for interlinking multiple knowledge bases (which can be formalized in different host logics) using special bridge rules and axioms that access and import information from other contexts to a local knowledge base, cf. Giunchiglia and Serafini (1994), Brewka and Eiter (2007), Bikakis and Antoniou (2008), and Bikakis and Antoniou (2010). Further extensions of ASP that allow for accessing information in external sources from logic programs include the DLV^{DB} system [Terracina et al., 2008], which allows for querying relational databases from the logic program, and *VI* programs [Calimeri et al., 2007], which allow for importing knowledge from (monotonic) external sources with possibly fresh constants. This thesis focuses on one particular extension, called HEX-*programs* [Eiter et al., 2005], which can be seen as a generalization of other formalisms with external sources. Indeed, many other ASP extensions and related formalisms can be translated to HEX-programs. For instance, Bögl et al. (2010) present a system for inconsistency analysis in multi-context systems by rewriting the problem to a HEX-program.

1.1 Motivation

HEX-programs extend ASP with so called external atoms, through which the user can couple any external data source, which can be represented by a computable function, with a logic program. Roughly, such atoms pass information from the program, given by predicate extensions and constants, to an external source, which returns output values of an (abstract) function that it computes. This is realized as a plugin system which supports the bidirectional communication between the logic program and user-defined library functions. This extension is convenient and very expressive due to the support of recursive data exchange between the logic program and external sources. It has been exploited for applications in different areas, e.g., in the Semantic Web, they have been used as a backend for the SPARQL Query Language for RDF [Prud'hommeaux and Seaborne, 2007], which can be conveniently translated to HEX-programs (see Polleres (2007)), for representing default knowledge in biomedical ontologies [Hoehndorf et al., 2007], and for ontology integration [Eiter et al., 2006c]. Further applications include planning with external functions [Van Nieuwenborgh et al., 2007], ranking services using fuzzy HEX-programs [Heymans and Toma, 2008], geographic information systems [Mosca and Bernini, 2008], complaint management [Zirtiloğlu and Yolum, 2008], multi-context systems [Brewka and Eiter, 2007], querying biological or biomedical ontologies in natural language [Erdogan et al., 2010], and belief (set) merging [Redl et al., 2011]. The latter application is in fact a further extension of HEX-programs by means which allow for the *nesting* of logic programs, i.e., HEX-programs may call further HEX-programs. The realization is based on special external atoms for calling subprograms, passing arguments to them, and accessing their answer sets as objects. This allows for reasoning on the level of *sets* of answer sets, and to aggregate and combine information from different answer sets. All these applications demonstrate the usefulness of integrating external knowledge by means of external atoms.

The traditional evaluation algorithm for HEX-programs uses a translation approach which rewrites them to ASP programs without external sources (we will sometimes call them *ordinary* ASP programs to stress the absence of external sources). The fundamental idea is to guess the truth values of external atoms (i.e., whether a particular fact is in the 'output' of the external source access) in a modified *guessing program*, which is evaluated by a state-of-the-art ASP system. After computing an answer set of the guessing program, a compatibility test checks whether the guesses coincide with the actual source behavior. While this approach is intuitive and natural, it turned out to be a bottleneck in advanced applications. It does not scale well, as the number of candidate answer sets grows exponentially with the number of external atoms in the program. This is because all combinations of truth values are blindly guessed, although many of them fail the final compatibility test, frequently even due to the same reason. However, when treating external sources as black-boxes, there is little room for improvement, as the internals of the model finding process are hidden in the ASP solver, which prevents a pruning of the search space. Hence, even if properties of external sources would be known, it is impossible to make use of them in the translation approach. In addition to efficiency problems, the translation approach suffers also expressiveness restrictions. In order to rewrite HEX-programs to ordinary ASP programs, the output values of external sources need to be known in advance (in general) for grounding purposes. This is currently ensured by strong syntactic criteria which limit the

use of external sources in many cases. These restrictions call for new *genuine* evaluation algorithms, which handle external sourced as *first-class citizens* to increase both efficiency and expressiveness.

1.2 State-of-the-Art

This section reviews existing approaches for three related topics which will be relevant for this thesis: *propositional model building* algorithms for ordinary answer set programs, *grounding methods* for programs with variables, and approaches of answer set programming with access to *external sources and domains*.

1.2.1 Propositional Model Building

We will formally introduce ASP programs as a special case of HEX-programs in Chapter 2. For now we give a more intuitive description. A propositional *answer set program* is a set of rules r of form

$$a_1 \vee \cdots \vee a_k \leftarrow b_1, \ldots, b_m, \text{not } b_{m+1}, \ldots, \text{not } b_n,$$

where not denotes default-negation, and $a_i, 1 \leq i \leq k$, $b_j, 1 \leq j \leq n$ are propositional atoms[1]. The part to the left of $\leftarrow$ is the *head* and the part right the *body* of r.

An interpretation is a consistent set $\mathbf{A}$ of literals[2]. $\mathbf{A}$ *satisfies* a rule if either some b_j for $1 \leq j \leq m$ is not in $\mathbf{A}$, some b_j for $m+1 \leq j \leq n$ is in $\mathbf{A}$, or some a_i for $1 \leq i \leq k$ is in $\mathbf{A}$; $\mathbf{A}$ *satisfies* a program Π (or is a *model* of Π) if $\mathbf{A}$ satisfies each rule r in Π. An interpretation $\mathbf{A}$ is an *answer set* of Π if $\mathbf{A}$ is a subset-minimal model of the program $f\Pi^{\mathbf{A}} = \{r \in \Pi \mid \{b_1, \ldots, b_m\} \subseteq \mathbf{A}, \{b_{m+1}, \ldots, b_n\} \cap \mathbf{A} = \emptyset\}$, which is called the *FLP-reduct of* Π *with respect to* $\mathbf{A}$ and consists of all rules whose body is satisfied by $\mathbf{A}$ [Faber et al., 2011]. For ordinary programs the FLP-reduct is equivalent to the seminal GL-reduct introduced by Gelfond and Lifschitz (1991), but preferable for programs with aggregates and HEX-programs.

Model building algorithms for such programs can be classified in two major groups. The first one consists of algorithms that translate the set of rules into another host logic (e.g., propositional logic or difference logic), for which one can apply specialized SAT solvers. This results in a *reduction* of the problem, i.e., the solutions to a constructed SAT instance can be used to construct the answer sets of the original problem. Approaches of the second kind, on which we focus in this thesis, search directly for models and are called *genuine* algorithms, cf. Giunchiglia et al. (2008) and Baral (2002). The underlying idea of genuine algorithms, such as those implemented by DLV or SMODELS, is to perform an intelligent (restricted) enumeration of truth assignments to atoms used in the search for an answer set. That is, deterministic consequences of the rules wrt. partial truth assignments are computed [Giunchiglia et al., 2008] in order to set the truth values of further atoms; e.g., if the body of a rule is satisfied, also its

[1] Strongly (classically) negated atoms $\neg p$ can be seen as new atoms together with a constraint which forbids that p and $\neg p$ are simultaneously true.

[2] For now this definition of an interpretation suffices, although we will introduce a more general one in Section 2 in order to support also partial assignments.

head must be satisfied. If the assignment is still partial after all deterministic consequences have been drawn, the value of a yet undefined atom is guessed in the style of DPLL algorithms for SAT, and again deterministic consequences are determined, etc.; in case the guess leads to a contradiction, the computation backtracks and the alternative value is considered.

In contrast to DLV and SMODELS, the CLASP system employs a conflict-driven method corresponding to conflict driven SAT solvers [Mitchell, 2005]. After some preprocessing steps (e.g., rewriting of optimization statements [Gebser et al., 2011a]), the solver creates a set of *nogoods* for the input program [Gebser et al., 2007a], where a nogood is a set of literals that must not occur simultanously in an answer set. For instance, it must never happen that all body literals of some rule are true, but the body as a whole (represented by an auxiliary variable) is not. However, as there are exponentially many nogoods, some of them (the so called *loop nogoods* which avoid cyclic justifications [Lin and Zhao, 2004; Gebser et al., 2007a]) are only introduced on the fly. The basic operation of the algorithm is then *unit propagation*: if there is some nogood with all except one literal satisfied, then the last literal must be false. This inference step is repeated as long as new literals can be derived, i.e., until a fixpoint is reached. If no further literal can be derived but some atoms have no truth value yet, then the algorithm guesses a truth value for some such atom just as explained before. However, the distinguishing feature of conflict-driven algorithms is *nogood learning*. Whenever a conflict emerges, the literals which were initially responsible for the conflict, are determined. This possibly results in adding further nogoods which prevent the algorithm from reconstructing an interpretation with the same conflict again. This considerably restricts the search space: instead of backtracking linearly, the reasoner immediately jumps to the assignment that initially caused the contradiction, and guides the algorithm into another part of the search space. As conflict-driven algorithms are predominant in modern SAT and ASP solving algorithms, we want to built upon them and introduce them formally and in more detail in Chapter 2.

1.2.2 Grounding Methods

Non-ground answer set programs are like propositional programs, but the atoms in a rule are of the form $p(t_1, \ldots, t_n)$, where p is a (first-order) predicate and the t_i are terms in a first order language. The semantics of such a program Π is defined in terms of its *grounding*, which consists of all possible ground instances of the rules in Π, i.e., variable-free rules that result by replacing all variables in r by ground terms in the language in all possible ways. For ordinary ASP programs the grounding is finite, while it may be infinite for certain extensions like ASP with function symbols and HEX-programs. However, in practice suitable safety conditions guarantee that only a finite subset of the grounding is relevant for answer set computation. Finding such safety conditions and developing efficient grounding algorithms for the resulting class of HEX-programs will be in the focus of Chapter 4.

Most state-of-the-art ASP solvers (including DLV and CLASP) step to the grounding of a program before the actual model finding algorithms are started. However, this is not done naively by plugging in each constant for each variable. Instead, the grounder usually employs advanced optimization techniques which try to eliminate irrelevant rules upfront. Modern grounders, like the ones incorporated in DLV [Faber et al., 1999; Leone et al., 2001; Calimeri et al., 2008b], or LPARSE [Syrjänen, 2009; SMODELS Website, 2014], and the GRINGO system as part of the

Potassco suite [Gebser et al., 2011b; GRINGO Website, 2014], in fact compute the answer set for monotonic programs or program components, such that only nondeterministic choices need to be handled by the actual solver.

In contrast to pre-grounding, *lazy grounding* generates ground instances of rules only during reasoning when the positive part of the body of a rule is already satisfied. This technique is used for instance in GASP [Palù et al., 2009] and in the ASPeRiX solver [Lefèvre and Nicolas, 2009]. An advantage is that generating irrelevant ground rules can be avoided more effectively. However, the complexity of rule applications is higher than in the pre-grounding case since the matching algorithm cannot compare rule bodies one-by-one with the current partial interpretation, but must check if any grounded version is satisfied. Empirical results by Palù et al. (2009) are encouraging. Because HEX-programs cannot simply be pre-grounded since parts of the relevant domain may never appear in the input program, we will partially build upon lazy grounding techniques. Actually, we will choose a *hybrid approach* in Chapter 4, which alternates evaluation and grounding for fragments of HEX-programs. That is, we instantiate program components larger than single rules, but we do not carry out to overall grounding prior to evaluation but interleave the two processes.

1.2.3 External Sources and Domains

There exist formalisms other than HEX-programs which have a similar intention and support external sources of computation. We now recall them and describe the most important differences to HEX-programs.

GRINGO and Lua interface

The GRINGO system is a grounder for ASP which provides an interface for calling functions written in the scripting language Lua [Lua Website, 2014] at certain points during the grounding process [Gebser et al., 2011b]. The functions may access GRINGO data structures and return new values to later grounding phases, which allows, e.g., to sort the input, to retrieve tuples from a relational database and add them as facts to the grounder, as well as to insert atoms of a model into the database. This is well-suited for implementing, for instance, user-defined built-in predicates. However, in contrast to HEX-programs, the communication between the ASP system and external scripts is only possible between specific grounding phases and is not tightly coupled to and interleaved with model building.

ASP modulo Theories

The ASP solver CLASP provides an interface for adding custom theory propagators to the reasoner, which are executed after unit and unfounded set propagation have finished. This interface was exploited by the CLINGCON system for integrating ASP with constraint satisfaction programming [Ostrowski and Schaub, 2012; Gebser et al., 2009]. The CLINGCON approach can be seen as a special case of HEX-programs: while it implements a solver for a specific theory, the HEX formalism abstractly couples a large variety of different external sources to the solver.

DLV−EX and DLV-Complex

DLV−EX, which is now part of DLV-Complex, is an extension of DLV which provides external predicates similar to external atoms in HEX-programs. This allows for accessing sources of computation that are defined outside the logic program [Calimeri et al., 2007], which is helpful for functions that are not conveniently or not efficiently expressible by rules (for instance mathematical functions). As for HEX-programs, external computations possibly extend the Herbrand universe of a logic program. Obviously, DLV−EX is related to HEX-programs, but a closer look reveals that it is less general since it only allows for passing terms as input parameters to external libraries, while HEX-programs allow for passing complete (or partial) interpretations by the use of predicate parameters [Eiter et al., 2006a]. Consequently, external atoms in HEX-programs are inherently more difficult to evaluate. The difficulty comes especially from nonmonotonic behavior, which is not possible if only terms can be input to external sources.

1.3 Contributions

The overall goal of this thesis is therefore the development of *advanced reasoning algorithms* which avoid the simple ASP translation approach in order to overcome the evaluation bottleneck of HEX-programs. This class of algorithms will be called *genuine algorithms* throughout this thesis. In contrast to the translation approach, they consider external atoms as first-class citizens and natively build model candidates from first principles and accesses external sources already during the model search, which allows to prune candidates early. For this purpose, they may also exploit meta-knowledge about the internals of external sources, such as asserted properties like monotonicity and functionality. These ideas are integrated with modern SAT and ASP solving techniques based on *clause learning* [Biere et al., 2009], which led to very efficient *conflict-driven* algorithms for (possibly disjunctive) answer-set computation [Drescher et al., 2008]. We extend them to external sources, which is a major contribution of this work. Since typical reasoning tasks over HEX-programs such as cautious and brave reasoning are on the second level of the polynomial hierarchy, the development of efficient algorithms is challenging.

As another important contribution we will also provide algorithms which handle programs with variables and possible *value invention*, i.e., external sources which return constants that do not show up in the original program. To this end, we will develop new *safety criteria* which restrict the use of external sources less than other approaches, as e.g. those presented by Gebser et al. (2007b) and Calimeri et al. (2007), but such that infinite value invention is still avoided. For this novel class of programs we then introduce an *efficient grounding algorithm*.

As a proof of concept, the new algorithm will be integrated into our prototype system DLVHEX, which is, to the best of our knowledge, the only implementation of the HEX-semantics. The implementation is designed in an extensible fashion, such that the provider of external sources can specify refined *learning functions* which exploit domain-specific knowledge about the source. Also the safety criteria for programs with variables are implemented in an extensible fashion such that application-specific knowledge can be exploited in addition to built-in criteria. The theoretical work is complemented with experiments that we conducted with our prototype on synthetic benchmarks and programs motivated by real-world applications. In many cases,

significant performance improvements compared to the previous algorithm are obtained, which shows the suitability and potential of the new techniques.

Finally, we discuss some existing and new applications and extensions of HEX-programs. We will focus on those applications which emerged or have been significantly extended during the work on this thesis, but we will also briefly discuss some traditional applications.

1.4 Organization of this Thesis

The remaining part of this thesis is organized as follows:

- In Chapter 2 we provide background about the concepts and techniques we will use throughout this thesis. In particular, we introduce the syntax, semantics and the traditional evaluation algorithm for HEX-programs, which is based on a translation to ASP. Moreover, we present state-of-the-art SAT and ASP solving algorithms based on *conflict-driven clause learning*.

- In Chapter 3 we present novel evaluation algorithms for *ground* HEX*-programs*, i.e., variable-free programs. Our approach integrates conflict-driven algorithms with additional learning concepts related to techniques used in solvers for *SAT modulo theories (SMT)* [Barrett et al., 2009]. We further develop new algorithms for minimality checking of answer set candidates. This is an important topic because in many cases the major part of the overall computational costs is caused by this check. The new minimality check is tightly integrated with the conflict-driven algorithms.

  In this chapter, we will also consider syntactic fragments of HEX-programs, which often allow for a more efficient evaluation.

- In Chapter 4 we address non-ground programs and value invention (also called *domain expansion*) in particular. That is, we consider external sources which may introduce new values which do not appear in the original program. This obviously prevents naive pre-grounding, as used in ordinary ASP solving, and requires *additional safety criteria*.

  As the traditionally used criteria are unnecessarily restrictive, an important goal in this chapter will be to *relax these criteria* whenever possible. However, we do not simply provide more liberal safety criteria, but a *generic and extensible notion of safety*, where concrete (syntactic and semantic) safety criteria can be plugged in. We then provide examples for *concrete safety criteria*, and prove that they are already *strictly more general* than various other notions from the literature.

  After the theoretical work on safety of HEX-programs, we will provide a *grounding algorithm* for the defined class of HEX-programs. This algorithm is then *integrated into the evaluation framework*, which is extended for this purpose.

- In Chapter 5 we first provide some information about the implementation of our algorithms in our prototype system DLVHEX. The system is available from http://www.kr.tuwien.ac.at/research/systems/dlvhex as open-source software. Then we evaluate the

system using a newly developed benchmark suite, which consists both of synthetic and of real-world applications. We compare our algorithms to the traditional translation-based algorithms and will be able to show a significant, in some cases even exponential, speedup. The evaluation addressed both the learning-based solving and the grounding algorithms which we developed in Chapters 3 and 4, respectively.

- Chapter 6 discusses applications and extensions of HEX-programs. The focus is on new applications which emerged during the work on this thesis or have been significantly extended, but we will also recapitulate important traditional applications.

- Chapter 7 summarizes the main results, concludes the thesis, and gives an outlook on future work.

- The HEX-encodings of benchmark problems are shown in Appendix A, lengthy proofs of some presented results are outsourced in Appendix B.

1.5 Publications and Evolution of this Work

We now give a brief overview about the evolution of the techniques over time. We further give references to the initial publications of the results presented this thesis.

This PhD project started started in October 2011 after the initial version of the model-building framework has been introduced and the traditional algorithms have been integrated into this framework. However, at this stage there was no tight integration of the ASP solver used as backend with the reasoning algorithms for HEX-programs. In particular, there were no learning techniques which consider external atoms as first-class citizens. Instead, the evaluation of the logic program and the external sources were strictly separated, which not only turned out to be an efficiency bottleneck in the evaluation, but also required strong syntactic limitations.

Thus, the goal in the first phase of the project from October 2011 to December 2012 was the development of novel evaluation algorithms for ground HEX-programs presented in Chapter 3. The main task was the development of the guess and check algorithm from Section 3.1, which adds new learning techniques that are specific for ASP with external sources. We developed the concept of *learning functions* in order to abstractly deal with external sources and allow for flexible instantiation for concrete applications. The results were initially published in the *Journal of Theory and Practice of Logic Programming (TPLP)* in September 2012 [Eiter et al., 2012a] and included preliminary benchmark results. In parallel to this part of the theoretical work, GRINGO and CLASP were integrated into our prototype system and replaced DLV as default reasoning backend. This work was mainly carried out during a research visit at the University of Potsdam, Germany in the group of Prof. Dr. Torsten Schaub in January and February 2012.

A subprocedure of this algorithm is the *minimality check* from Section 3.2. While the traditional approach used an explicit search for smaller models, the new approach is based on unfounded sets, which were previously already exploited for normal logic programs and disjunctive ASP. We presented the new minimality checking algorithm and initial benchmark results at the *Thirteenth European Conference on Logics in Artificial Intelligence (JELIA 2012)* in September 2012 [Eiter et al., 2012c]. We then developed the idea that this step might not be necessary in

all cases, which was soon confirmed by formal results. This led to the identification of a *decision criterion* which was published in the proceedings of the *Fifth Workshop on Answer Set Programming and Other Computing Paradigms* in September 2012 [Eiter et al., 2012b].

The minimality check was then further improved by using a more advanced encoding of the unfounded set search. Compared to the initial technique, the improved version minimizes repetitive computation and led to a further speedup. We presented the final version of this technique and the respective benchmark encodings and results in a technical report [Eiter et al., 2013d] and in the *Journal of Artificial Intelligence Research* in 2014 [Eiter et al., 2014b], which coincide with the contents presented in this thesis.

In the second phase of this PhD project from January 2013 to March 2014 we considered *grounding* and *domain expansion*. The results are presented in Chapter 4. The first goal in this phase was the relaxation of the syntactic restrictions. The traditional grounding algorithm assumes that all constants are known in advance whenever external atoms are involved in cycles. This requires the user to obey restrictive *safety conditions*. It turned out that in many cases these restrictions are unnecessary, i.e., better algorithms do not need such restrictions. Based on this observation we developed a new notion of safety which we called *liberal domain-expansion safety* (Section 4.2). However, this notion does not only relax the previous one but is also extensible and customizable. This led to the concept of *liberal safety* which we presented at the *Twenty-Seventh AAAI Conference (AAAI 2013)* in July 2013 [Eiter et al., 2013c].

The theoretical work was then complemented by a new *grounding algorithm* for HEX-programs which are safe according to the new notion (Section 4.3). The algorithm and benchmark results were presented at the *Second Workshop on Grounding and Transformations for Theories with Variables (GTTV 2013)* in September 2013 [Eiter et al., 2013a]. This work also presents a new heuristics for the model-building framework (Section 4.5), which is tailored to the properties of the new grounding algorithm and aims at avoiding the worst-case of this algorithm, which is possible in many cases. However, this task is challenging since the heuristics influences both the grounding and the solving algorithm and some strategies might reduce the grounding time but increase the solving time or vice versa.

In parallel to both phases of the project we also continuously developed new *applications* of HEX-programs and extended existing ones, some of which are presented in Chapter 6. In particular, the idea of HEX-*programs with nested program calls* which was initially developed by Redl (2010) and published in the proceedings of the *Thirteenth International Symposium on Practical Aspects of Declarative Languages (PADL 2011)* in January 2011 [Redl et al., 2011] was significantly extended in order to serve as a development tool during the implementation of the algorithms from this thesis. The results have been presented at the *Nineteenth International Conference on Applications of Declarative Programming and Knowledge Management (INAP 2011)* [Eiter et al., 2011b] and in more detail in the post-proceedings of the conference [Eiter et al., 2013f]. Another application are HEX-*programs with existential quantification* that were mainly developed in order to show the generality of the new grounding algorithm. We presented them at the *Twentieth International Conference on Applications of Declarative Programming and Knowledge Management (INAP 2013)* [Eiter et al., 2013b] and in more detail in the post-proceedings [Eiter et al., 2014a]. Less related but still in context of this thesis is an *alternative semantics* for ASP programs with aggregates and HEX-programs, for which

we provided an implementation which was presented in the *Journal of Arificial Intelligence* in 2014 [Shen et al., 2014].

The development of the previously mentioned applications was tightly coupled with the development of the methods from this thesis and influenced their fine-tuning. However, while the work on this thesis was in progress some applications were also developed by end users of the system. Although the core team was involved in those applications on a more abstract level as well, the details were developed and implemented by persons who do not directly work on the reasoning algorithms in the system core. Thus, the encodings of those applications can not unintentionally bias the results by guiding the algorithms (i.e., avoiding problematic cases), and can thus be considered as real-world applications. One such application is ACTHEX, which has been significantly extended during the work on this thesis[3]. The results have been published in the proceedings of the *Twelfth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2013)* in September 2013 [Fink et al., 2013]. In summer 2013 we further participated at the *AngryBirds AI Competition*[4] with an agent based on HEX-programs [Calimeri et al., 2013a]. The agent is called *AngryHEX* and was joint work of the University of Calabria and the Vienna University of Technology[5]. Later, the system was significantly extended, e.g., by introducing a second strategy layer for choosing among levels, and more details were presented at the *National Workshop and Prize on Popularize Artificial Intelligence (PAI 2013)* [Calimeri et al., 2013b], where we were conferred the best paper award; in the competition the system was in the semifinal. Finally, a prototypical implementation of constraint ASP on top of HEX-programs was developed as part of a master's thesis [Stashuk, 2013].

[3]The implementation was mainly carried out by our guest student Stefano Germano in summer 2012, to whom the author is grateful for his work.

[4]Co-located with IJCAI 2013, Beijing, China; http://www.aibirds.org

[5]We are grateful to Daria Stepanova, who presented the poster as a deputy of the AngryHEX team.

Chapter 2

Preliminaries

In this chapter we discuss the background of this thesis, which will also show the notions we are going to use. We first introduce syntax and semantics of HEX-programs as an extension of the answer set programming (ASP) paradigm. This knowledge representation and reasoning formalism, which was first introduced by Eiter et al. (2005) and described in more detail by Schindlauer (2006), is in the focus of this thesis.

The first evaluation algorithm for HEX-programs was introduced by Schindlauer (2006), but has been significantly changed since then. In particular, a flexible *model building framework* was introduced by Eiter et al. (2011a) and Schüller (2012), which evaluates HEX-programs by modular decomposition of the input program, driven by some heuristics. The original algorithms of Schindlauer (2006) are then obtained by instantiating the framework with a specific decomposition heuristics. However, we will directly recapitulate the work of Schüller (2012) instead of Schindlauer (2006) because it is strictly more general and leads to significantly better benchmark results.

We then present conflict-driven SAT and answer set solving, following Mitchell (2005) and Drescher et al. (2008), as this is a very promising technique which is fundamental to state-of-the-art solvers. This is the basis for the algorithms developed in later chapters.

2.1 HEX-Programs

HEX-programs have been introduced by Eiter et al. (2005) as a generalization of (disjunctive) extended logic programs under the answer set semantics [Gelfond and Lifschitz, 1991]. The latter will sometimes be called *ordinary ASP* throughout this thesis to stress the absence of external sources. In order to cater for the requirements of modern trends in distributed systems and the World Wide Web, HEX-programs provide a universal bidirectional interface between the logic program and external sources of computation, which is realized as *external atom*. External

atoms allow for a tight integration of arbitrary sources, which are provided as plugins to the reasoner, with the logic program.

We start our discussion with some basic notions that we need in order to formally define the syntax and semantics of HEX-programs. We assume that for a given program the alphabet consists of mutually disjoint sets

- $\mathcal{C}$ of *constants*;
- $\mathcal{P}$ of *predicates*;
- $\mathcal{V}$ of *variables*; and
- $\mathcal{X}$ of *external predicates*.

Noticeably, $\mathcal{C}$ may be larger than the set of constants which explicitly show up in the program and can even be infinite. In our examples we adopt the following naming convention: constants start with lower case letters at the beginning of the alphabet ($a, b, \ldots$), predicates start with lower case letters beginning from p ($p, q, \ldots$), variables start with upper case letters ($X, Y, \ldots$), and external predicate names start with symbol *&* (*&*g, *&*$h, \ldots$). The set of *terms* is defined as $\mathcal{C} \cup \mathcal{V}$. Note that we have no notion of function symbols because they are disallowed in HEX-programs, but can be simulated by external atoms as we will show in Chapter 6.

For an ordinary predicate $p \in \mathcal{P}$, let $ar(p)$ denote the arity of p and for an external predicate *&*$g \in \mathcal{X}$, let $ar_{\text{I}}($*&*$g)$ be the (fixed) *input arity* and $ar_{\text{O}}($*&*$g)$ the (fixed) *output arity* of *&*g[1].

As with ordinary ASP programs, the basic building blocks of HEX-programs are atoms.

Definition 1 (Atom)**.** An (ordinary) atom a is of form $p(t_1, \ldots, t_\ell)$, with predicate $p \in \mathcal{P}$ and $\ell = ar(p)$ and terms $t_1, \ldots, t_\ell \in \mathcal{C} \cup \mathcal{V}$.

Sometimes we will call an atom also *ordinary atom*, to stress that we do not talk about external atoms (which are formally introduced in the following). An atom $p(t_1, \ldots, t_\ell)$ is called *ground* if all terms are constants, i.e., $t_i \in \mathcal{C}$ for all $1 \leq i \leq \ell$.

In this thesis we will often use lists of elements $l_1, \ldots, l_\ell$, which will be compactly notated by bold face, i.e., $\mathbf{l} = l_1, \ldots, l_\ell$. For instance, an atom of form $p(t_1, \ldots, t_\ell)$ might be denoted $p(\mathbf{t})$. For a list $\mathbf{l} = l_1, \ldots, l_\ell$ we write $l \in \mathbf{l}$ if $l = l_i$ for some $1 \leq i \leq \ell$. When we explicitly write a list $\mathbf{l} = l_1, \ldots, l_\ell$ of length $\ell > 1$, we may add parentheses for better readability, i.e., we write $(l_1, \ldots, l_\ell)$. Moreover, whenever we write l_i in context of a list $\mathbf{l}$ whose elements are not explicitly stated, we assume that l_i refers to the i-th element of the list, i.e., the list is implicitly given by $\mathbf{l} = l_1, \ldots, l_\ell$.

Since we need to represent *partial assignments* in many sections of this thesis, we cannot simply represent assignments as sets of atoms, as in the intuitive description in Chapter 1. This would not allow for distinguishing between false and undefined atoms. Instead, we use the following notion of assignments based on signed literals.

Definition 2 (Signed Literal)**.** A *(signed) literal* is a positive or negated ground atom $\mathbf{T}a$ or $\mathbf{F}a$.

[1]For user convenience the implementation supports external predicates with variable input arity in some cases, but we restrict our formal investigation to external predicates with fixed input arity.

For a literal $\sigma = \mathbf{T}a$ or $\sigma = \mathbf{F}a$, let $\overline{\sigma}$ denote its opposite, i.e. $\overline{\mathbf{T}a} = \mathbf{F}a$ and $\overline{\mathbf{F}a} = \mathbf{T}a$.

Definition 3 (Assignment)**.** An assignment $\mathbf{A}$ over a (finite) set of atoms A is a *consistent* (i.e, for any atom $a \in A$, we have $\{\mathbf{T}a, \mathbf{F}a\} \not\subseteq \mathbf{A}$) set of signed literals $\mathbf{T}a$ or $\mathbf{F}a$, where $\mathbf{T}a$ expresses that a is true and $\mathbf{F}a$ that it is false.

An assignment $\mathbf{A}$ is called *complete* wrt. a set of atoms A (or an *interpretation* of the atoms in A), if $\mathbf{T}a \in \mathbf{A}$ or $\mathbf{F}a \in \mathbf{A}$ for all $a \in A$.

We write $\mathbf{A}^{\mathbf{T}}$ to refer to the set of elements $\mathbf{A}^{\mathbf{T}} = \{a \mid \mathbf{T}a \in \mathbf{A}\}$ and $\mathbf{A}^{\mathbf{F}}$ to refer to $\mathbf{A}^{\mathbf{F}} = \{a \mid \mathbf{F}a \in \mathbf{A}\}$. In abuse of notation, we will sometimes write complete interpretations $\mathbf{A}$ as the set of its positive signed literals $\{\mathbf{T}a \in \mathbf{A}\}$, or as the set of the atoms $\{a \mid \mathbf{T}a \in \mathbf{A}\}$ which are true in $\mathbf{A}$, if the set of atoms A is clear from context.

We will often need to refer to all tuples of arguments $\mathbf{c}$, for which a certain predicate p is true in an assignment $\mathbf{A}$. This is called the *extension of p in $\mathbf{A}$*.

Definition 4 (Extension of a Predicate)**.** The *extension of a predicate* p wrt. an assignment $\mathbf{A}$ is defined as $ext(p, \mathbf{A}) = \{\mathbf{c} \mid \mathbf{T}p(\mathbf{c}) \in \mathbf{A}\}$.

Let further $\mathbf{A}|_p$ be the set of all signed literals over atoms of form $p(\mathbf{c})$ in $\mathbf{A}$. For a list $\mathbf{p} = p_1, \ldots, p_k$ of predicates, we let $\mathbf{A}|_{\mathbf{p}} = \mathbf{A}|_{p_1} \cup \cdots \cup \mathbf{A}|_{p_k}$.

2.1.1 Syntax

We are now ready to introduce HEX-programs formally. Ordinary ASP programs correspond then to a fragment of HEX-programs, which we will describe subsequently. We start the introduction of the syntax of HEX-programs with external atoms as the most specific part of the language.

Definition 5 (External Atom)**.** An *external atom* is of the form

$$\&g[Y_1, \ldots, Y_k](X_1, \ldots, X_l),$$

where $g \in \mathcal{X}$ is an external predicate name with $ar_{\mathrm{I}}(\&g) = k$ and $ar_{\mathrm{O}}(\&g) = l$, $Y_i \in \mathcal{C} \cup \mathcal{P} \cup \mathcal{V}$ for all $1 \leq i \leq k$ are *input terms*, and $X_i \in \mathcal{C} \cup \mathcal{V}$ for all $1 \leq i \leq l$ are *output terms*.

The lists $\mathbf{Y} = Y_1, \ldots, Y_k$ and $\mathbf{X} = X_1, \ldots, X_l$ are called *input* and *output list*, respectively. A predicate in the input list is called an *input predicate*.

An external atom $\&g[Y_1, \ldots, Y_k](X_1, \ldots, X_l)$ is *ground*, if $Y_i \in \mathcal{C} \cup \mathcal{P}$ for all $1 \leq i \leq k$ and $X_i \in \mathcal{C}$ for all $1 \leq i \leq l$. Using our list notation, we abbreviate $\&g[Y_1, \ldots, Y_k](X_1, \ldots, X_l)$ as $\&g[\mathbf{Y}](\mathbf{X})$.

We further assume that the input parameters of every external predicate $\&g \in \mathcal{X}$ are typed such that $type(\&g, i) \in \{\mathbf{const}, \mathbf{pred}\}$ for every $1 \leq i \leq ar_{\mathrm{I}}(\&g)$. We make also the restriction that $Y_i \in \mathcal{P}$ if $type(\&g, i) = \mathbf{pred}$ and $Y_i \in \mathcal{C} \cup \mathcal{V}$ otherwise. Intuitively, for a parameter of type $\mathbf{const}$ the constant at the respective argument position is passed to the external source, while for a parameter of type $\mathbf{pred}$ the extension of the given predicate name is passed. Moreover, we will sometimes assume that we know for predicate parameters if they are *monotonic* or

nonmonotonic. A formal definition these properties will follow in Chapter 3, for now we only use them as tags to the external source and explain them intuitively as follows. An external atom is monotonic in a predicate parameter p, if the output of the external atom does not shrink if the extension of p grows, otherwise it is nonmonotonic in p.

Example 1. A non-ground external atom is $\&diff[set1, set2](X)$, a ground external atom is $\&diff[set1, set2](a)$, where $type(\&diff, 1) = type(\&diff, 2) = \mathbf{pred}$. □

HEX-programs are then defined similar to ordinary ASP programs.

Definition 6 (HEX-programs)**.** A HEX-program is a finite set of rules of form

$$a_1 \vee \cdots \vee a_k \leftarrow b_1, \ldots, b_m, \mathrm{not}\, b_{m+1}, \ldots, \mathrm{not}\, b_n,$$

where each a_i for $1 \leq i \leq k$ is an atom $p(t_1, \ldots, t_\ell)$ with terms t_j, $1 \leq j \leq \ell$, and each b_i for $1 \leq i \leq n$ is either an ordinary (classical) atom or an external atom. Moreover, we require $k + n > 0$, and call a rule a *constraint* if $k = 0$, and a *fact* or *disjunctive fact* if $n = 0, k = 1$ or $n = 0, k > 1$, respectively. For facts and disjunctive facts we may omit $\leftarrow$. Sometimes we terminate rules with a dot if this improves readability.

The *head* of a rule r is defined as $H(r) = \{a_1, \ldots, a_n\}$ and the *body* is defined as $B(r) = \{b_1, \ldots, b_m, \mathrm{not}\, b_{m+1}, \ldots, \mathrm{not}\, b_n\}$. We call b or $\mathrm{not}\, b$ in a rule body a *default literal*; $B^+(r) = \{b_1, \ldots, b_m\}$ is the *positive body*, $B^-(r) = \{b_{m+1}, \ldots, b_n\}$ is the *negative body*.

A rule is *ground* if all atoms and external atoms are ground. A program is *ground* if all rules are ground.

Example 2. The following set of rules forms a non-ground HEX-program Π:

$$\begin{aligned} sel(X) &\leftarrow domain(X), \&diff[domain, nsel](X) \\ nsel(X) &\leftarrow domain(X), \&diff[domain, sel](X) \\ domain(a) &\leftarrow \end{aligned}$$

□

As already mentioned in Footnote [1] in Chapter 1, we do not formally introduce strong negation but see classical literals of form $\neg a$ as new atoms together with a constraint which disallows that a and $\neg a$ are simultaneously true.

We next define groundings of rules and programs similar to Gelfond and Lifschitz (1991).

Definition 7 (Grounding)**.** The *grounding* $grnd_{\mathcal{C}}(r)$ of a rule r wrt. a set of constants $\mathcal{C}$ is the set of all rules $\{\sigma(r) \mid \sigma \colon \mathcal{V} \mapsto \mathcal{C}\}$, where σ is a *grounding substitution* mapping each variable to a constant, and $\sigma(r)$ denotes the rule which results if each variable X in r is replaced by $\sigma(X)$.

The *grounding of a program* Π is defined as $grnd_{\mathcal{C}}(\Pi) = \bigcup_{r \in \Pi} grnd_{\mathcal{C}}(r)$.

Example 3 (ctd.)**.** Let Π be the HEX-program from Example 2 and $\mathcal{C} = \{a\}$ be a set of constants. Then $grnd_{\mathcal{C}}(\Pi)$ is:

$$sel(a) \leftarrow domain(a), \&diff[domain, nsel](a)$$
$$nsel(a) \leftarrow domain(a), \&diff[domain, sel](a)$$
$$domain(a) \leftarrow$$

□

Note that $grnd_{\mathcal{C}}(r)$ and $grnd_{\mathcal{C}}(\Pi)$ may be infinite because $\mathcal{C}$ may be infinite. It will be the focus of Chapter 4 to identify classes of programs which have a finite answer-set preserving grounding, i.e., a finite grounding with the same answer sets as the complete grounding wrt. $\mathcal{C}$.

Definition 8. The *Herbrand base* HB_Π of program Π is the set of all ground atoms constructible from the predicates occurring in Π and the constants from $\mathcal{C}$.

We further let $A(r)$ and $A(\Pi)$ denote the set of ordinary atoms in a rule r or in a program Π, respectively, and $EA(r)$ and $EA(\Pi)$ denote the set of external atoms in a rule r or in a program Π, respectively. Importantly, the sets $A(\cdot)$ and $EA(\cdot)$ include the atoms and external atoms, which occur in default-negated form in the given rule or program.

2.1.2 Semantics

We start with the semantics of ordinary ground atoms wrt. an assignment.

Definition 9 (Satisfaction of Ground Atoms)**.** A ground atom $p(\mathbf{c})$ is *true* in assignment $\mathbf{A}$, denoted $\mathbf{A} \models p(\mathbf{c})$, if $\mathbf{T}p(\mathbf{c}) \in \mathbf{A}$, and *false* in assignment $\mathbf{A}$, if $\mathbf{F}p(\mathbf{c}) \in \mathbf{A}$.

To evaluate an external atom the reasoner passes the constants and extensions of the predicates in the list of input terms to the external source associated with the external atom, which is plugged into the reasoner. The external source computes a set of output tuples, which are matched with the output list. The external atom evaluates then to true for every output tuple which matches with any of the tuples returned from the external source.

The semantics of a ground external atom $\&g[\mathbf{y}](\mathbf{x})$ wrt. an assignment $\mathbf{A}$ is given by the value of a $1{+}k{+}l$-ary Boolean *oracle function* $f_{\&g}$, where $ar_{\text{I}}(\&g) = k$ and $ar_{\text{O}}(\&g) = l$, that is defined for all possible values of $\mathbf{A}$, $\mathbf{y}$ and $\mathbf{x}$.

Definition 10 (Satisfaction of Ground External Atoms)**.** A ground external atom $\&g[\mathbf{y}](\mathbf{x})$ is *true* in assignment $\mathbf{A}$, denoted $\mathbf{A} \models \&g[\mathbf{p}](\mathbf{c})$, if $f_{\&g}(\mathbf{A}, \mathbf{y}, \mathbf{x}) = 1$, and *false* in assignment $\mathbf{A}$ if $f_{\&g}(\mathbf{A}, \mathbf{y}, \mathbf{x}) = 0$.

Importantly, in this thesis we restrict oracle functions $f_{\&g}$ for a given external predicate $\&g$ with input list $\mathbf{y}$ to *computable* functions s.t. the set $\{\mathbf{x} \mid f_{\&g}(\mathbf{A}, \mathbf{y}, \mathbf{x}) = 1\}$ is *finite* and *enumerable*. We further assume that $f_{\&g}(\mathbf{A}, \mathbf{y}, \mathbf{x}) = f_{\&g}(\mathbf{A}', \mathbf{y}, \mathbf{x})$ for all $\mathbf{A}$, $\mathbf{A}'$ s.t. $\mathbf{A}|_{\mathbf{y}_{pred}} = \mathbf{A}'|_{\mathbf{y}_{pred}}$ where $\mathbf{y}_{pred}$ is the sublist of $\mathbf{y}$ consisting of all input parameters p_i at a position $1 \leq i \leq ar_{\text{I}}(\&g)$ with $type(\&g, i) = \mathbf{pred}$, i.e., only the extensions of predicates which occur explicitly as input term of type $\mathbf{pred}$ influence the value of $f_{\&g}(\cdot, \mathbf{x}, \mathbf{y})$.

When specifying the semantics of external atoms throughout this thesis, we will sometimes only explicitly define under which conditions it is true and implicitly assume that it is false otherwise; this will be stated as 'iff$_{\text{def}}$' expression, as demonstrated by the following example.

Example 4 (ctd.)**.** The semantics of the external predicate *&diff* from Examples 2 and 3 is given by the oracle function $f_{\&diff}$ defined such that for predicates p and q, $f_{\&diff}(\mathbf{A}, p, q, x) = 1$ iff$_{\text{def}}$ $x \in \mathbf{A}|_p$ and $x \notin \mathbf{A}|_q$ for all assignments $\mathbf{A}$, predicates p, q and constants x. Intuitively, *&diff* computes the set difference of the extensions of p and q in $\mathbf{A}$, i.e., the external atom is true for all constants x which are in the extension of p but not in that of q. The external predicate *&diff* is monotonic in the first parameter and nonmonotonic in the second. □

Definition 11. A *ground default-literal* of form $\text{not}\, a$ (where a can be a ground ordinary or a ground external atom), is true in assignment $\mathbf{A}$, denoted $\mathbf{A} \models \text{not}\, a$, if a is false in $\mathbf{A}$.

The notion of extension $ext(\cdot, \mathbf{A})$ for external predicates $\&g$ with input lists $\mathbf{y}$ is naturally defined as follows.

Definition 12 (Extension of an External Predicate with Input List)**.** The *extension of an external predicate* $\&g$ with input list $\mathbf{y}$ wrt. an assignment $\mathbf{A}$ is defined as $ext(\&g[\mathbf{y}], \mathbf{A}) = \{\mathbf{x} \mid f_{\&g}(\mathbf{A}, \mathbf{y}, \mathbf{x}) = 1\}$.

Satisfaction of ordinary rules and ASP programs [Gelfond and Lifschitz, 1991] is then extended to HEX-rules and programs in the obvious way:

Definition 13 (Satisfaction of Ground Rules)**.** A ground rule r of form

$$a_1 \vee \cdots \vee a_k \leftarrow b_1, \ldots, b_m, \text{not}\, b_{m+1}, \ldots, \text{not}\, b_n$$

is satisfied by $\mathbf{A}$, denoted $\mathbf{A} \models r$ if one of a_i for $1 \leq i \leq k$ is true in $\mathbf{A}$, or one of b_i for $1 \leq i \leq m$ is false in $\mathbf{A}$, or one of b_i for $m+1 \leq i \leq n$ is true in $\mathbf{A}$.

An assignment $\mathbf{A}$ is a *model* of a program Π, denoted $\mathbf{A} \models \Pi$, if $\mathbf{A} \models r$ for all $r \in \Pi$.

We can now define answer sets of HEX-programs similar to ordinary ASP programs, but using the *Faber-Leone-Pfeifer (FLP) reduct* [Faber et al., 2011] instead of the classical Gelfond-Lifschitz (GL-)reduct [Gelfond and Lifschitz, 1988]. The FLP-reduct and the GL-reduct are equivalent for ordinary ASP programs, but the former is superior for programs with aggregates as it eliminates unintuitive answer sets.

Definition 14 (FLP-Reduct [Faber et al., 2011])**.** For an interpretation $\mathbf{A}$ over a ground program Π, the *FLP-reduct* $f\Pi^{\mathbf{A}}$ of Π wrt. $\mathbf{A}$ is the set $\{r \in \Pi \mid \mathbf{A} \models b \text{ for all } b \in B(r)\}$ of all rules whose body is satisfied by $\mathbf{A}$.

In contrast to the FLP-reduct, which simply removes all rules with unsatisfied bodies, the *GL-reduct* $\Pi^{\mathbf{A}} = \{\bigvee_{a_i \in H(r)} a_i \leftarrow \bigwedge_{b_i \in B^+(r)} b_i \mid r \in \Pi, \mathbf{A} \models \text{not}\, b \text{ for all } b \in B^-(r)\}$ also removes the default-negated literals from the remaining rules (where a rule as by Definition 6 is written as $a_1 \vee \cdots \vee a_k \leftarrow b_1 \wedge \ldots \wedge b_m \wedge \text{not}\, b_{m+1} \wedge \ldots \wedge \text{not}\, b_n$).

Definition 15. Given assignments $\mathbf{A}_1$, $\mathbf{A}_2$ we say that $\mathbf{A}_1$ is *smaller than* $\mathbf{A}_2$, denoted $\mathbf{A}_1 \leq \mathbf{A}_2$, if $\{\mathbf{T}a \in \mathbf{A}_1\} \subseteq \{\mathbf{T}a \in \mathbf{A}_2\}$. We say that it is *strictly smaller than*, denoted $\mathbf{A}_1 < \mathbf{A}_2$, if $\{\mathbf{T}a \in \mathbf{A}_1\} \subsetneq \{\mathbf{T}a \in \mathbf{A}_2\}$.

Definition 16 (Answer Set)**.** An *answer set* of a ground program Π is a $\leq$-minimal model $\mathbf{A}$ of $f\Pi^{\mathbf{A}}$. An *answer set* of a program Π is an answer set of $grnd_{\mathcal{C}}(\Pi)$.

The set of all answer sets of a program Π is denoted $\mathcal{AS}(\Pi)$. We let $\equiv^{pos}$ denote equivalence of the answer sets of two programs Π and Π' in their positive parts, i.e., we write $\Pi \equiv^{pos} \Pi'$ if $\left\{\mathbf{A}^{\mathbf{T}} \mid \mathbf{A} \in \mathcal{AS}(\Pi)\right\} = \left\{\mathbf{A}^{\mathbf{T}} \mid \mathbf{A} \in \mathcal{AS}(\Pi')\right\}$.

Example 5 (ctd.)**.** Let Π be the HEX-program from Example 2. Then the answer sets are $\mathbf{A}_1 = \{\mathbf{T}sel(a), \mathbf{F}nsel(a)\}$ and $\mathbf{A}_2 = \{\mathbf{F}sel(a), \mathbf{T}nsel(a)\}$ because they are subset-minimal models (in the positive atoms) of

$$f\Pi^{\mathbf{A}_1} = \{sel(a) \leftarrow domain(a), \&diff[domain, nsel](a);\ domain(a) \leftarrow\}$$

and

$$f\Pi^{\mathbf{A}_2} = \{nsel(a) \leftarrow domain(a), \&diff[domain, sel](a);\ domain(a) \leftarrow\},$$

respectively. □

Note that for a non-ground program, answer sets possibly contain an infinite number of signed literals because the grounding is possibly infinite. However, in Chapter 4 we will identify criteria which ensure the existence of a finite grounding which preserves the finite positive part of answer sets.

To see why the FLP-reduct is preferable to the GL-reduct for HEX-programs, consider the following example.

Example 6. Let Π be the HEX-program Π

$$p(a) \leftarrow \text{not } \&not[p](a)$$
$$f \leftarrow \text{not } p(a), \text{not } f$$

where $f_{\&not}(\mathbf{A}, p, a) = 1$ iff$_{\text{def}}$ $a \notin ext(\mathbf{A}, p)$.

Then we have four answer set candidates $\mathbf{A}_1 = \{\mathbf{T}p(a), \mathbf{F}f\}$, $\mathbf{A}_2 = \{\mathbf{T}p(a), \mathbf{T}f\}$, $\mathbf{A}_3 = \{\mathbf{F}p(a), \mathbf{F}f\}$ and $\mathbf{A}_4 = \{\mathbf{F}p(a), \mathbf{T}f\}$. Under the GL-reduct, we have $\Pi^{\mathbf{A}_1} = \Pi^{\mathbf{A}_2} = \{p(a)\}$, $\Pi^{\mathbf{A}_3} = \{f\}$ and $\Pi^{\mathbf{A}_4} = \emptyset$. Then (the positive part of) $\mathbf{A}_1$ is the only model of $\Pi^{\mathbf{A}_1}$, i.e., it reproduces itself under the reduct and is thus a GL-answer set (while the other models do not reproduce themselves under the respective reduct).

However, it is not intuitive that $\mathbf{A}_1$ is an answer set because $p(a)$ supports itself. If we use the FLP-reduct instead, then we get $f\Pi^{\mathbf{A}_1} = \{p(a) \leftarrow \text{not } \&not[p](a)\}$. But now $\mathbf{A}_1$ is not a minimal model of $f\Pi^{\mathbf{A}_1}$ because $\mathbf{A}_3$ is also a model of $f\Pi^{\mathbf{A}_1}$ and $\mathbf{A}_3 < \mathbf{A}_1$. Thus, under the FLP-reduct all interpretations fail to be answer sets. □

This concludes our introduction of the syntax and semantics of HEX-programs. The class of (ordinary) ASP programs corresponds then simply to the class of HEX-programs without external atoms.

2.1.3 Atom Dependency Graph and Domain-Expansion Safety

To guarantee the existence of a finite grounding of a program which preserves the answer sets of the original program, we need additional safety criteria. The traditional safety criterion in ASP is recapitulated as follows.

Definition 17 (Safety)**.** A rule r is *safe*, if every variable in r occurs in an ordinary positive body atom $b \in B^+(r)$ or in the output list of an external body atom $b \in B^+(r)$ such that all variables in its input list are safe. A program is *safe* if all its rules are safe.

However, the usual notion of safety is not sufficient in presence of external sources as there exist safe programs which do not have a finite grounding which preserve all answer sets.

Example 7. Consider the following program $\Pi = \{s(a);\ s(Y) \leftarrow s(X), \&concat[X, a](Y)\}$, where for strings X, Y and C, $\&concat[X, Y](C)$ is true iff$_{\text{def}}$ C is the string concatenation of X and Y. Then this program is safe, but it does not have a finite grounding with the same answer sets as the original program because it derives infinitely many strings. □

Therefore the additional notion of *strong safety* was introduced by Eiter et al. (2006a) and further developed by Schüller (2012), which ensures that the output of cyclic external atoms is limited. For defining strong safety formally, we need the notion of *atom dependencies* in a program.

Definition 18 (External Atom Dependencies)**.** Let Π be a HEX-program with external atoms in different rules being standardized apart[2].

- If a is an external atom of form $\&g[X_1, \ldots, X_\ell](\mathbf{Y})$ in Π and $b = p(\mathbf{Z})$ is an atom in the head of a rule in Π. Then a *depends external monotonically (nonmonotonically) on* b, denoted $a \rightarrow^e_m b$ (resp. $a \rightarrow^e_n b$) if $X_i = p$ for some $1 \leq i \leq \ell$ and $type(\&g, i) = \mathbf{pred}$ is a monotonic (nonmonotonic) predicate parameter[3].

- If $\&g[X_1, \ldots, X_\ell](\mathbf{Y}), p(\mathbf{Z}) \in B^+(r)$ for some $r \in \Pi$ such that for some $1 \leq i \leq \ell$ we have $type(\&g, i) = \mathbf{const}$ and $X_i \in \mathbf{Z}$, then $\&g[X_1, \ldots, X_\ell](\mathbf{Y}) \rightarrow^e_m p(\mathbf{Z})$.

- If $\&g[X_1, \ldots, X_\ell](\mathbf{Y}), \&h[\mathbf{V}](\mathbf{U}) \in B^+(r)$ for some $r \in \Pi$ such that for some $1 \leq i \leq \ell$ we have $type(\&g, i) = \mathbf{const}$ and $X_i \in \mathbf{U}$, then $\&g[X_1, \ldots, X_\ell](\mathbf{Y}) \rightarrow^e_m \&h[\mathbf{V}](\mathbf{U})$.

We define $\rightarrow^e = \rightarrow^e_m \cup \rightarrow^e_n$. We further need the concept of *unification* of atoms.

Definition 19. An atom a *unifies* with with atom b, denoted $a \sim b$, if there exist mappings $\sigma_a\colon\ \mathcal{V} \rightarrow \mathcal{V} \cup \mathcal{C}$ and $\sigma_b\colon\ \mathcal{V} \rightarrow \mathcal{V} \cup \mathcal{C}$ such that $\sigma_a(a) = \sigma_b(b)$, where $\sigma_a(a)$ and $\sigma_b(b)$ denote the atoms constructed from a and b by replacing each variable X in a and Y in b by $\sigma_a(X)$ and $\sigma_b(Y)$, respectively.

[2]That is, we distinguish syntactically equal external atoms in different rules, e.g., by introducing new external predicates defined by the same oracle function.

[3]Note that Schüller (2012) did not use monotonicity of single predicate parameters but only monotonicity of all parameters. However, since the following concepts use only $\rightarrow^e_m \cup \rightarrow^e_n$ this definition is equivalent for our purposes.

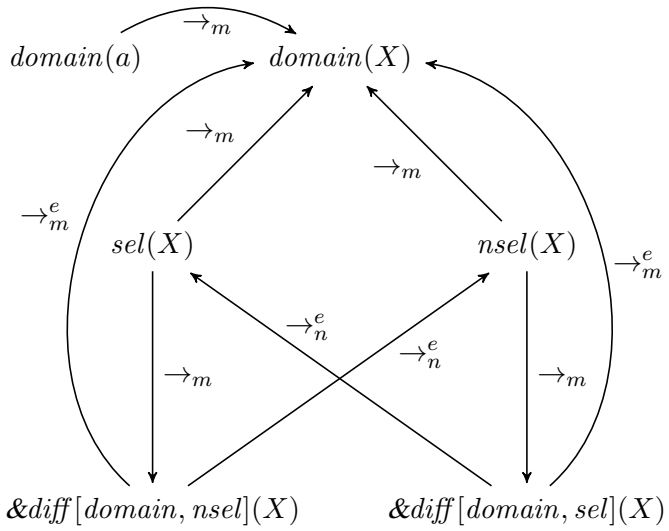


Figure 2.1: Atom Dependency Graph of the Program in Example 2

We then introduce atom dependencies as follows.

Definition 20 (Atom Dependencies)**.** For a HEX-program Π and atoms $a, b \in A(\Pi)$, we say:

(i) a *depends monotonically on* b, denoted $a \to_m b$, if

- some rule $r \in \Pi$ has $a \in H(r)$ and $b \in B^+(r)$; or
- there are rules $r_1, r_2 \in \Pi$ such that $a \in B^+(r_1) \cup B^-(r_1)$ and $b \in H(r_2)$ and $a \sim b$; or
- some rule $r \in \Pi$ has $a \in H(r)$ and $b \in H(r)$.

(ii) a *depends nonmonotonically on* b, denoted $a \to_n b$, if there is some rule $r \in \Pi$ such that $a \in H(r)$ and $b \in B^-(r)$.

Let $\to^+$ be the transitive closure of $\to = \to_m \cup \to_n \cup \to^e_m \cup \to^e_n$. We write $a \not\to b$ if $a \to b$ does not hold; similar for the other types of relations.

We can now introduce a graph which represents these kinds of dependencies in a program.

Definition 21 (Atom Dependency Graph)**.** For a HEX-program Π, the *atom dependency graph* $ADG(\Pi) = (V_A, E_A)$ of Π has as nodes $V_A = A(\Pi) \cup EA(\Pi)$ the (non-ground) ordinary and external atoms occurring in Π and as edges E_A the dependency relations $\to_m, \to_n, \to^e_m, \to^e_n$ between these atoms in Π, labeled with the type of the relation.

Example 8 (ctd.)**.** The atom dependency graph of the program in Example 2 is shown in Figure 2.1 and those of Example 7 is shown in Figure 2.2. □

This allows us to introduce strong safety as follows [Schüller, 2012].

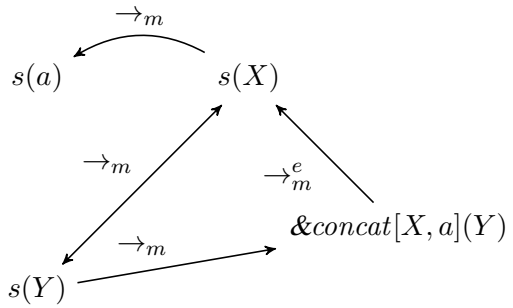


Figure 2.2: Atom Dependency Graph of the Programs in Example 7

Definition 22 (Strong Safety)**.** An external atom b in a rule r in a HEX-program Π is *strongly safe* wrt. r and Π if either

(i) $b \not\dashrightarrow^+ b$, i.e., there is no cyclic dependency over b; or

(ii) every variable in the output list of b occurs also in a positive ordinary atom $a \in B^+(r)$ such that a does not depend on b in $ADG(\Pi)$.

Example 9 (ctd.)**.** In the program in Example 7, $\&concat[X, a](Y)$ is not strongly safe because it occurs in a cycle and there is no ordinary body atom in the rule which binds Y and is not involved in the cycle. To make the program strongly safe we have to add a domain predicate as in $\Pi' = \{s(a); s(Y) \leftarrow s(X), \&concat[X, a](Y), limit(Y)\}$. □

These definitions can be used as follows to define *strong domain-expansion safety*. Note that Schindlauer (2006) and Schüller (2012) called this just *domain-expansion safety*. In this thesis we call it *strong*, because we will develop a more liberal notion of domain-expansion safety in Chapter 4, which gives the user more freedom when modelling a search problem in HEX.

Definition 23 (Strong Domain-Expansion Safety)**.** A HEX-program Π is *strongly domain-expansion safe*, if each rule in Π is safe and each external atom in a rule $r \in \Pi$ is strongly safe wrt. r and Π.

Tightly related to this notion of strong domain-expansion safety is that of *pre-groundable external atoms* and *pre-groundable* HEX*-programs* [Schüller, 2012].

Definition 24 (Pre-groundable External Atoms and Programs)**.** An external atom b in a rule r in a HEX-program Π is *pre-groundable wrt. r and Π* if for each variable X in the output list of b there exists a positive ordinary atom $a \in B^+(r)$ containing X such that $a \not\dashrightarrow^+ b$, i.e., a does not transitively depend on b. A HEX-program Π is *pre-groundable* if all external atoms in all rules $r \in \Pi$ are pre-groundable wrt. r and Π.

This exactly reflects Condition (ii) in Definition 22.

Example 10 (ctd.)**.** In the program Π' in Example 9, the external atom $\&concat[X, a](Y)$ is pre-groundable wrt. rule $s(Y) \leftarrow s(X), \&concat[X, a](Y), limit(Y)$ and Π' because Y occurs in $limit(Y)$, which does not transitively depend on $\&concat[X, a](Y)$. □

This notion was then extended to *extended pre-groundable* HEX*-programs* [Schüller, 2012]. Intuitively, extended pre-groundable HEX-programs may contain external atoms which are not pre-groundable, but then they do depend on facts but not on any other rules. Such external atoms will sometimes be called *outer external atoms*. The truth values of outer external atoms can be deterministically computed once the facts of the program are fixed.

Definition 25 (Extended Pre-groundable Programs)**.** A HEX-program is *extended pre-groundable* if for each external atom b in a rule $r \in \Pi$ it holds that either b is pre-groundable wrt. r and Π, or every atom a that b depends on is the head of a non-disjunctive fact in Π. That is, if atom a occurs in a rule head in Π, this rule must be of the form $a \leftarrow$.

Example 11. In the program $\Pi = \{q(a);\ p(X) \leftarrow \&id[q](X)\}$ the external atom $\&id[p](X)$ is not pre-groundable but the program is still extended pre-groundable because $\&id[p](X)$ depends only on the fact $q(a)$. □

It is easy to see that an external atom in a strongly domain-expansion safe program, which is not pre-groundable, does not cyclically depend on itself (Proposition 12 by Schüller (2012)). This is because of Condition (i) in Definition 22, which must hold if Condition (ii) does not hold. This property will be exploited by the *model-building framework* (which is formally introduced in Chapter 4) as follows. The overall program is split into fragments, called *(evaluation) units*, such that every unit is extended pre-groundable. Each component can then be evaluated by an algorithm which grounds the whole unit prior to evaluation. Intuitively, this means that value invention occurs only between but not within evaluation units.

Extended pre-groundable HEX-programs can be evaluated by Algorithm EvaluateExtended-PreGroundable, which computes *the positive parts of* all answer sets of a program, augmented with facts from some input interpretation [Schüller, 2012]. The basic idea is to first evaluate all external atoms which are not pre-groundable. As they depend only on facts, their input list must be ground, because if there would be a variable in the input list, then the external atom would depend on at least one non-ground atom. But then they can be immediately evaluated as soon as the input interpretation $\mathbf{A}$ is known, thus also the truth values for all ground instances of the external atom are fixed. An auxiliary fact of form $e_{\&g[\mathbf{y}]}(\mathbf{z})$ is added for each (ground) output tuple $\mathbf{z} \in ext(\mathbf{A}, \&g[\mathbf{y}])$ of the external atom with input list $\&g[\mathbf{y}]$, which unifies ($\sim$) with the output list $\mathbf{X}$ given in the program. Then each external atom $\&g[\mathbf{Y}](\mathbf{X}) \in E_{outer}$ is replaced by its auxiliary atom $e_{\&g[\mathbf{Y}]}(\mathbf{X})$. The resulting program Π' is now pre-groundable because all external atoms which were not pre-groundable have been resolved. Thus, the program can now be grounded similar to ordinary ASP programs using the procedure, but taking variables in the input to external atoms into account, as discussed in the next subsection; we abstractly use the resulting grounding algorithm as GroundProgram. We do not discuss the grounding procedure in detail because we will develop a strictly more general approach in Chapter 4. The resulting ground program is then given to procedure EvalGroundHexProgram(Π'_{grnd}), which computes the answer sets of the input ground HEX-program Π'_{grnd}.

At this point we do not discuss ground model building, i.e. the implementation of the procedure EvalGroundHexProgram. The development of appropriate algorithms are one of the main goals of this thesis and will we shown them in detail in Chapter 3. However, the traditional algorithm sketched by Schindlauer (2006) works roughly as follows. Each (ground) external atom is replaced by an auxiliary atom and its truth value is guessed. The resulting ordinary ASP program is then solved by a state-of-the-art ASP solver to produce answer set candidates. Each candidate is then checked for compliance with the external sources. If all guesses coincide with the semantics of external atoms, then minimality of the candidate wrt. the FLP-reduct is checked. If also this check is passed, an answer set has been found.

Algorithm EvaluateExtendedPreGroundable

Input: A HEX-program Π, an input interpretation $\mathbf{A}$
Output: Positive parts of all answer sets of $\Pi \cup \{a \leftarrow . \mid \mathbf{T}a \in \mathbf{A}\}$ without $\mathbf{A}$

`// determine non-disjunctive facts in` Π `and add them to` $\mathbf{A}$
$\mathbf{A}' \leftarrow \mathbf{A} \cup \{\mathbf{T}a \mid r \in \Pi$ such that $H(r) = \{a\}$ and $B(r) = \emptyset\}$
`// determine external atoms that get input only from` $\mathbf{A}'$
$E_{outer} \leftarrow \{\&g[\mathbf{X}](\mathbf{Y})$ in $\Pi \mid$ if $\&g[\mathbf{X}](\mathbf{Y}) \rightarrow^e b$ then $\mathbf{T}b \in \mathbf{A}'\}$
`// evaluate external atoms and create corresponding`
`// ground replacement atoms`
$\mathbf{A}_{aux} \leftarrow \{\mathbf{T}e_{\&g[\mathbf{y}]}(\mathbf{z}) \mid \&g[\mathbf{y}](\mathbf{X}) \in E_{outer}, \mathbf{z} \in ext(\mathbf{A}', \&g[\mathbf{y}]), \mathbf{z} \sim \mathbf{X}\}$
`// introduce auxiliary atoms for outer external atoms`
$\Pi' \leftarrow \Pi$ with external atoms $\&g[\mathbf{Y}](\mathbf{X}) \in E_{outer}$ replaced by auxiliaries $e_{\&g[\mathbf{Y}]}(\mathbf{X})$
`// add input and auxiliaries as facts`
$\Pi' \leftarrow \Pi' \cup (a \leftarrow . \mid \mathbf{T}a \in \mathbf{A} \cup \mathbf{A}_{aux})$
`// ground the program`
$\Pi'_{grnd} \leftarrow$ GroundProgram(Π')
`// ground program evaluation and output projection`
return $\{\mathbf{A}'' \setminus (\mathbf{A} \cup \mathbf{A}_{aux} \cup \{\mathbf{F}a \in \mathbf{A}''\}) \mid \mathbf{A}'' \in$ EvalGroundHexProgram$(\Pi'_{grnd})\}$

2.1.4 External Atom Input Grounding

Following Schindlauer (2006) and Schüller (2012), we use the following precedure for grounding the input of external atoms. We create for each external atom in the program an auxiliary rule which computes the relevant substitutions for all variables in its input list. Then we evaluate the set of all auxiliary rules and retrieve the tuples we need to substitute for the variables.

The following definition corresponds to Definition 4.6.11 by Schindlauer (2006) and Definition 23 by Schüller (2012). However, we call the following concept *basic* input auxiliary rule, because we make use of a slightly adopted notion of input auxiliary rules in Chapter 4.

Definition 26 (Basic Input Auxiliary Rule)**.** Let Π be a HEX-program, and let $\&g[\mathbf{Y}](\mathbf{X})$ be some external atom with input list $\mathbf{Y}$ occurring in a rule $r \in \Pi$. Then, for each such atom, a rule $r^{\&g}_{inp}$ is composed as follows:

- The head $H\big(r_{inp}^{\&g}\big)$ contains an atom $g_{inp}^{\&g}(\mathbf{Y})$ with a fresh predicate $g_{inp}^{\&g}$.
- The body $B\big(r_{inp}^{\&g}\big)$ of the auxiliary rule contains all body literals of r other than $\&g[\mathbf{Y}](\mathbf{X})$ that have at least one variable in its arguments (resp. in its output list if b is another external atom) that occurs also in $\mathbf{Y}$.

For each external atom in Π we create such a rule and denote the resulting set of rules Π_{inp}.

Example 12 (adopted from Example 43 by Schüller (2012))**.** Consider the non-ground HEX-program

$$\Pi = \{out(Y) \leftarrow \&concat[a,b](X), \&concat[X,c](Y)\}.$$

Then the program Π_{inp} consists of the single rule $g_{inp}^{\&concat}(X) \leftarrow \&concat[a,b](X)$. If we evaluate Π_{inp} we get the single answer set $\{\mathbf{T}g_{inp}^{\&concat}(ab)\}$, which is used for grounding Π to $\Pi_{grnd} = \{out(Y) \leftarrow \&concat[a,b](ab), \&concat[ab,c](Y)\}$. Then we can evaluate Π_{grnd} and get the single answer set $\{\mathbf{T}out(abc)\}$. □

The process of creating auxiliary rules needs to be iterated in general. For instance, for the program $\Pi = \{out(Z) \leftarrow \&concat[a,b](X), \&concat[X,c](Y), \&concat[Y,d](Z)\}$, we first compute the input to $\&concat[X,c](Y)$, which is ab, and then the input to $\&concat[Y,d](Z)$, which is abc.

2.1.5 Modular Evaluation of HEX-Programs

The evaluation of HEX-programs is based on a *model-building framework* [Eiter et al., 2011a; Schüller, 2012], which splits the non-ground program into smaller program components, each of which is extended pre-groundable.

At this point we do not recapitulate the details of the model-building framework because they are not necessary for the general understanding of HEX-programs and for the development of algorithms for ground HEX-program evaluation in Chapter 3. We rather delay the introduction to Chapter 4. Here we only note that the decomposition is done for two reasons. First, this may increase efficiency, as observed by Schüller (2012), and second, the decomposition is sometimes even *necessary* because the actual evaluation in Algorithm EvaluateExtendedPreGroundable can only handle extended pre-groundable HEX-programs. Thus, if the input program is not extended pre-groundable, then the framework must split it such that each unit becomes extended pre-groundable. It was shown by Schüller (2012) that such a splitting exists for every strongly domain-expansion safe program. In Chapter 4 we will develop more advanced algorithms which can handle a larger class of programs directly. This gives the framework more freedom in the decision whether units are split or not.

2.2 Conflict-Driven Learning and Nonchronological Backtracking

In this section we describe the basic algorithm of conflict-driven SAT solvers. *Conflict-driven clause learning (CDCL)* for SAT was first introduced by Mitchell (2005) and has turned out to be very efficient for practical applications. Later, the approach has been adopted to conflict-driven

ASP solving [Gebser et al., 2007a] and to disjunctive ASP solving [Drescher et al., 2008]. Algorithms based on conflict-driven techniques also fit into the framework for *abstract ASP solving*, which formalize reasoning as a state transition system [Lierler, 2011]. That is, unit propagation, learning and forgetting (see below) are described in terms of changes which are made to the clause set and to the assignment.

We will use *nogoods* instead of classical clauses, following Drescher et al. (2008). The approach is therefore referred to as *conflict-driven nogood learning (CDNL)*.

Definition 27 (Nogood)**.** A *nogood* $\{L_1, \ldots, L_n\}$ is a set of (signed) literals $L_i, 1 \leq i \leq n$.

Note that every classical clause can be transformed into an equivalent nogood and vice versa by negating all literals.

Definition 28. An assignment $\mathbf{A}$ is a *solution* to a nogood δ (resp. a set of nogoods Δ), if $\delta \not\subseteq \mathbf{A}$ (resp. $\delta \not\subseteq \mathbf{A}$ for all $\delta \in \Delta$).

If for a nogood δ (resp. a set of nogoods Δ) we have $\delta \subseteq \mathbf{A}$ (resp. $\delta \subseteq \mathbf{A}$ for some $\delta \in \Delta$), we say that $\mathbf{A}$ *violates* nogood δ (resp. set of nogoods Δ).

DPLL-style SAT solvers rely on an alternation of drawing deterministic consequences and guessing the truth value of an atom towards a complete interpretation [Davis et al., 1962]. Deterministic consequences are drawn by the basic operation of *unit propagation*, i.e., whenever all but one signed literals of a nogood are true, the last one must be false. The solver stores an integer *decision level* dl, written $@dl$ as postfix to the signed literal. An atom which is set by unit propagation using nogood δ gets the highest decision level of all already assigned atoms in δ, whereas guessing increments the current decision level.

Most modern SAT solvers are *conflict-driven*, i.e., they learn additional nogoods when the current assignment violates a nogood. Practical systems also implement heuristics for removal of learned nogoods, which is called *forgetting* and prevents the reasoner from running into memory problems, while learning prevents the solver from running into the same conflict again. The learned nogood is determined by initially setting the conflict nogood to the violated one. As long as it contains multiple literals on the same decision level, it is resolved with the *reason* of one of these literals, i.e., the nogood which implied it[4]. This strategy is referred to as *first unique implication point* or *first UIP* [Marques-Silva and Sakallah, 1999].

Example 13. Consider the nogoods

$$\{\mathbf{T}a, \mathbf{T}b\}, \{\mathbf{T}a, \mathbf{T}c\}, \{\mathbf{F}a, \mathbf{T}x, \mathbf{T}y\}, \{\mathbf{F}a, \mathbf{T}x, \mathbf{F}y\}, \{\mathbf{F}a, \mathbf{F}x, \mathbf{T}y\}, \{\mathbf{F}a, \mathbf{F}x, \mathbf{F}y\}$$

and suppose the assignment is $\mathbf{A} = \{\mathbf{F}a@1, \mathbf{T}b@2, \mathbf{T}c@3, \mathbf{T}x@4\}$. Then the third nogood is unit and implies $\mathbf{F}y@4$, which violates the fourth nogood $\{\mathbf{F}a, \mathbf{T}x, \mathbf{F}y\}$. As it contains multiple literals ($\mathbf{T}x$ and $\mathbf{F}y$) which were set at decision level 4, it is resolved with the reason for setting y to false, which is the nogood $\{\mathbf{F}a, \mathbf{T}x, \mathbf{T}y\}$. This results in the nogood $\{\mathbf{F}a, \mathbf{T}x\}$, which contains the single literal $\mathbf{T}x$ set at decision level 4, and thus is the learned nogood.

[4]If one sees nogoods as conjunctions of literals which imply falsity, this amounts to a resolution of implicants which is known as *rule of consensus*, cf. Brown (2003).

In standard clause notation, the nogood set corresponds to

$$(\neg a \vee \neg b) \wedge (\neg a \vee \neg c) \wedge (a \vee \neg x \vee \neg y) \wedge (a \vee \neg x \vee y) \wedge (a \vee x \vee \neg y) \wedge (a \vee x \vee y)$$

and the violated clause is $(a \vee \neg x \vee y)$. It is resolved with $(a \vee \neg x \vee \neg y)$ and results in the learned clause $(a \vee \neg x)$. □

State-of-the-art SAT and ASP solvers backtrack then to the second-highest decision level in the learned nogood. In Example 13, this is decision level 1. All assignments after decision level 1 are undone ($\mathbf{T}b@2$, $\mathbf{T}c@3$, $\mathbf{T}x@4$). Only variable $\mathbf{F}a@1$ remains assigned. This makes the new nogood $\{\mathbf{F}a, \mathbf{T}x\}$ unit and derives $\mathbf{F}x$ at decision level 1.

In contrast, the classical DPLL algorithm without learning would only undo the last decision level 4 and try the alternative guess $\mathbf{F}x@4$, which would produce another related conflict.

2.3 Conflict-Driven ASP Solving

In this subsection we summarize conflict-driven answer-set solving and disjunctive answer-set solving as described by Gebser et al. (2012) and Drescher et al. (2008). The fundamental algorithm, which will be used as foundation for our techniques developed in Chapter 3, is shown in Algorithm DASP-CDNL. It corresponds to Algorithm Hex-CDNL in Chapter 3 without Parts (c) and (d). To employ conflict-driven techniques from SAT solving in ASP, programs are represented as sets of nogoods. For a ground ASP program let $BA(\Pi) = \big\{\{B(r)\} \mid r \in \Pi\big\}$ be the set of all rule bodies of Π, where each body is viewed as a fresh atom, which are later projected from the interpretations.

We first define the set

$$\gamma(C) = \big\{\{\mathbf{F}C\} \cup \{\mathbf{t}\ell \mid \ell \in C\}\big\} \cup \big\{\{\mathbf{T}C, \mathbf{f}\ell\} \mid \ell \in C\big\}$$

of nogoods to encode that a set C of default literals must be assigned $\mathbf{T}$ or $\mathbf{F}$ in terms of the conjunction of its elements, where $\mathbf{t}\,\mathrm{not}\,a = \mathbf{F}a$, $\mathbf{t}a = \mathbf{T}a$, $\mathbf{f}\,\mathrm{not}\,a = \mathbf{T}a$, and $\mathbf{f}a = \mathbf{F}a$. That is, the conjunction is true iff each literal is true.

Example 14. For the set $C = \{a, \mathrm{not}\,b\}$, which represents the conjunction of a and $\mathrm{not}\,b$, we have $\gamma(C) = \big\{\{\mathbf{F}C, \mathbf{T}a, \mathbf{F}b\}, \{\mathbf{T}C, \mathbf{F}a\}, \{\mathbf{T}C, \mathbf{T}b\}\big\}$. □

Clark's completion Δ_Π of a program Π over atoms $A(\Pi) \cup BA(\Pi)$ amounts then to the following set of nogoods [Clark, 1977]:

$$\Delta_\Pi = \bigcup\nolimits_{r \in \Pi} \gamma\big(B(r)\big) \cup \big\{\{\mathbf{T}B(r)\} \cup \{\mathbf{F}a \mid a \in H(r)\}\big\}$$

It encodes that the body of a rule is true iff each literal is true, and if the body is true, at least one head atom must also be true. Unless a program is in the class of *tight programs* [Fages, 1994][5], Clark's completion does not fully capture the semantics of a program as unfounded sets may

[5] Without going into detail, tightness is a syntactic condition hinging on positive atom dependencies in the program and is defined using level mappings.

occur, i.e., sets of atoms which only cyclically support each other, also called a *loop*. We will formally introduce unfounded sets in a more general fashion in Section 3.2 and give an intuitive explanation for now.

Example 15. Consider $\Pi = \{a \leftarrow b;\ b \leftarrow a\}$. Then Clark's completion

$$\Delta_\Pi = \big\{\{\mathbf{F}\{b\}, \mathbf{T}b\}, \{\mathbf{T}\{b\}, \mathbf{F}b\}, \{\mathbf{T}\{b\}, \mathbf{F}a\}, \{\mathbf{F}\{a\}, \mathbf{T}a\}, \{\mathbf{T}\{a\}, \mathbf{F}a\}, \{\mathbf{T}\{a\}, \mathbf{F}b\}\big\}$$

has the solution $\mathbf{A} = \{\mathbf{T}a, \mathbf{T}b, \mathbf{T}\{a\}, \mathbf{T}\{b\}\}$, but the projection of $\mathbf{A}$ to signed literals over atoms from $A(\Pi)$, $\mathbf{A} \cap \{\mathbf{T}a, \mathbf{F}a \mid a \in A(\Pi)\} = \{\mathbf{T}a, \mathbf{T}b\}$, it is not an answer set of Π because a and b support each other only cyclically. □

Avoidance of unfounded sets requires additional *loop nogoods*, but as there are exponentially many, they are only introduced on-the-fly (see below).

It is common for disjunctive programs to introduce additional nogoods $\Theta_{sh(\Pi)}$ which regulate support of singletons. They are based on a transformation $sh(\Pi)$ of the programm, called *shifted program*. This allows for a more efficient implementation of UnfoundedSet because the procedure can safely ignore head-cycle free program components. However, as this is not relevant for the understanding of the overall algorithm and concerns only the ordinary ASP solver but not the work in this thesis, we abstractly use them as $\Theta_{sh(\Pi)}$; for an exhaustive description we refer to Drescher et al. (2008).

With these concepts we are ready to describe the basic algorithm for answer set computation, which is shown in Algorithm DASP-CDNL. The algorithm keeps a set $\Delta_\Pi \cup \Theta_{sh(\Pi)}$ of *static* nogoods (from Clark's completion and from the shifted program), and a set ∇ of *dynamic* nogoods which are learned from conflicts and unfounded sets. During construction of the assignment $\mathbf{A}$, the algorithm stores for each atom $a \in A(\Pi) \cup BA(\Pi) \cup BA(sh(\Pi))$ a *decision level* dl. The decision level is initially 0 and incremented for each choice. Deterministic consequences of a set of assigned values have the same decision level as the highest decision level in this set.

The main loop iteratively derives deterministic consequences using Propagation in Part (a) trying to complete the assignment. This includes both unit propagation and unfounded set propagation. Unit propagation derives $\overline{d}$ if $\delta \setminus \{d\} \subseteq \mathbf{A}$ for some nogood δ, i.e. all but one literal of a nogood are true, therefore the last one needs to be false. Unfounded set propagation detects atoms which only cyclically support each other and falsifies them. For instance, in Example 15, unfounded set propagation would immediately set one of a or b to false, and unit propagation sets subsequently also the other one to false.

Part (b) checks if there is a conflict, i.e. a violated nogood $\delta \subseteq \mathbf{A}$. If this is the case the algorithm needs to backtrack. For this purpose, the call to Analysis computes a learned nogood ϵ and a backtrack decision level k. The learned nogood is added to the set of dynamic nogoods, and assignments above decision level k are undone. Otherwise, Part (c) checks if the assignment is complete. In this case, a final unfounded set check is necessary due to disjunctive heads. If the candidate is founded, i.e., no unfounded set exists, then it is an answer set. Otherwise, a violated loop nogood δ from the set

$$\lambda_\Pi(U) = \left\{ \{\sigma_1, \ldots, \sigma_m\} \mid (\sigma_1, \ldots, \sigma_m) \in \{\mathbf{T}a \mid a \in U\} \times \prod_{r \in sup_\Pi(U)} sat_r(U) \right\}$$

of all loop nogoods for an unfounded set U is selected, where

$$sat_r(U) = \{\mathbf{F}\{B(r)\}\} \cup \{\mathbf{T}a \mid a \in H(r) \setminus U\}$$

is the set of all signed literals which satisfy r independently of U and

$$sup_\Pi(U) = \{r \in \Pi \mid H(r) \cap U \neq \emptyset, B(r) \cap U = \emptyset\}$$

is the set of rules which may be used to derive one of U without depending on U. Intuitively, the nogoods in $\lambda_\Pi(U)$ encode that it must never happen, that an atom in the unfounded set is true, but each rule which supports this atom is already satisfied independently of U (because then the rule cannot be used to justify this atom being true). After adding such a nogood, conflict analysis and backtracking is carried out. If no more deterministic consequences can be derived and the assignment is still incomplete, some truth value is guessed in Part (d) and the decision level is incremented. The function Select implements a variable selection heuristics. In the simplest case it chooses an arbitrary signed literal σ over a yet unassigned variable, but state-of-the-art heuristics are more sophisticated. E.g., Goldberg and Novikov (2007) prefer variables which are involved in recent conflicts. We will explain parts of the algorithm in more detail whenever this becomes necessary throughout this thesis.

Note that the algorithm uses two variants of unfounded set detection. One is implemented in the procedure Propagation and runs *a priori*, i.e., possible unfounded sets are already detected before they have become manifest in the assignment. This form of unfounded set check runs in polynomial time but cannot detect all kinds of unfounded sets. In contrast, the unfounded set check UnfoundedSet in Part (c) runs *a posteriori*, i.e., after the assignment has been completed and possibly contains the unfounded set. This unfounded set detection is only necessary due to head-cycles in disjunctive programs and is co-NP-complete itself. The procedure UnfoundedSet may be implemented as a SAT search problem itself, as described by Drescher et al. (2008). We will make use of a related but more general approach in Section 3.2.

2.4 Complexity

We assume familiarity with basic concepts of complexity theory, e.g., Turing machines, complexity classes and reductions; for details we refer to Papadimitriou (1994). We denote by P and NP the classes of *decision problems* (i.e., computational problems with yes/no answer) which can be solved in polynomial time by deterministic and nondeterministic Turing machines, respectively. For a complexity class C, class co-C contains problems whose complement language is in C. The *polynomial hierarchy (PH)* is a hierarchy of complexity classes Σ_k^P, Π_k^P, Δ_k^P, defined by $\Sigma_0^P = \Pi_0^P = \Delta_0^P = \mathsf{P}$, and for $k \geq 1$, we have $\Sigma_k^P = \mathsf{NP}^{\Sigma_{k-1}^P}$ and $\Pi_k^P = \text{co-}\Sigma_k^P$ and $\Delta_k^P = \mathsf{P}^{\Sigma_{k-1}^P}$. By P^O (NP^O) we denote the class of problems which can be solved by a deterministic (nondeterministic) Turing machine in polynomial time if equipped with an *oracle* for complexity class O, i.e., problems in O can be solved in one step. In particular, $\Sigma_1^P = \mathsf{NP}$ and $\Pi_1^P = \text{co-NP}$, which will be the most relevant complexity classes throughout this thesis. We further have $\mathsf{PH} = \bigcup_{k \geq 0} \Sigma_k^P$. Classes PSPACE resp. NPSPACE contain the decision problems solvable by deterministic resp. nondeterministic Turing machines with polynomial space.

Algorithm DASP-CDNL

Input: A ground ASP program Π
Output: An answer set of Π, or $\bot$ if none exists

$\mathbf{A} \leftarrow \emptyset$ // assignment over $A(\Pi) \cup BA(\Pi) \cup BA\big(sh(\Pi)\big)$
$\nabla \leftarrow \emptyset$ // dynamic nogoods
$dl \leftarrow 0$ // decision level
while *true* **do**
(a) $(\mathbf{A}, \nabla) \leftarrow$ Propagation$(\Pi, \nabla, \mathbf{A})$
(b) **if** $\delta \subseteq \mathbf{A}$ *for some* $\delta \in \Delta_\Pi \cup \Theta_{sh(\Pi)} \cup \nabla$ **then**
  if $dl = 0$ **then return** $\bot$
  $(\epsilon, k) \leftarrow$ Analysis$(\delta, \Pi, \nabla, \mathbf{A})$
  $\nabla \leftarrow \nabla \cup \{\epsilon\}$
  $\mathbf{A} \leftarrow \mathbf{A} \setminus \big\{\sigma \in \mathbf{A} \mid k < dl(\sigma)\big\}$
  $dl \leftarrow k$
(c) **else if** $\mathbf{A^T} \cup \mathbf{A^F} = A(\Pi) \cup BA(\Pi) \cup BA\big(sh(\Pi)\big)$ **then**
  $U \leftarrow$ UnfoundedSet$(\Pi, \mathbf{A})$
  if $U \neq \emptyset$ **then**
    Let $\delta \in \lambda_\Pi(U)$ such that $\delta \subseteq \mathbf{A}$
    if $\big\{\sigma \in \delta \mid 0 < dl(\sigma)\big\} = \emptyset$ **then return** $\bot$
    $(\epsilon, k) \leftarrow$ Analysis$(\delta, \Pi, \nabla, \mathbf{A})$
    $\nabla \leftarrow \nabla \cup \{\epsilon\}$
    $\mathbf{A} \leftarrow \mathbf{A} \setminus \big\{\sigma \in \mathbf{A} \mid k < dl(\sigma)\big\}$
    $dl \leftarrow k$
  else
    return $\mathbf{A} \cap \big\{\mathbf{T}a, \mathbf{F}a \mid a \in A(\Pi)\big\}$
(d) **else**
  $\sigma \leftarrow$ Select$(\Pi, \nabla, \mathbf{A})$
  $dl \leftarrow dl + 1$
  $\mathbf{A} \leftarrow \mathbf{A} \cup \{\sigma\}$

The complexity classes EXPTIME resp. NEXPTIME contain the decision problems solvable by deterministic resp. nondeterministic Turing machines in exponential time. It is known that

$$\mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{PH} \subseteq \mathsf{PSPACE} = \mathsf{NPSPACE} \subseteq \mathsf{EXPTIME}.$$

Since $\mathsf{P} \subsetneq \mathsf{EXPTIME}$, at least one of the inclusions must be strict, but it is widely believed that all of them are strict, although not proven. It is also believed that the classes Σ_k^P, Π_k^P for $k \geq 0$ form a true hierarchy of infinitely many levels.

Chapter 3

Propositional HEX-Program Solving

In this chapter we present genuine evaluation algorithms for ground HEX-programs. The idea is related to conflict-driven disjunctive ASP solving [Drescher et al., 2008], but strictly more general as it integrates additional novel learning techniques to capture also HEX-programs with external atoms. The term *learning* refers to the process of adding further nogoods to the nogood set, which represents the program at hand, during exploration of the search space.

While additional nogoods are traditionally derived from conflict situations in order to avoid similar conflicts during further search, we add a second type of learning which captures the behavior of external sources, called *external behavior learning (EBL)*. Whenever an external atom is evaluated, the algorithm might learn from the respective call. If we have no further information about the internals of a source, we may learn only very general input-output relationships, but if we have more information, we can learn more effective nogoods. In general, we will associate a *learning-function* with each external source which tells the system which nogoods to learn. This learning function may be derived automatically from known properties of external sources (such as monotonicity or functionality), but can also be overridden by the user in order to give hints to the system why some tuple is in the output or why it is not in the output of an external source.

All programs in this chapter are assumed to be ground. The algorithms introduced in this chapter are intended to be used in place of EvalGroundHexProgram in Algorithm Evaluate-ExtendedPreGroundable in Chapter 2. In particular, we will introduce two Algorithms GuessAndCheckHexEvaluation and WellfoundedHexEvaluation. The Algorithm GuessAndCheck-HexEvaluation may be applied to any ground HEX-program. Intuitively, the algorithm guesses the truth values of all ground external atoms in the program, employs an ordinary ASP solver to generate model candidates, and verifies then the guesses; but in contrast to the traditional algorithm, it learns during this process in order to guide the algorithm. Instead of guessing, Algorithm WellfoundedHexEvaluation performs a fixpoint iteration. This is usually faster, but can be applied only to purely monotonic programs.

We start our discussion with the algorithm GuessAndCheckHexEvaluation. We abstractly make use of a set of nogoods learned from the evaluation of some external predicate with input list $\&g[\mathbf{y}]$ wrt. assignment $\mathbf{A}$. This set is specified by a *learning function*, denoted $\Lambda(\&g[\mathbf{y}], \mathbf{A})$. Based on this concept we then introduce the evaluation algorithm for HEX-programs. At this point we use the *minimality checker* as a black box. Minimality checking of candidate answer sets is an important issue because unfounded sets may involve the semantics of external sources and are thus tricky to detect in some cases. However, we separate the construction of candidate answer sets from the minimality check to make the presentation simpler.

Section 3.1.2 will then provide definitions of particular nogoods that can be learned for various types of external sources, i.e., $\Lambda(\cdot, \cdot)$ is instantiated. The learning functions are automatically derived from known properties of external sources, which are asserted by the user.

In Section 3.2 we introduce concrete algorithms for the final minimality check of candidate answer sets. We show that there is a rather straightforward approach which explicitly searches for models smaller than the current candidate. But since this method does not scale well in practice, we then present an advanced algorithm based on unfounded sets. Our approach is related to those of Drescher et al. (2008) but strictly more general because it also respects external sources. We further provide a syntactic decision criterion, which allows for skipping the whole minimality check or restricting it to relevant program components in some practically relevant cases.

Finally, in Section 3.3 we present the Algorithm WellfoundedHexEvaluation which is applicable to a fragment of HEX, called *monotonic* HEX*-programs*, and is more efficient in many cases.

3.1 Guess and Check Algorithm for General Ground HEX-Programs

We assume that we have an arbitrary ground HEX-program Π and want to compute all its answer sets $\mathcal{AS}(\Pi)$. Schindlauer (2006) and Schüller (2012) proposed algorithms which determine the answer sets of a HEX-program Π using a transformation to ordinary ASP programs as follows. Each external atom $a = \&g[\mathbf{y}](\mathbf{x})$ in a rule $r \in \Pi$ is replaced by an ordinary ground *(external) replacement atom* $\hat{a} = e_{\&g[\mathbf{y}]}(\mathbf{x})$ (resulting in a rule $\hat{r}$), and an additional rule $e_{\&g[\mathbf{y}]}(\mathbf{x}) \vee ne_{\&g[\mathbf{y}]}(\mathbf{x}) \leftarrow$ is added to the program. The answer sets of the resulting *guessing program* $\hat{\Pi}$ are determined by an ordinary ASP solver and projected to non-replacement atoms. However, the resulting assignments are not necessarily models of Π, as the values of $\&g[\mathbf{y}]$ and $e_{\&g[\mathbf{y}]}(\mathbf{x})$ relative to an interpretation may not coincide. Each answer set of $\hat{\Pi}$ is thus a *candidate compatible set* (or *model candidate*) which must be checked against the external sources. If no discrepancy is found, the model candidate is a *compatible set* of Π. More precisely,

Definition 29 (Guessing Program)**.** For a ground HEX-program Π, let $\hat{\Pi}$ be the *guessing program* where for each external atom $\&g[\mathbf{y}](\mathbf{x})$

- $\&g[\mathbf{y}](\mathbf{x})$ in a rule $r \in \Pi$ is replaced by an ordinary ground (external) replacement atom $e_{\&g[\mathbf{y}]}(\mathbf{x})$ (resulting in a rule $\hat{r}$); and

- a rule $e_{\&g[\mathbf{y}]}(\mathbf{x}) \vee ne_{\&g[\mathbf{y}]}(\mathbf{x}) \leftarrow$ is added to the program.

For an external atom $\&g[\mathbf{y}](\mathbf{x})$, the rule $e_{\&g[\mathbf{y}]}(\mathbf{x}) \vee ne_{\&g[\mathbf{y}]}(\mathbf{x}) \leftarrow$ is called *ground external atom guessing rule*. In Chapter 4 we will also make use of *non-ground external atom guessing rules*. Since in this chapter we always mean ground external atom guessing rules, we drop the prefix *ground*.

Definition 30 (Compatible Set)**.** A *compatible set* of a program Π is an assignment $\hat{\mathbf{A}}$

(i) which is an answer set [Gelfond and Lifschitz, 1991] of the *guessing program* $\hat{\Pi}$[1]; and

(ii) $f_{\&g}(\hat{\mathbf{A}}, \mathbf{y}, \mathbf{x}) = 1$ if $\mathbf{T}e_{\&g[\mathbf{y}]}(\mathbf{x}) \in \hat{\mathbf{A}}$ and $f_{\&g}(\hat{\mathbf{A}}, \mathbf{y}, \mathbf{x}) = 0$ otherwise for all external atoms $\&g[\mathbf{y}](\mathbf{x})$ in Π, i.e. the guessed values coincide with the values of the oracle functions.

Note that for each external atom a in a program Π, a compatible set $\hat{\mathbf{A}}$ must assign exactly one of e_a and ne_a to true.

Proposition 3.1. *Let $\hat{\mathbf{A}}$ be a compatible set of a program Π. Then for each external atom a in Π we have $\left|\{\mathbf{T}e_a, \mathbf{T}ne_a\} \cap \hat{\mathbf{A}}\right| = 1$.*

Proof. Because of the guessing rules in $\hat{\Pi}$, at least one of e_a and ne_a must be true for each external atom a in Π, otherwise the according guessing rule would be unsatisfied, which contradicts the assumption that $\hat{\mathbf{A}}$ is an answer set of $\hat{\Pi}$.

However, if both e_a and ne_a would be true, then $\left(\hat{\mathbf{A}} \setminus \{\mathbf{T}ne_a\}\right) \cup \{\mathbf{F}ne_a\}$ would also be a model of $\hat{\Pi}$ because the guessing rule corresponding to a would still be satisfied and ne_a does not occur elsewhere in $\hat{\Pi}$. But then by minimality of answer sets, $\hat{\mathbf{A}}$ cannot be an answer set of $\hat{\Pi}$, which contradicts the assumption that it is a compatible set of Π. □

Proposition 3.1 allows us to slightly abuse notation by defining the truth value of only one of e_a or ne_a explicitly whenever we write compatible sets. We assume that the other atom has implicitly the opposite truth value.

For a compatible set $\hat{\mathbf{A}}$, let $\mathbf{A}$ be its projection to non-replacement atoms.

Example 16. Let Π be the program

$$\begin{aligned} dom(a);\ dom(b) & \\ p(a) &\leftarrow dom(a), \&g[p](a) \\ p(b) &\leftarrow dom(b), \&g[p](b) \end{aligned}$$

where $\&g$ implements the following mapping from $ext(\mathbf{A}, p)$ to $ext(\mathbf{A}, \&g[p])$:

$$\emptyset \mapsto \{b\}; \{a\} \mapsto \{a\}; \{b\} \mapsto \emptyset; \{a, b\} \mapsto \{a, b\}$$

[1] An assignment $\hat{\mathbf{A}}$ is an answer set of program $\hat{\Pi}$, if it is the minimal model of $\hat{\Pi}^{\hat{\mathbf{A}}}$.

Then the guessing program $\hat{\Pi}$ is given by the following set of rules:

$$\begin{aligned} dom(a);\ dom(b) & \\ p(a) &\leftarrow dom(a), e_{\&g[p]}(a) \\ p(b) &\leftarrow dom(b), e_{\&g[p]}(b) \\ e_{\&g[p]}(a) \vee ne_{\&g[p]}(a) &\leftarrow \\ e_{\&g[p]}(b) \vee ne_{\&g[p]}(b) &\leftarrow \end{aligned}$$

The interpretation

$$\hat{\mathbf{A}} = \{\mathbf{T}dom(a), \mathbf{T}dom(b), \mathbf{T}p(a), \mathbf{F}p(b), \mathbf{T}e_{\&g[p]}(a), \mathbf{F}ne_{\&g[p]}(a), \mathbf{F}e_{\&g[p]}(b), \mathbf{T}ne_{\&g[p]}(b)\}$$

is a compatible set of $\hat{\Pi}$. □

The search for compatible sets will sometimes be called *Main Search* and is based on conflict-driven nogood learning (CDNL) as recapitulated in Section 2.3.

A compatible set is not necessarily an FLP answer set of a program, but the set of all compatible sets includes all FLP answer sets. More formally, an answer set $\mathbf{A}$ of a program Π corresponds to the compatible set

$$\begin{aligned} \kappa(\Pi, \mathbf{A}) = \mathbf{A} \cup \big\{ & \mathbf{T}e_a, \mathbf{F}ne_a \mid a \text{ is an external atom in } \Pi, \mathbf{A} \models a \big\} \\ \cup \big\{ & \mathbf{F}e_a, \mathbf{T}ne_a \mid a \text{ is an external atom in } \Pi, \mathbf{A} \not\models a \big\}. \end{aligned}$$

We will prove this formally below.

Identifying the answer sets under all compatible sets requires an additional *minimality check*. That is, for each compatible set $\hat{\mathbf{A}}$ one needs to check whether $\mathbf{A}$ is a subset-minimal model of $f\Pi^{\mathbf{A}}$. Because the minimality is checked with respect to the FLP-reduct, this check will also be called *FLP check*. However, we postpone the discussion of this check to later subsections but first describe an algorithm which computes compatible sets of HEX-programs. The algorithm is based on the traditional guess and check algorithm as shown above, but learns additional nogoods from external source evaluations. The overall approach is visualized in Figure 3.1.

3.1.1 Learning-Based Evaluation Algorithm

It was observed that the naive guess and check approach suffers scalability problems. Although the introduction of the model-building framework [Schüller, 2012] eased these problems, applicability of the formalism to real-world applications is still moderate. The reasons for this can be found in the blind guessing of all possible truth assignments to the external atoms. This leads to an exponential number of candidate compatible sets, which have to be checked against the external sources. In practice, many of them fail the check because of the same reason, i.e., there might be repetitive computation. Therefore, it is a good idea to *learn* from evaluations of external atoms such that assignments which violate the known behavior of external sources are not generated again. The learned knowledge is represented by additional nogoods which are added to the reasoner. Naively, one can simply observe the input-output relationships of external atoms

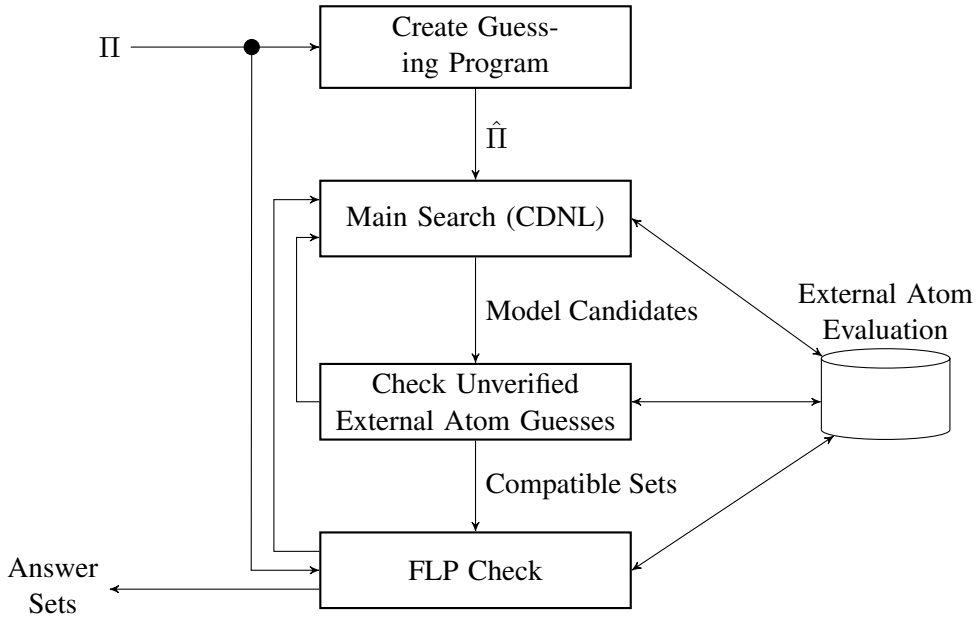


Figure 3.1: Visualization of the Overall Algorithm

during evaluation, but more advanced learning techniques are possible if side information about properties of the external sources are known. As we will see, it is even possible to allow the user for writing customized rules for nogood learning, which can exploit specific domain-dependent knowledge for each external source.

In this section we abstractly use the set of nogoods $\Lambda(\&g[\mathbf{y}], \mathbf{A})$ which are learned during the evaluation of $\&g$ with input list $\mathbf{y}$ given the assignment $\mathbf{A}$. Clearly, the learned information must not be arbitrary but related to the actual behavior of the external source. The crucial requirement is called *correctness*, which intuitively holds if the nogood can be added without eliminating compatible sets.

Definition 31 (Correct Nogoods)**.** A nogood δ is *correct wrt. a program* Π, if all compatible sets of Π are solutions to δ.

The overall algorithm consists of two parts. First, Algorithm Hex-CDNL computes model candidates; it is essentially an ordinary ASP solver (cf. Algorithm DASP-CDNL), but includes calls to external sources in order to learn additional nogoods. The external calls in this algorithm are not required for correctness of the algorithm, but may influence performance dramatically as discussed in Chapter 5. Second, Algorithm GuessAndCheckHexEvaluation uses Algorithm Hex-CDNL to produce model candidates and checks each of them against the external sources (followed by a minimality check). Here, the external calls are crucial for correctness of the algorithm.

For computing a model candidate, Algorithm Hex-CDNL basically employs the conflict-driven approach presented by Drescher et al. (2008) as summarized in Section 2, where the main difference is the addition of Parts (c) and (d).

Algorithm GuessAndCheckHexEvaluation

Input: A HEX-program Π

Output: All answer sets of Π

$\hat{\Pi} \leftarrow \Pi$ with all external atoms $\&g[\mathbf{y}](\mathbf{x})$ replaced by $e_{\&g[\mathbf{y}]}(\mathbf{x})$
Add guessing rules for all external atoms to $\hat{\Pi}$
$\nabla \leftarrow \emptyset$ `// set of dynamic nogoods`
$S \leftarrow \emptyset$ `// set of all compatible sets`

(a) **while** $\hat{\mathbf{C}} \neq \bot$ **do**
  $\hat{\mathbf{C}} \leftarrow \bot$
  $inconsistent \leftarrow false$
(b) **while** $\hat{\mathbf{C}} = \bot$ *and* $inconsistent = false$ **do**
(c) $\hat{\mathbf{A}} \leftarrow$ Hex-CDNL$(\Pi, \hat{\Pi}, \nabla)$
    if $\hat{\mathbf{A}} = \bot$ **then**
      $inconsistent \leftarrow true$
    else
      $compatible \leftarrow true$
(d) **for** *all external atoms with input list* $\&g[\mathbf{y}]$ *in* Π **do**
        Evaluate $\&g[\mathbf{y}]$ wrt. $\hat{\mathbf{A}}$
(e) $\nabla \leftarrow \nabla \cup \Lambda(\&g[\mathbf{y}], \hat{\mathbf{A}})$
        Let $\hat{\mathbf{A}}^{\&g[\mathbf{y}](\mathbf{x})} = 1 \Leftrightarrow \mathbf{T}e_{\&g[\mathbf{y}](\mathbf{x})} \in \hat{\mathbf{A}}$
        if $\exists \boldsymbol{x} : f_{\&g}(\hat{\mathbf{A}}, \boldsymbol{y}, \boldsymbol{x}) \neq \hat{\mathbf{A}}^{\&g[\mathbf{y}](\boldsymbol{x})}$ **then**
          Add $\hat{\mathbf{A}}$ to ∇
          $compatible \leftarrow false$
      if $compatible$ **then** $\hat{\mathbf{C}} \leftarrow \hat{\mathbf{A}}$
  if $inconsistent = false$ **then**
    `//` $\hat{\mathbf{C}}$ `is a compatible set of` Π
    $\nabla \leftarrow \nabla \cup \{\hat{\mathbf{C}}\}$
    if *FLPCheck*$(\Pi, \hat{\mathbf{C}}, \nabla)$ **then**
      $S \leftarrow S \cup \{\hat{\mathbf{C}}\}$

return $\big\{\{\mathbf{T}a \in \hat{\mathbf{C}} \mid a \in A(\Pi)\} \mid \hat{\mathbf{C}} \in S\big\}$

Algorithm Hex-CDNL

Input: A program Π, its guessing program $\hat{\Pi}$, a set of nogoods ∇ of Π
Output: An answer set of $\hat{\Pi}$ (candidate for a compatible set of Π) which is a solution to all nogoods $d \in \nabla$, or $\bot$ if none exists

$\hat{\mathbf{A}} \leftarrow \emptyset$ // assignment over $A(\hat{\Pi}) \cup BA(\hat{\Pi}) \cup BA\big(sh(\hat{\Pi})\big)$
$dl \leftarrow 0$ // decision level
while *true* **do**
  $(\hat{\mathbf{A}}, \nabla) \leftarrow$ Propagation$(\hat{\Pi}, \nabla, \hat{\mathbf{A}})$
(a) **if** $\delta \subseteq \hat{\mathbf{A}}$ *for some* $\delta \in \Delta_{\hat{\Pi}} \cup \Theta_{sh(\hat{\Pi})} \cup \nabla$ **then**
    if $dl = 0$ **then return** $\bot$
    $(\epsilon, k) \leftarrow$ Analysis$(\delta, \hat{\Pi}, \nabla, \hat{\mathbf{A}})$
    $\nabla \leftarrow \nabla \cup \{\epsilon\}$
    $\hat{\mathbf{A}} \leftarrow \hat{\mathbf{A}} \setminus \big\{\sigma \in \hat{\mathbf{A}} \mid k < dl(\sigma)\big\}$
    $dl \leftarrow k$
(b) **else if** $\hat{\mathbf{A}}^{\mathbf{T}} \cup \hat{\mathbf{A}}^{\mathbf{F}} = A(\hat{\Pi}) \cup BA(\hat{\Pi}) \cup BA\big(sh(\hat{\Pi})\big)$ **then**
    $U \leftarrow$ UnfoundedSet$(\hat{\Pi}, \hat{\mathbf{A}})$
    if $U \neq \emptyset$ **then**
      Let $\delta \in \lambda_{\hat{\Pi}}(U)$ such that $\delta \subseteq \hat{\mathbf{A}}$
      if $\big\{\sigma \in \delta \mid 0 < dl(\sigma)\big\} = \emptyset$ **then return** $\bot$
      $(\epsilon, k) \leftarrow$ Analysis$(\delta, \hat{\Pi}, \nabla, \hat{\mathbf{A}})$
      $\nabla \leftarrow \nabla \cup \{\epsilon\}$
      $\hat{\mathbf{A}} \leftarrow \hat{\mathbf{A}} \setminus \big\{\sigma \in \hat{\mathbf{A}} \mid k < dl(\sigma)\big\}$
      $dl \leftarrow k$
    else
      return $\hat{\mathbf{A}} \cap \big\{\mathbf{T}a, \mathbf{F}a \mid a \in A(\hat{\Pi})\big\}$
(c) **else if** *Heuristics decides to evaluate* $\&g[\mathbf{y}]$ **then**
    Evaluate $\&g[\mathbf{y}]$ wrt. $\hat{\mathbf{A}}$
    $\nabla \leftarrow \nabla \cup \Lambda(\&g[\mathbf{y}], \hat{\mathbf{A}})$
(d) **else if** *Heuristics decides to do a UFS check* **then**
    Let $\Pi' \subseteq \Pi$ s.t. $\hat{\mathbf{A}} \cap \big\{\mathbf{T}a, \mathbf{F}a \mid a \in A(\hat{\Pi}')\big\}$ is a compatible set of Π'
    FLPCheck$(\Pi', \mathbf{A}, \nabla)$
(e) **else**
    $\sigma \leftarrow$ Select$(\hat{\Pi}, \nabla, \hat{\mathbf{A}})$
    $dl \leftarrow dl + 1$
    $\hat{\mathbf{A}} \leftarrow \hat{\mathbf{A}} \cup \{\sigma\}$

Our extension in Part (c) is driven by the following idea: whenever unit and unfounded set propagation does not derive any further atoms and the assignment is still incomplete, the algorithm possibly evaluates external atoms with already known input instead of guesing truth values. This might lead to the addition of new nogoods, which can in turn cause the propagation procedure to derive further atoms. Guessing of truth values only becomes necessary if no deterministic conclusions can be drawn and the evaluation of external atoms does not yield further nogoods, or evaluation of external atoms is denied by the heuristics. Our default heuristics evaluates an external atom $\&g[\mathbf{y}]$ whenever the input $\mathbf{y}$ is completely known. Different heuristics may evaluate external atoms even if some of their input atoms are not yet assigned, i.e., they evaluate external atoms wrt. partial assignments. This allows for adding nogoods in Part (c) which imply the truth values of other, yet unassigned input atoms. This technique comes from the field of SMT and is known as *theory propagation* [Nieuwenhuis and Oliveras, 2005]. However, while our algorithm (and our implementation) fully supports theory propagation, the development of concrete evaluation heuristics and learning functions which make use of this technique is strongly application specific. As this is out of the scope of this thesis we will use the default heuristics for all our benchmarks in Chapter 5.

Our second extension concerns Part (d), which may perform unfounded set checks already during the search for compatible sets. Here we simply assume that this check does not eliminate answer sets of Π, i.e., $FLPCheck(\Pi', \mathbf{A}, \nabla)$ does not add any nogoods to ∇ which are violated by some answer set of Π. Unfounded set checking will be described in Section 3.2, where we also discuss how to do such checks wrt. partial assignments. For the minimality check, the interpretation must be complete for a subprogram and the guesses of all external atoms in this subprogram must be correct.

For a more formal treatment, let $EI(\Pi)$ be the set of all external predicates with input list that occur in Π, and let $\mathcal{D}(\Pi)$ be the set of all signed literals over atoms in $A(\Pi) \cup A(\hat{\Pi}) \cup BA(\hat{\Pi})$. Then, a *learning function* for Π is a mapping $\Lambda\colon\ EI(\Pi) \times 2^{\mathcal{D}(\Pi)} \mapsto 2^{2^{\mathcal{D}(\Pi)}}$. We extend our notion of correct nogoods to correct learning functions $\Lambda(\cdot,\cdot)$, as follows:

Definition 32. A learning function Λ is *correct* for program Π, if every $d \in \Lambda(\&g[\mathbf{y}], \mathbf{A})$ is correct for Π, for all $\&g[\mathbf{y}]$ in $EI(\Pi)$ and $\mathbf{A} \in 2^{\mathcal{D}(\Pi)}$.

Restricting to learning functions that are correct for Π, the following results hold.

Proposition 3.2. *If for input Π, $\hat{\Pi}$ and ∇, Algorithm Hex-CDNL returns (i) an interpretation $\hat{\mathbf{A}}$, then $\hat{\mathbf{A}}$ is an answer set of $\hat{\Pi}$ and a solution to ∇; (ii) $\bot$, then Π has no compatible set that is a solution to ∇ and such that its restriction to positive atoms is an answer set of Π.*

Proof. (i) The proof mainly follows Drescher et al. (2008). In our algorithm we have potentially more nogoods, which can never produce further answer sets but only eliminate them. Hence, each produced interpretation $\hat{\mathbf{A}}$ is an answer set of $\hat{\Pi}$.

(ii) By completeness of the algorithm of Drescher et al. (2008) we only need to justify that adding nogoods in Parts (c) and (d) does not eliminate compatible sets whose restriction to positive atoms are answer sets of Π. In Part (c), adding the nogoods $\Lambda(\&g[\mathbf{y}], \hat{\mathbf{A}})$ after evaluation of $\&g[\mathbf{y}]$ does not eliminate compatible sets of Π. For this purpose we need to show that when one of the added nogoods is violated, the interpretation is incompatible with the external sources

anyway. But this follows from the correctness of $\Lambda(\cdot,\cdot)$ and (for derived nogoods) from the completeness of the algorithm of Drescher et al. (2008). In Part (d), it holds by our assumption about Algorithm FLPCheck that all answer sets of Π are solutions to all added nogoods. □

The basic idea of Algorithm GuessAndCheckHexEvaluation is to compute all compatible sets of Π by the loop at (a) and checking subset-minimality wrt. the FLP-reduct afterwards. While minimality checking is explained in detail in Section 3.2, we first focus on the computation of compatible sets and assume that there is a proper implementation of Algorithm FLPCheck which identifies the FLP answer sets among all compatible sets of the program Π at hand and adds only nogoods to ∇ such that all answer sets of Π are solutions to them. However, it is essential that the compatible sets restricted to ordinary atoms include all FLP answer sets, i.e., we do not miss answer sets. This is formalized as follows.

Proposition 3.3. *For every program Π, each answer set $\mathbf{A}$ can be extended to a compatible set $\hat{\mathbf{A}} = \kappa(\Pi, \mathbf{A})$.*

Proof. Let $\mathbf{A}$ be an answer set of Π and let

$$\begin{aligned}\hat{\mathbf{A}} = \kappa(\Pi, \mathbf{A}) = \mathbf{A} \cup \big\{ &\mathbf{T}e_a, \mathbf{F}ne_a \mid a \text{ is an external atom in } \Pi, \mathbf{A} \models a \big\} \\ \cup \big\{ &\mathbf{F}e_a, \mathbf{T}ne_a \mid a \text{ is an external atom in } \Pi, \mathbf{A} \not\models a \big\}.\end{aligned}$$

We show that $\hat{\mathbf{A}}$ is a compatible set of Π.

Since $\mathbf{A}$ is an answer set of Π, it is also a model. Since the truth values of all replacement atoms in $\hat{\mathbf{A}}$ coincide with the oracle function of all corresponding external atoms in Π wrt. $\mathbf{A}$ by definition of $\hat{\mathbf{A}}$, it satisfies all rules in $\hat{r} \in \hat{\Pi}$ which result from a rule in $r \in \Pi$ by substituting external atoms by replacement atoms. Moreover, for each external atom $\&g[\mathbf{y}](\mathbf{x})$ in Π, exactly one of $e_{\&g[\mathbf{y}]}(\mathbf{x})$ or $ne_{\&g[\mathbf{y}]}(\mathbf{x})$ is true in $\hat{\mathbf{A}}$, thus it also satisfies the external atom guessing rules in $\hat{\Pi}$ and is thus a model of $\hat{\Pi}$.

It remains to show that $\hat{\mathbf{A}}$ is also a subset-minimal model of $f\hat{\Pi}^{\hat{\mathbf{A}}}$. Suppose $\hat{\mathbf{A}}$ is not a subset-minimal model of $f\hat{\Pi}^{\hat{\mathbf{A}}}$. We show that then $\mathbf{A}$ is also not a subset-minimal model of $f\Pi^{\mathbf{A}}$, i.e., it is not an answer set of Π. If $\hat{\mathbf{A}}$ is not a subset-minimal model of $f\hat{\Pi}^{\hat{\mathbf{A}}}$, then there is a smaller model $\hat{\mathbf{A}}'$. Note that the truth values of the replacement atoms in $\hat{\mathbf{A}}$ and $\hat{\mathbf{A}}'$ are the same because $\hat{\mathbf{A}}$ contains exactly one of $\mathbf{T}e_{\&g[\mathbf{y}]}(\mathbf{x})$ or $\mathbf{T}ne_{\&g[\mathbf{y}]}(\mathbf{x})$ for each external atom $\&g[\mathbf{y}](\mathbf{x})$ in Π. We show that the restriction $\mathbf{A}'$ of $\hat{\mathbf{A}}'$ to ordinary atoms is a model of $f\Pi^{\mathbf{A}}$. Because $\hat{\mathbf{A}}'$ is a model of $f\hat{\Pi}^{\hat{\mathbf{A}}}$, it is a model of every rule $\hat{r} \in f\hat{\Pi}^{\hat{\mathbf{A}}}$. But then either $\mathbf{T}h \in \hat{\mathbf{A}}'$ for some $h \in H(\hat{r})$ or $\mathbf{F}b \in \hat{\mathbf{A}}'$ for some $b \in B(\hat{r})$. However, in the latter case b cannot be a (positive or default-negated) external atom replacement, because this would imply $\mathbf{F}b \in \hat{\mathbf{A}}$ ($\hat{\mathbf{A}}$ and $\hat{\mathbf{A}}'$ coincide on replacement atoms) and contradict the assumption that $\hat{r}$ is in the reduct.

Observe that $f\hat{\Pi}^{\hat{\mathbf{A}}}$ contains all rules from $f\Pi^{\mathbf{A}}$, but with replacement atoms in place of external atoms. The corresponding rule $r \in f\Pi^{\mathbf{A}}$ of $\hat{r}$ contains the same ordinary atoms in the rule head and body as $\hat{r}$. As $\mathbf{A}'$ is the restriction of $\hat{\mathbf{A}}'$ to ordinary atoms, we have $\mathbf{A}' \models r$. Thus, $\mathbf{A}'$ is a model of $f\Pi^{\mathbf{A}}$, which contradicts the assumption that $\mathbf{A}$ is an answer set of Π. □

For computing compatible sets, the loop at (b) uses Algorithm Hex-CDNL to compute answer sets of $\hat{\Pi}$ in (c), i.e., candidate compatible sets of Π, and subsequently checks compatibility for each external atom in (d). Here the external calls are crucial for correctness. However, different from the translation approach, the external source evaluation serves not only for compatibility checking, but also for generating additional dynamic nogoods $\Lambda(\&g[\mathbf{y}], \hat{\mathbf{A}})$ in Part (e). We have the following result, provided that Algorithm FLPCheck identifies the answer sets among the compatible sets of Π and adds only nogoods to ∇ s.t. all answer sets of Π are solutions to them.

Theorem 1 (Soundness and Completeness of Algorithm GuessAndCheckHexEvaluation)**.** *Algorithm GuessAndCheckHexEvaluation computes all answer sets of* Π.

Proof. We first show that the loop at (b) yields after termination a compatible set $\hat{\mathbf{C}}$ of Π that is a solution of ∇ at the stage of leaving the loop iff such a compatible set does exist, and yields $\hat{\mathbf{C}} = \bot$ iff no such compatible set exists.

Suppose that $\hat{\mathbf{C}} \neq \bot$ after the loop. Then $\hat{\mathbf{C}}$ was assigned $\hat{\mathbf{A}} \neq \bot$, which was returned by Hex-CDNL(Π, $\hat{\Pi}$, ∇). From Proposition 3.2 (i) it follows that $\hat{\mathbf{C}}$ is an answer set of $\hat{\Pi}$ and a solution to ∇. Thus (i) of Definition 30 holds. As $compatible = true$, the for loop guarantees the compatibility with the external sources in (ii) of Definition 30: if some source output on input from $\hat{\mathbf{C}}$ is not compatible with the guess, $\hat{\mathbf{C}}$ is rejected (and added as nogood). Otherwise $\hat{\mathbf{C}}$ coincides with the behavior of the external sources, i.e., it satisfies (ii) of Definition 30 and *no* further nogoods are added. Thus, $\hat{\mathbf{C}}$ is a compatible set of Π wrt. ∇ at the time of leaving the loop.

Otherwise, after the loop $\hat{\mathbf{C}} = \bot$. Then $inconsistent = true$, which means that the call Hex-CDNL(Π, $\hat{\Pi}$, ∇) returned $\bot$. By Proposition 3.2 (ii) there is no answer set of $\hat{\Pi}$ which is a solution to ∇. As only correct nogoods were added to ∇, there exists also no answer set of $\hat{\Pi}$ which is a solution to set ∇. Thus the loop at (b) operates as desired.

If Algorithm FLPCheck adds no nogoods to ∇, then the loop at (a) then enumerates one by one all compatible sets of Π and terminates: the update of ∇ with $\hat{\mathbf{C}}$ prevents recomputing $\hat{\mathbf{C}}$, and thus the number of compatible sets decreases. If we assume that Algorithm FLPCheck correctly identifies the FLP answer sets among all compatible sets of Π, as we will formally show in Theorem 3 after introducing an algorithm which implements Algorithm FLPCheck, we have shown that the overall algorithm correctly outputs all answer sets of Π. If Algorithm FLPCheck adds nogoods to ∇, then by assumption all answer sets of Π are solutions to them. Thus, if these nogoods eliminate compatible sets of Π, then they are not relevant because they cannot be answer sets anyway, thus we do not lose relevant interpretations. □

Example 17. Let $\&empty$ be an external atom with one (nonmonotonic) predicate input p, such that its output is c_0 if the extension of p is empty and c_1 otherwise. Consider the program Π consisting of the following rules:

$$p(c_0);\ \ dom(c_0);\ \ dom(c_1);\ \ dom(c_2)$$
$$p(X) \leftarrow dom(X), \&empty[p](X)$$

Algorithm GuessAndCheckHexEvaluation transforms Π into the guessing program $\hat{\Pi}$:

$$p(c_0);\ \ dom(c_0);\ \ dom(c_1);\ \ dom(c_2)$$
$$p(X) \leftarrow dom(X), e_{\&empty[p]}(X)$$
$$e_{\&empty[p]}(X) \vee ne_{\&empty[p]}(X) \leftarrow dom(X)$$

The traditional evaluation strategy without learning will then produce 2^3 model candidates in Algorithm Hex-CDNL, which are subsequently checked in Algorithm GuessAndCheckHexEvaluation. For instance, the guessed truth values of external atom replacements $\{\mathbf{T}ne_{\&empty[p]}(c_0), \mathbf{T}e_{\&empty[p]}(c_1), \mathbf{T}ne_{\&empty[p]}(c_2)\}$ lead to the candidate compatible set $\{\mathbf{T}ne_{\&empty[p]}(c_0), \mathbf{T}e_{\&empty[p]}(c_1), \mathbf{T}ne_{\&empty[p]}(c_2), \mathbf{T}p(c_1)\}$ (neglecting false atoms and facts). This is also the only model candidate which passes the compatibility check: $p(c_0)$ is always true, and therefore $e_{\&empty[p]}(c_1)$ must also be true due to definition of the external atom. This allows for deriving $p(c_1)$ by the first rule of the program. All other atoms are false due to minimality of answer sets (note that minimality wrt. the ordinary ASP program is already guaranteed by Hex-CDNL). □

3.1.2 Concrete Learning Functions for External Behavior Learning

We now discuss nogoods generated for external behavior learning (EBL) in detail. EBL is triggered by external source evaluations instead of conflicts. The basic idea is to integrate knowledge about the external source behavior into the program to guide the search. The program evaluation then starts with an empty set of learned nogoods and the preprocessor generates a guessing rule for each ground external atom, as discussed above, but further nogoods are added during the evaluation as more information about external sources becomes available. This is in contrast to traditional evaluation, where external atoms are assigned arbitrary truth values which are checked only after the assignment was completed.

We will first show how to construct useful learned nogoods after evaluating external atoms, if we have no further information about the internals of external sources, which we call *uninformed learning*. In this case we can only learn simple input/output relationships. Subsequently we consider *informed learning*, where additional information about properties of external sources is available. This allows for using more elaborated learning strategies.

Uninformed Learning

We first assume that we do not have information about the internals and consider external sources as black boxes. Hence, we can just apply very general rules for learning: whenever an external predicate with input list $\&g[\mathbf{y}]$ is evaluated wrt. an assignment $\mathbf{A}$, we learn that the input $\mathbf{A}|_{\mathbf{y}_{pred}}$ for $\mathbf{y}_{pred} = p_1, \ldots, p_n$ to the external atom $\&g$ produces the output $ext(\&g[\mathbf{y}], \mathbf{A})$, where $\mathbf{y}_{pred}$ is the sublist of $\mathbf{y}$ containing all predicate input parameters. This can be formalized as the following set of nogoods.

Definition 33. The learning function for a general external predicate with input list $\&g[\mathbf{y}]$ in program Π wrt. assignment $\mathbf{A}$ is defined as follows:

$$\Lambda_g(\&g[\mathbf{y}], \mathbf{A}) = \left\{ \mathbf{A}|_{\mathbf{y}_{pred}} \cup \{\mathbf{F}e_{\&g[\mathbf{y}]}(\mathbf{x})\} \mid \mathbf{x} \in ext(\&g[\mathbf{y}], \mathbf{A}) \right\}$$

Guess	Learned Nogood
$\left\{ \begin{array}{l} \mathbf{T}e_{\&empty[p]}(c_0), \mathbf{T}ne_{\&empty[p]}(c_1), \\ \mathbf{T}ne_{\&empty[p]}(c_2) \end{array} \right\}$	$\{\mathbf{T}p(c_0), \mathbf{F}p(c_1), \mathbf{F}p(c_2), \mathbf{F}e_{\&empty[p]}(c_1)\}$
$\left\{ \begin{array}{l} \mathbf{T}e_{\&empty[p]}(c_0), \mathbf{T}ne_{\&empty[p]}(c_1), \\ \mathbf{T}e_{\&empty[p]}(c_2), \mathbf{T}p(c_2) \end{array} \right\}$	$\{\mathbf{T}p(c_0), \mathbf{F}p(c_1), \mathbf{T}p(c_2), \mathbf{F}e_{\&empty[p]}(c_1)\}$
$\left\{ \begin{array}{l} \mathbf{T}e_{\&empty[p]}(c_0), \mathbf{T}e_{\&empty[p]}(c_1), \\ \mathbf{T}ne_{\&empty[p]}(c_2), \mathbf{T}p(c_1) \end{array} \right\}$	$\{\mathbf{T}p(c_0), \mathbf{T}p(c_1), \mathbf{F}p(c_2), \mathbf{F}e_{\&empty[p]}(c_1)\}$
$\left\{ \begin{array}{l} \mathbf{T}e_{\&empty[p]}(c_0), \mathbf{T}e_{\&empty[p]}(c_1), \\ \mathbf{T}e_{\&empty[p]}(c_2), \mathbf{T}p(c_1), \mathbf{T}p(c_2) \end{array} \right\}$	$\{\mathbf{T}p(c_0), \mathbf{T}p(c_1), \mathbf{T}p(c_2), \mathbf{F}e_{\&empty[p]}(c_1)\}$

Table 3.1: Learned Nogoods of Example 18

In the simplest case, an external atom has no input and the learned nogoods are unary, i.e., of the form $\{\mathbf{F}e_{\&g[]}(\mathbf{x})\}$. Thus, it is learned that certain tuples are in the output of the external source, i.e. they must not be false. For external sources with input predicates, the added rules encode the relationship between the output tuples and the provided input.

Example 18 (ctd.)**.** Recall Π from Example 17. Without learning, the algorithms produce 2^3 model candidates and check them subsequently. It turns out that EBL allows for falsification of some of the guesses without actually evaluating the external atoms. Suppose the reasoner first tries the guesses containing literal $\mathbf{T}e_{\&empty[p]}(c_0)$. While they are checked against the external sources, the described learning function allows for adding the externally learned nogoods shown in Table 3.1. Observe that the combination $\mathbf{T}p(c_0), \mathbf{F}p(c_1), \mathbf{F}p(c_2)$ will be reconstructed also for different choices of the guessing variables. As $p(c_0)$ is a fact, it is true independent of the choice between $e_{\&empty[p]}(c_0)$ and $ne_{\&empty[p]}(c_0)$. E.g., the guess $\mathbf{F}e_{\&empty[p]}(c_0)$, $\mathbf{F}e_{\&empty[p]}(c_1)$, $\mathbf{F}e_{\&empty[p]}(c_2)$ leads to the same extension of p. This allows for reusing the nogood, which is immediately invalidated without evaluating the external atoms. Different guesses with the same input to an external source allow for reusing learned nogoods, at the latest when the candidate is complete, but before the external source is called for validation. However, very often learning allows for discarding guesses even earlier. For instance, we can derive $\{\mathbf{T}p(c_0), \mathbf{F}e_{\&empty[p]}(c_1)\}$ from the nogoods above in 3 resolution steps. Such derived nogoods will be learned after running into a couple of conflicts. We can derive $\mathbf{T}e_{\&empty[p]}(c_1)$ from $p(c_0)$ even before the truth value of $\mathbf{F}e_{\&empty[p]}(c_1)$ is set, i.e., external learning guides the search while the traditional evaluation algorithm considers the behavior of external sources only during postprocessing. □

Lemma 3.1. *Let* Π *be a program which contains an external atom of form* $\&g[\mathbf{y}](\cdot)$*. For all assignments* $\mathbf{A}$*, the nogoods* $\Lambda_g(\&g[\mathbf{y}], \mathbf{A})$ *in Definition 33 are correct wrt.* Π*.*

Proof. The added nogood for an output tuple $\mathbf{x} \in ext(\&g[\mathbf{y}], \mathbf{A})$ contains $\mathbf{A}|_{\mathbf{y}_{pred}}$ and the negated replacement atom $\mathbf{F}e_{\&g[\mathbf{y}]}(\mathbf{x})$. If the nogood is violated, then the guess was wrong as the replacement atom is guessed false but the tuple $(\mathbf{x})$ is in the output. Hence, the interpretation is not compatible and cannot be a compatible set anyway. □

Informed Learning

The learned nogoods of the above form can become quite large as they include the whole input to the external source. However, known properties of external sources can be exploited in order to learn smaller and more general nogoods. For example, if one of the input parameters of an external source is monotonic, it is not necessary to include information about false atoms in its extension, as the output will not shrink given larger input.

Properties for informed learning can be stated either on the level of *predicates* or on the level of individual *external atoms* (see Chapter 5). The former means that all usages of the predicate have the property. To understand this, consider predicate $\&union$ which takes two predicate inputs p and q and computes the set of all elements which are in at least one of the extensions of p or q. It will be *always* monotonic in both parameters, independently of its usage in a program. While an external source may lack a property in general, it may hold for particular usages.

Example 19. Consider an external atom $\&db[r_1, \ldots, r_n, query](\mathbf{y})$ as an interface to an SQL query processor, which evaluates a given query (given as string) over tables (relations) provided by predicates $r_1, \ldots, r_n$. In general, the atom will be nonmonotonic, but for special queries (e.g., simple selection of all tuples), it will be monotonic. □

Next, we discuss three particular cases of informed learning which customize the default learning function for generic external sources by exploiting properties of external sources, and finally present examples where the learning of user-defined nogoods might be useful.

Monotonic and Antimonotonic Atoms. A predicate parameter p_i of an external atom $\&g$ is called *monotonic*, if $f_{\&g}(\mathbf{A}, \mathbf{y}, \mathbf{x}) = 1$ implies $f_{\&g}(\mathbf{A}', \mathbf{y}, \mathbf{x}) = 1$ for all $\mathbf{A}'$ with $\mathbf{A}'|_{p_i} \supseteq \mathbf{A}|_{p_i}$ and $\mathbf{A}'|_{p'} = \mathbf{A}|_{p'}$ for all other predicate parameters $p' \neq p_i$. It is called *antimonotonic*, if $f_{\&g}(\mathbf{A}, \mathbf{y}, \mathbf{x}) = 0$ implies $f_{\&g}(\mathbf{A}', \mathbf{y}, \mathbf{x}) = 0$ for all $\mathbf{A}'$ with $\mathbf{A}'|^{\mathbf{T}}_{p_i} \supseteq \mathbf{A}|^{\mathbf{T}}_{p_i}$ and $\mathbf{A}'|_{p'} = \mathbf{A}|_{p'}$ for all other predicate parameters $p' \neq p_i$. It is called *nonmonotonic*, if it is neither monotonic nor antimonotonic. The learned nogoods $\Lambda(\&g[\mathbf{y}], \mathbf{A})$ after evaluating $\&g[\mathbf{y}]$ are not required to include $\mathbf{F}p_i(t_1, \ldots, t_\ell)$ for monotonic $p_i \in \mathbf{y}$ or $\mathbf{T}p_i(t_1, \ldots, t_\ell)$ for antimonotonic $p_i \in \mathbf{y}$. That is, for an external predicate with input list $\&g[\mathbf{y}]$ with monotonic predicate input parameters $\mathbf{p}_m \subseteq \mathbf{y}$, antimonotonic predicate input parameters $\mathbf{a}_m \subseteq \mathbf{y}$ and nonmonotonic predicate parameters $\mathbf{p}_n = \mathbf{y} \setminus (\mathbf{p}_m \cup \mathbf{p}_a)$, the set of learned nogoods can be restricted as follows.

Definition 34. The learning function for an external predicate $\&g$ with input list $\mathbf{y}$ in program Π wrt. assignment $\mathbf{A}$, such that $\&g$ is monotonic in predicate input parameters $\mathbf{p}_m \subseteq \mathbf{y}$ and antimonotonic in predicate input parameters $\mathbf{p}_a \subseteq \mathbf{y}$, is defined as follows:

$$\Lambda_m(\&g[\mathbf{y}], \mathbf{A}) = \left\{ \begin{array}{l} \{\mathbf{T}a \in \mathbf{A}|_{\mathbf{p}_m}\} \cup \{\mathbf{F}a \in \mathbf{A}|_{\mathbf{p}_a}\} \cup \\ \mathbf{A}|_{\mathbf{p}_n} \cup \{\mathbf{F}e_{\&g[\mathbf{y}]}(\mathbf{x})\} \end{array} \;\middle|\; \mathbf{x} \in ext(\&g[\mathbf{y}], \mathbf{A}) \right\}$$

Example 20. Consider the external atom $\&diff[p, q](X)$ which computes the set of all elements X that are in the extension of p, but not in the extension of q. Suppose it is evaluated wrt. $\mathbf{A}$, s.t. $ext(p, \mathbf{A}) = \{\mathbf{T}p(a), \mathbf{T}p(b), \mathbf{F}p(c)\}$ and $ext(q, \mathbf{A}) = \{\mathbf{F}q(a), \mathbf{T}q(b), \mathbf{F}q(c)\}$. Then the output of the atom is $ext(\&diff[p, q], \mathbf{A}) = \{a\}$ and the (only) naively learned nogood is

$\{\mathbf{T}p(a), \mathbf{T}p(b), \mathbf{F}p(c), \mathbf{F}q(a), \mathbf{T}q(b), \mathbf{F}q(c), \mathbf{F}e_{\&diff[p,q]}(a)\}$. However, due to monotonicity of $\&diff[p,q]$ in p and antimonotonicity in q, it is not necessary to include $\mathbf{F}p(c)$ or $\mathbf{T}q(b)$ in the nogood; the output of the external source will not shrink even if $p(c)$ becomes true or $q(b)$ becomes false. Therefore the (more general) nogood $\{\mathbf{T}p(a), \mathbf{T}p(b), \mathbf{F}q(a), \mathbf{F}q(c), \mathbf{F}e_{\&diff[p,q]}(a)\}$ suffices to correctly describe the input-output behavior. □

Lemma 3.2. *Let Π be a program which contains an external atom of form $\&g[\mathbf{y}](\cdot)$. For all assignments $\mathbf{A}$, the nogoods $\Lambda_m(\&g[\mathbf{y}], \mathbf{A})$ in Definition 34 are correct wrt. Π.*

Proof. We must show that negative input literals over monotonic parameters and positive input literals over antimonotonic parameters can be removed from the learned nogoods without affecting correctness. For uninformed learning, we argued that for output tuple $\mathbf{x} \in ext(\&g[\mathbf{y}], \mathbf{A})$, the replacement atom $e_{\&g[\mathbf{y}]}(\mathbf{x})$ must not be guessed false if the input to $\&g[\mathbf{y}](\mathbf{x})$ is $\mathbf{A}|_{\mathbf{y}_{pred}}$. However, as the output of $\&g$ grows (shrinks) monotonically with the extension of a monotonic (antimonotonic) parameter $p \in \mathbf{p}_m$ ($p \in \mathbf{p}_a$), the same applies for any $\mathbf{A}'$ s.t. p has a larger (smaller) extension wrt. $\mathbf{A}'$, i.e., $\{\mathbf{T}a \in \mathbf{A}'|_p\} \supseteq \{\mathbf{T}a \in \mathbf{A}|_p\}$ ($\{\mathbf{T}a \in \mathbf{A}'|_p\} \subseteq \{\mathbf{T}a \in \mathbf{A}|_p\}$) and consequently $\{\mathbf{F}a \in \mathbf{A}'|_p\} \subseteq \{\mathbf{F}a \in \mathbf{A}|_p\}$ ($\{\mathbf{F}a \in \mathbf{A}'|_p\} \supseteq \{\mathbf{F}a \in \mathbf{A}|_p\}$). Hence, the negative literals over monotonic parameters and the positive literals over antimonotonic parameters are not relevant wrt. output tuple $\mathbf{x}$ and can be removed from the nogood. □

Functional Atoms. When evaluating $\&g[\mathbf{y}]$ with some $\&g$ that is a function wrt. assignment $\mathbf{A}$, only one output tuple can be contained in $ext(\&g[\mathbf{y}], \mathbf{A})$, formally: for all assignments $\mathbf{A}$ and all $\mathbf{x}$, if $f_{\&g}(\mathbf{A}, \mathbf{y}, \mathbf{x}) = 1$ then $f_{\&g}(\mathbf{A}, \mathbf{y}, \mathbf{x}') = 0$ for all $\mathbf{x}' \neq \mathbf{x}$. Therefore the following nogoods may be added right from the beginning.

Definition 35. The learning function for a functional external predicate $\&g$ with input list $\mathbf{y}$ in program Π wrt. assignment $\mathbf{A}$ is defined as follows:

$$\Lambda_f(\&g[\mathbf{y}], \mathbf{A}) = \left\{\{\mathbf{T}e_{\&g[\mathbf{y}]}(\mathbf{x}), \mathbf{T}e_{\&g[\mathbf{y}]}(\mathbf{x}')\} \mid \mathbf{x} \neq \mathbf{x}'\right\}$$

However, our implementation of this learning rule does not generate all pairs of output tuples beforehand. Instead, it memorizes all generated output tuples $\mathbf{x}^i$, $1 \leq i \leq k$ during evaluation of external sources. Whenever a new output tuple $\mathbf{x}'$ is added, it also adds all nogoods which force previously derived output tuples $\mathbf{x}^i$ to be false.

Example 21. Consider the rules

$$\begin{aligned} out(X) &\leftarrow \&concat[A, x](X), strings(A), dom(X) \\ strings(X) &\leftarrow dom(X), \text{not } out(X) \end{aligned}$$

where $\&concat[X, Y](C)$ is true iff$_{\text{def}}$ string C is the concatenation of strings X and Y, and observe that the external atom is involved in a cycle through negation. As the extension of the domain dom can be large, many ground instances of the external atom are generated. The traditional evaluation algorithm guesses their truth values in a completely uninformed fashion. E.g., $e_{\&concat}(x, x, xx)$ (the replacement atom of $\&concat[A, x](X)$ with $A = x$ and $X = xx$, where $dom(x)$ and $dom(xx)$ are supposed to be facts) is in each guess set randomly to true

or to false, independent of previous guesses. In contrast, with learning from external sources, the algorithm learns after the first evaluation that $e_{\&concat}(x, x, xx)$ must be true. Knowing that $\&concat$ is functional, all atoms $e_{\&concat}(x, x, O)$ with $O \neq xx$ must be false. □

Lemma 3.3. *Let* Π *be a program which contains an external atom of form* $\&g[\mathbf{y}](\cdot)$ *s.t.* $\&g$ *is functional. For all assignments* $\mathbf{A}$*, the nogoods* $\Lambda_f(\&g[\mathbf{y}], \mathbf{A})$ *in Definition 35 are correct wrt.* Π*.*

Proof. We must show that the nogoods $\left\{\{\mathbf{T}e_{\&g[\mathbf{y}]}(\mathbf{x}), \mathbf{T}e_{\&g[\mathbf{y}]}(\mathbf{x}')\} \mid \mathbf{x} \neq \mathbf{x}'\right\}$ are correct. Due to functionality, the external source cannot return more than one output tuple for the same input. Therefore no such guess can be part of a compatible set. Hence, the nogoods do not eliminate possible compatible sets. □

Linear Atoms. In some cases the evaluation of an external atom in fact answers multiple independent queries simultanously. Splitting such queries into simpler ones might increase the effects of learning. To this end, we call an external predicate $\&g$ with input list $\mathbf{y}$ *linear* wrt. a partitioning $\mathbf{A}_1, \cdots, \mathbf{A}_n$ of an assignment $\mathbf{A}$ if $ext(\mathbf{A}, \&g[\mathbf{y}]) = \bigcup_{i=1}^{n} ext(\mathbf{A}_i, \&g[\mathbf{y}])$.

Example 22. External predicate with input list $\&diff[p, q]$ is linear in partitionings $\mathbf{A}_1, \cdots, \mathbf{A}_n$ s.t. for all c, $\mathbf{T}p(c), \mathbf{T}q(c) \in \mathbf{A}$ implies $\mathbf{T}p(c), \mathbf{T}q(c) \in \mathbf{A}_i$ for some i.

For example, the assignment $\mathbf{A} = \{\mathbf{T}p(a), \mathbf{T}p(b), \mathbf{F}p(c), \mathbf{T}q(a), \mathbf{F}q(b), \mathbf{T}q(c)\}$ can be partitioned into $\mathbf{A}_1 = \{\mathbf{T}p(a), \mathbf{T}q(a)\}, \mathbf{A}_2 = \{\mathbf{T}p(b), \mathbf{F}q(b)\}, \mathbf{A}_3 = \{\mathbf{F}p(c), \mathbf{T}q(c)\}$. Obviously $ext(\mathbf{A}, \&diff[p, q]) = \{b\} = \bigcup_{i=1}^{3} ext(\mathbf{A}_i, \&diff[p, q])$. In contrast, the partitioning $\mathbf{A}'_1 = \{\mathbf{T}p(a)\}, \mathbf{A}'_2 = \{\mathbf{T}p(b), \mathbf{T}q(a), \mathbf{F}q(b), \mathbf{F}p(c), \mathbf{T}q(c)\}$ is illegal (i.e., $\&diff[p, q]$ is not linear wrt. it) because $\bigcup_{i=1}^{2} ext(\mathbf{A}'_i, \&diff[p, q]) = \{a\} \cup \{b\} \neq \{b\}$. □

Linearity often allows for learning more general nogoods. In the above example, the suggested partitioning allows for learning the nogood $\{\mathbf{T}p(b), \mathbf{F}q(b), \mathbf{F}e_{\&diff[p,q]}(b)\}$, while without exploiting linearity (but using monotonicity and antimonotonicity) the less general nogood $\left\{\mathbf{T}p(a), \mathbf{T}p(b), \mathbf{F}q(b), \mathbf{F}e_{\&diff[p,q]}(b)\right\}$ is learned.

Definition 36. The learning function for an external predicate $\&g$ with input list $\mathbf{y}$ in program Π, wrt. assignment $\mathbf{A}$ is defined as

$$\Lambda_l(\&g[\mathbf{y}], \mathbf{A}) = \bigcup_{i=1}^{n} \Lambda_m(\&g[\mathbf{y}], \mathbf{A}_i),$$

where $\mathbf{A}_1, \cdots, \mathbf{A}_n$ is a partitioning of $\mathbf{A}$ s.t. $\&g[\mathbf{y}]$ is linear wrt. it.

Lemma 3.4. *For all assignments* $\mathbf{A}$*, the nogoods* $\Lambda_l(\&g[\mathbf{y}], \mathbf{A})$ *in Definition 36 are correct wrt.* Π*.*

Proof. By assumption $\mathbf{A}_1, \cdots, \mathbf{A}_n$ is a partitioning of $\mathbf{A}$ s.t. $\&g[\mathbf{y}]$ is linear wrt. it. Hence, $ext(\mathbf{A}, \&g[\mathbf{y}]) = \bigcup_{i=1}^{n} ext(\mathbf{A}_i, \&g[\mathbf{y}])$. The correctness of $\Lambda_l(\&g[\mathbf{y}], \mathbf{A})$ follows then from correctness of $\Lambda_m(\&g[\mathbf{y}], \mathbf{A}_i)$ for all $1 \leq i \leq n$ (Lemma 3.2). □

Two special cases of linearity are linearity on the level of atoms and on the level of tuples. We say that $\&g[\mathbf{y}]$ is linear *on the level of atoms* if it is linear wrt. partitioning $\big\{\{l\} \mid l \in \mathbf{A}\big\}$ for all assignments $\mathbf{A}$, i.e., the assignment can be fully decomposed into assignments containing only one signed literal each. We say that $\&g[\mathbf{y}]$ is linear *on the level of tuples* if $\mathbf{y}_{pred} = p_1, \ldots, p_n$ are predicate input parameters, the arities of all p_i, $1 \leq i \leq n$ are the same, and it is linear wrt. partitioning $\big\{\{\mathbf{T}p_i(\mathbf{x}), \mathbf{F}p_i(\mathbf{x}) \in \mathbf{A} \mid 1 \leq i \leq n\} \mid \mathbf{x} \in S\big\}$ with $S = \{\mathbf{x} \mid \mathbf{T}p_i(\mathbf{x}) \in \mathbf{A}$ or $\mathbf{F}p_i(\mathbf{x}) \in \mathbf{A}$ for some $1 \leq i \leq n\}$ for all assignments $\mathbf{A}$, i.e., the assignment can be decomposed into multiple assignments containing all input atoms over the same argument tuple.

Example 23. A useful learning function for $\&diff[p, q](X)$ is the following: whenever an element is in p but not in q, it belongs to the output of the external atom. This function works elementwise and produces nogoods with three literals each, which models linearity on the level of tuples. In contrast, the naive learning function from the Section 3.1.2 includes the complete extensions of p and q in the nogoods, which are less general. □

Apart from linearity on the level of atoms or on the level of tuples, customized types of linearity may be used for specific external sources to decompose the query into smaller subqueries. In all cases the validity of the decomposition must be asserted by showing that the source is indeed linear in the respective sense.

User-Defined Learning. In many cases the developer of an external atom has more information about the internal behavior. This allows for defining more effective nogoods. It is therefore beneficial to give the user the possibility to customize learning functions. Currently, user-defined functions may either specify the learned nogoods directly, or by ASP-style rules (the details are discussed in Chapter 5).

Example 24. Consider the program

$$r(X,Y) \vee nr(X,Y) \leftarrow d(X), d(Y)$$
$$r(V,W) \leftarrow \&tc[r](V,W), d(V), d(W)$$

It guesses, for some set of nodes $d(X)$ of an undirected graph, all subgraphs of the complete graph. Suppose $\&tc[r]$ checks if the edge selection $r(X,Y)$ is transitively closed; if this is the case, the output is empty, otherwise the set of missing transitive edges is returned. For instance, if the extension of r is $\{(a,b),(b,c)\}$, then the output of $\&tc$ will be $\{(a,c)\}$, as this edge is missing in order to make the graph transitively closed. The second rule eliminates all subgraphs which are not transitively closed. Note that $\&tc$ is nonmonotonic. The guessing program is

$$r(X,Y) \vee nr(X,Y) \leftarrow d(X), d(Y)$$
$$r(V,W) \leftarrow e_{\&tc[r]}(V,W), d(V), d(W)$$
$$e_{\&tc[r]}(V,W) \vee ne_{\&tc[r]}(V,W) \leftarrow d(V), d(W)$$

The naive implementation guesses for n nodes all $2^{\frac{n(n-1)}{2}}$ subgraphs and checks the transitive closure for each of them, which is costly. Consider the domain $D = \{a, b, c, d, e, f\}$. After

checking one selection with $r(a,b), r(b,c), nr(a,c)$, we know that *no* selection containing these three atoms will be transitively closed. This can be formalized as a user-defined learning function. Suppose we have just checked our first guess $r(a,b), r(b,c)$, and $nr(x,y)$ for all other $(x,y) \in D \times D$. Compared to the nogood learned by the general learning function, the nogood $\{\mathbf{T}r(a,b), \mathbf{T}r(b,c), \mathbf{F}r(a,c), \mathbf{F}e_{\&tc[r]}(a,c)\}$ is a more general description of the conflict reason, containing only relevant edges. It is immediately violated and future guesses containing $\{\mathbf{T}r(a,b), \mathbf{T}r(b,c), \mathbf{F}r(a,c)\}$ are avoided. □

For user-defined learning, correctness of the learning function must be asserted.

Using Negative Information. Up to now we have learned positive facts about the output of external sources, i.e., the learned nogoods in $\Lambda(\&g[\mathbf{y}], \mathbf{A})$ encode under which conditions the atoms in $ext(\mathbf{A}, \&g[\mathbf{y}])$ become true.

However, there is no reason to restrict learning to positive examples. Signed literals of kind $\mathbf{T}e_{\&g[\mathbf{y}]}(\mathbf{x})$ can be used to encode that a certain ground external atom must *not* be true, i.e. tuple $(\mathbf{x})$ is not in the output of the external source. Since we could in principle learn arbitrary many negative facts, an important question is which tuples $\mathbf{x}$ to consider. We call this the *scope* S of tuples. Clearly, it is unnecessary to consider tuples which do not occur in the program. Our current implementation considers all tuples which were previously wrongly guessed to true.

Definition 37. The negative learning function for an external predicate $\&g$ with input list $\mathbf{y}$ in program Π wrt. assignment $\mathbf{A}$, such that $\&g$ is monotonic in predicate input parameters $\mathbf{p}_m \subseteq \mathbf{y}$ and antimonotonic in predicate input parameters $\mathbf{p}_a \subseteq \mathbf{y}$, and a finite set of tuples S (*scope*) is defined as follows:

$$\Lambda_\neg(\&g[\mathbf{y}], \mathbf{A}) = \left\{ \begin{array}{l} \{\mathbf{F}a \in \mathbf{A}|_{\mathbf{p}_m}\} \cup \{\mathbf{T}a \in \mathbf{A}|_{\mathbf{p}_a}\} \cup \\ \mathbf{A}|_{\mathbf{p}_n} \cup \{\mathbf{T}e_{\&g[\mathbf{y}]}(\mathbf{x})\} \end{array} \;\middle|\; \mathbf{x} \in S, \mathbf{x} \notin ext(\&g[\mathbf{y}], \mathbf{A}) \right\}$$

Lemma 3.5. *For all assignments* $\mathbf{A}$*, the nogoods* $\Lambda_\neg(\&g[\mathbf{y}], \mathbf{A})$ *in Definition 37 are correct wrt.* Π.

Proof. We first focus on external sources without monotonic or antimonotonic input parameters. Then a nogood encodes that $e_{\&g[\mathbf{y}]}(\mathbf{x})$ must not be true if $(\mathbf{x}) \notin ext(\&g[\mathbf{y}], \mathbf{A})$ and the external atom input is as in $\mathbf{A}$. Such nogoods cannot eliminate compatible sets, because if $e_{\&g[\mathbf{y}]}(\mathbf{x})$ would be true, the assignment would not pass the compatibility check anyway.

We now show that positive input literals over monotonic parameters and negative input literals over antimonotonic parameters can be removed from the learned nogoods without affecting correctness. Above, we argued that for output tuple $\mathbf{x} \notin ext(\&g[\mathbf{y}], \mathbf{A})$, the replacement atom $e_{\&g[\mathbf{y}]}(\mathbf{x})$ must not be guessed true if the input to $\&g[\mathbf{y}](\mathbf{x})$ is $\mathbf{A}|_{\mathbf{y}_{pred}}$. However, as the output of $\&g$ shrinks with growing (shrinking) extension of a antimonotonic (monotonic) parameter $p \in \mathbf{p}_m$ ($p \in \mathbf{p}_a$), the same applies for any $\mathbf{A}'$ in which the extension of p is larger (smaller), i.e., $\{\mathbf{T}a \in \mathbf{A}'|_p\} \supseteq \{\mathbf{T}a \in \mathbf{A}|_p\}$ ($\{\mathbf{T}a \in \mathbf{A}'|_p\} \subseteq \{\mathbf{T}a \in \mathbf{A}|_p\}$) and consequently $\{\mathbf{F}a \in \mathbf{A}'|_p\} \subseteq \{\mathbf{F}a \in \mathbf{A}|_p\}$ ($\{\mathbf{F}a \in \mathbf{A}'|_p\} \supseteq \{\mathbf{F}a \in \mathbf{A}|_p\}$). Hence, the negative literals over antimonotonic parameters and the positive literals over monotonic parameters are not relevant wrt. non-output tuple $\mathbf{x}$ and can be removed from the nogood. □

3.2 Minimality Check

We now turn to the last statement of Algorithm GuessAndCheckHexEvaluation, which is the minimality check, due to the use of the FLP-reduct subsequently also called the *FLP check*. This check identifies assignments $\mathbf{A}$ extracted from compatible sets $\hat{\mathbf{A}}$ of a program Π that are also answer sets of Π, i.e., subset-minimal models of $f\Pi^{\mathbf{A}}$. It appears that in many practical applications most assignments extracted from compatible sets $\hat{\mathbf{A}}$ pass the FLP check. Moreover, this check is computationally costly: in a naive realization all models of $f\Pi^{\mathbf{A}}$ must be enumerated, along with calls to the external sources to ensure compatibility. Even worse, as one needs to search for a smaller *model* and not just for a smaller compatible set, $f\Pi^{\mathbf{A}}$ usually has even more models then the original program. The *explicit FLP check (explicit minimality check)* corresponds to the search for compatible sets of the following program:

$$\begin{aligned} \mathit{Check}(\Pi, \mathbf{A}) = f\hat{\Pi}^{\hat{\mathbf{A}}} \cup \big\{ \leftarrow a \mid a \in A(\Pi), \mathbf{T}a \notin \hat{\mathbf{A}} \big\} \cup \big\{ a \vee a' \leftarrow . \mid \mathbf{T}a \in \hat{\mathbf{A}} \big\} \\ \cup \big\{ \leftarrow \text{not } \mathit{smaller} \big\} \cup \big\{ \mathit{smaller} \leftarrow \text{not } a \mid a \in A(\Pi), \mathbf{T}a \in \hat{\mathbf{A}} \big\} \end{aligned}$$

It consists of the reduct $f\hat{\Pi}^{\hat{\mathbf{A}}}$ and rules that restrict the search to proper sub-interpretations of $\hat{\mathbf{A}}$, where $\mathit{smaller}$ is a new atom. Moreover, as one actually needs to search for models and not just compatible sets, rules of the form $a \vee a' \leftarrow$ (where a' is a new atom for each $\mathbf{T}a \in \hat{\mathbf{A}}$) make sure that atoms can be arbitrarily true without having a justifying rule in Π.

Proposition 3.4. *Let $\mathbf{A}$ be an interpretation extracted from a compatible set $\hat{\mathbf{A}}$ of a program Π. Program $\mathit{Check}(\Pi, \mathbf{A})$ has an answer set $\mathbf{A}'$ such that $f_{\&g}(\mathbf{A}', \boldsymbol{y}, \boldsymbol{x}) = 1$ iff $\mathbf{T}e_{\&g[\mathbf{y}]}(\boldsymbol{x}) \in \mathbf{A}'$ for all external atoms $\&g[\boldsymbol{y}](\boldsymbol{x})$ in Π, if and only if $\mathbf{A}$ is not an answer set of Π.*

Proof. ($\Rightarrow$) Let $\mathbf{A}'$ be an answer set of program $\mathit{Check}(\Pi, \mathbf{A})$ such that $f_{\&g}(\mathbf{A}', \mathbf{y}, \mathbf{x}) = 1$ iff $\mathbf{T}e_{\&g[\mathbf{y}]}(\mathbf{x}) \in \mathbf{A}'$ for all external atoms $\&g[\mathbf{y}](\mathbf{x})$ in Π.

Since $\hat{\mathbf{A}}$ is a compatible set of Π, $f_{\&g}(\mathbf{A}, \mathbf{y}, \mathbf{x}) = 1$ iff $\mathbf{T}e_{\&g[\mathbf{y}]}(\mathbf{x}) \in \hat{\mathbf{A}}$ for all external atoms $\&g[\mathbf{y}](\mathbf{x})$ in Π. Thus, $f\hat{\Pi}^{\hat{\mathbf{A}}}$ is the same as $f\Pi^{\mathbf{A}}$ with replacement atoms in place of external atoms, and with additional guessing rules for replacement atoms. Since $\mathbf{A}'$ is a model of $\mathit{Check}(\Pi, \mathbf{A})$ it is also a model of $f\hat{\Pi}^{\hat{\mathbf{A}}}$. Let $\mathbf{A}'' = \{\mathbf{T}a \in \mathbf{A}' \mid a \in A(\Pi)\} \cup \{\mathbf{F}a \in \mathbf{A}' \mid a \in A(\Pi)\}$. Since $f_{\&g}(\mathbf{A}', \mathbf{y}, \mathbf{x}) = f_{\&g}(\mathbf{A}'', \mathbf{y}, \mathbf{x}) = 1$ iff $\mathbf{T}e_{\&g[\mathbf{y}]}(\mathbf{x}) \in \mathbf{A}'$ for all external atoms $\&g[\mathbf{y}](\mathbf{x})$ in Π by assumption, $\mathbf{A}''$ is a model of $f\Pi^{\mathbf{A}}$.

Since $\mathbf{A}'$ is an answer set of $\mathit{Check}(\Pi, \mathbf{A})$, and $\leftarrow a \in \mathit{Check}(\Pi, \mathbf{A})$ for all $a \in A(\Pi)$ with $\mathbf{T}a \notin \hat{\mathbf{A}}$ (and thus $\mathbf{T}a \notin \mathbf{A}$), we have $\{\mathbf{T}a \in \mathbf{A}'' \mid a \in A(\Pi)\} \subseteq \mathbf{A}$. Finally, due to $\{\leftarrow \text{not } \mathit{smaller}\} \cup \{\mathit{smaller} \leftarrow \text{not } a \mid a \in A(\Pi), \mathbf{T}a \in \hat{\mathbf{A}}\} \in \mathit{Check}(\Pi, \mathbf{A})$, there is at least one $a \in A(\Pi)$ s.t. $\mathbf{T}a \in \hat{\mathbf{A}}$ (and thus also $\mathbf{T}a \in \mathbf{A}$), but $\mathbf{F}a \in \mathbf{A}'$ (and thus also $\mathbf{F}a \in \mathbf{A}''$). Therefore $\{\mathbf{T}a \in \mathbf{A}'' \mid a \in A(\Pi)\} \subsetneq \mathbf{A}$ is a model of Π, and thus $\mathbf{A}$ is not an answer set of Π.

($\Leftarrow$) If $\mathbf{A}$ is not an answer set of Π, then there is a model $\mathbf{A}''$ of $f\Pi^{\mathbf{A}}$ which is smaller in the positive part, i.e., $\{\mathbf{T}a \in \mathbf{A}''\} \subsetneq \{\mathbf{T}a \in \mathbf{A}\}$.

Let

$$\mathbf{A}' = \kappa(\Pi, \mathbf{A}'') \cup \{\mathbf{T}a' \mid \mathbf{T}a \in \hat{\mathbf{A}}, \mathbf{F}a \in \mathbf{A}''\} \cup \{\mathbf{F}a' \mid \mathbf{T}a \in \hat{\mathbf{A}}, \mathbf{T}a \in \mathbf{A}''\} \cup \{\mathbf{T}\mathit{smaller}\}.$$

We show that $\mathbf{A}'$ is an answer set of $\mathit{Check}(\Pi, \mathbf{A})$ such that $f_{\&g}(\mathbf{A}', \mathbf{y}, \mathbf{x}) = 1$ iff $\mathbf{T}e_{\&g[\mathbf{y}]}(\mathbf{x}) \in \mathbf{A}'$ for all external atoms $\&g[\mathbf{y}](\mathbf{x})$ in Π.

Since $\mathbf{A}$ has been extracted from a compatible set $\hat{\mathbf{A}}$ of Π, $f\hat{\Pi}^{\hat{\mathbf{A}}}$ is the same as $f\Pi^{\mathbf{A}}$ with replacement atoms in place of external atoms, and with additional guessing rules for replacement atoms. Since $\mathbf{A}''$ is a model of $f\Pi^{\mathbf{A}}$, the truth values of all replacement atoms in $\mathbf{A}'$ coincide with the oracle functions by definition of $\kappa(\Pi, \mathbf{A}'')$, and exactly one of e_a or ne_a for each external atom a in Π is set to true (and thus the guessing rules for replacement atoms are satisfied), $\mathbf{A}'$ is a model of $f\hat{\Pi}^{\hat{\mathbf{A}}}$. Since $\{\mathbf{T}a \in \mathbf{A}''\} \subsetneq \{\mathbf{T}a \in \mathbf{A}\}$ and thus also $\{\mathbf{T}a \in \mathbf{A}' \mid a \in A(\Pi)\} \subsetneq \{\mathbf{T}a \in \hat{\mathbf{A}}\}$, no constraint of type $\leftarrow a$ in $\mathit{Check}(\Pi, \mathbf{A})$ with $a \in A(\Pi), \mathbf{T}a \notin \hat{\mathbf{A}}$ is violated. Moreover, for each a with $\mathbf{T}a \in \hat{\mathbf{A}}$ we have either $\mathbf{T}a \in \mathbf{A}'$ or $\mathbf{T}a' \in \mathbf{A}'$, thus the corresponding rule $a \vee a' \leftarrow$ in $\mathit{Check}(\Pi, \mathbf{A})$ is satisfied. Finally, since $\mathbf{T}\mathit{smaller} \in \mathbf{A}'$, the rules $\{\mathit{smaller} \leftarrow \mathrm{not}\, a \mid a \in A(\Pi), \mathbf{T}a \in \hat{\mathbf{A}}\}$ are satisfied and the constraint $\leftarrow \mathrm{not}\, \mathit{smaller}$ does not fire. Thus $\mathbf{A}'$ is a model of $\mathit{Check}(\Pi, \mathbf{A})$.

We show now that $\mathbf{A}'$ is also a subset-minimal model of $f\,\mathit{Check}(\Pi, \mathbf{A})^{\mathbf{A}'}$. Observe that

$$f\,\mathit{Check}(\Pi, \mathbf{A})^{\mathbf{A}'} = f\hat{\Pi}^{\hat{\mathbf{A}}} \cup \big\{a \vee a' \leftarrow . \mid \mathbf{T}a \in \hat{\mathbf{A}}\big\} \cup \big\{\mathit{smaller} \leftarrow \mathrm{not}\, a \mid a \in A(\Pi), \mathbf{T}a \in \hat{\mathbf{A}}\big\}.$$

However, if an atom $a \in A(f\,\mathit{Check}(\Pi, \mathbf{A})^{\mathbf{A}'})$ with $\mathbf{T}a \in \mathbf{A}'$ is changed to false, then the interpretation is not a model anymore because the corresponding rule $a \vee a' \leftarrow$ remains unsatisfied since only one of a and a' is true in $\mathbf{A}'$ by definition, thus no interpretation which is smaller in the positive part than $\mathbf{A}'$ can be a model of $f\,\mathit{Check}(\Pi, \mathbf{A})^{\mathbf{A}'}$, thus $\mathbf{A}'$ is an answer set of $\mathit{Check}(\Pi, \mathbf{A})$.

Finally, $f_{\&g}(\mathbf{A}', \mathbf{y}, \mathbf{x}) = 1$ iff $\mathbf{T}e_{\&g[\mathbf{y}]}(\mathbf{x}) \in \mathbf{A}'$ for all external atoms $\&g[\mathbf{y}](\mathbf{x})$ in Π by definition of $\kappa(\Pi, \mathbf{A}'')$. □

Example 25 (ctd.)**.** Consider the program $\Pi = \{p \leftarrow \&id[q](); q \leftarrow p\}$ Then the corresponding guessing program is $\hat{\Pi} = \{p \leftarrow e_{\&id[q]}(); q \leftarrow p; e_{\&id[q]}() \vee ne_{\&id[q]}() \leftarrow\}$ and yields the compatible sets $\hat{\mathbf{A}}_1 = \{\mathbf{F}p, \mathbf{F}q, \mathbf{F}e_{\&id[p]}\}$ and $\hat{\mathbf{A}}_2 = \big\{\mathbf{T}p, \mathbf{T}q, \mathbf{T}e_{\&id[p]}\big\}$. While $\mathbf{A}_1 = \{\mathbf{F}p, \mathbf{F}q\}$ is also a $\leq$-minimal model of $f\Pi^{\mathbf{A}_1} = \emptyset$, $\mathbf{A}_2 = \{\mathbf{T}p, \mathbf{T}q\}$ is not a $\leq$-minimal model of $f\Pi^{\mathbf{A}_2} = \Pi$. Indeed, the program

$$\begin{aligned}\mathit{Check}(\Pi, \mathbf{A}_2) = \hat{\Pi} &\cup \big\{p \vee p' \leftarrow; q \vee q' \leftarrow; e_{\&id[q]}() \vee e'_{\&id[q]}() \leftarrow\big\} \\ &\cup \big\{\leftarrow \mathrm{not}\, \mathit{smaller}\big\} \\ &\cup \big\{\mathit{smaller} \leftarrow \mathrm{not}\, p; \mathit{smaller} \leftarrow \mathrm{not}\, q\big\} \\ &\cup \big\{\mathit{smaller} \leftarrow \mathrm{not}\, e_{\&id[q]}()\big\}\end{aligned}$$

has the answer set $\mathbf{A}' = \{\mathbf{F}p, \mathbf{T}p', \mathbf{F}q, \mathbf{T}q', \mathbf{F}e_{\&id[q]}(), \mathbf{T}ne_{\&id[q]}(), \mathbf{T}e'_{\&id[q]}(), \mathbf{T}\mathit{smaller}\}$ and $f_{\&id}(\mathbf{A}', q, \epsilon) = 0$ (where ϵ denotes the empty output list) and $\mathbf{F}e_{\&id[q]}() \in \mathbf{A}'$. □

Because of the guessing rules $a \vee a' \leftarrow$ for all a with $\mathbf{T}a \in \hat{\mathbf{A}}$, the rules in the reduct $f\hat{\Pi}^{\hat{\mathbf{A}}}$, except for the guesses on replacement atoms, can be rewritten to constraints. This might be more more efficient.

We define

$$\mathit{CheckOpt}(\Pi, \mathbf{A}) = \bar{f}\hat{\Pi}^{\hat{\mathbf{A}}} \cup \big\{ \leftarrow a \mid a \in A(\Pi), \mathbf{T}a \notin \hat{\mathbf{A}} \big\} \cup \big\{ a \vee a' \leftarrow . \mid \mathbf{T}a \in \hat{\mathbf{A}} \big\} \\ \cup \big\{ \leftarrow \text{not } \mathit{smaller} \big\} \cup \big\{ \mathit{smaller} \leftarrow \text{not } a \mid a \in A(\Pi), \mathbf{T}a \in \hat{\mathbf{A}} \big\},$$

where $\bar{f}\hat{\Pi}^{\hat{\mathbf{A}}}$ denotes the FLP-reduct of $\hat{\Pi}$ wrt. interpretation $\hat{\mathbf{A}}$ with each rule (cf. Definition 6) except guessing rules for replacement atoms being rewritten to

$$\leftarrow \text{not } a_1, \ldots, \text{not } a_k, b_1, \ldots, b_m, \text{not } b_{m+1}, \ldots, \text{not } b_n.$$

Proposition 3.5. *Let* $\mathbf{A}$ *be an interpretation extracted from a compatible set* $\hat{\mathbf{A}}$ *of a program* Π. *Program* $\mathit{CheckOpt}(\Pi, \mathbf{A})$ *has an answer set* $\mathbf{A}'$ *such that* $f_{\&g}(\mathbf{A}', \boldsymbol{y}, \boldsymbol{x}) = 1$ *iff* $\mathbf{T}e_{\&g[\mathbf{y}]}(\boldsymbol{x}) \in \mathbf{A}'$ *for all external atoms* $\&g[\boldsymbol{y}](\boldsymbol{x})$ *in* Π*, if and only if* $\mathbf{A}$ *is not an answer set of* Π.

Proof. The proof is very similar to the one of Proposition 3.4.

($\Rightarrow$) Let $\mathbf{A}'$ be an answer set of program $\mathit{CheckOpt}(\Pi, \mathbf{A})$ such that $f_{\&g}(\mathbf{A}', \mathbf{y}, \mathbf{x}) = 1$ iff $\mathbf{T}e_{\&g[\mathbf{y}]}(\mathbf{x}) \in \mathbf{A}'$ for all external atoms $\&g[\mathbf{y}](\mathbf{x})$ in Π.

Since $\hat{\mathbf{A}}$ is a compatible set of Π, $f_{\&g}(\mathbf{A}, \mathbf{y}, \mathbf{x}) = 1$ iff $\mathbf{T}e_{\&g[\mathbf{y}]}(\mathbf{x}) \in \hat{\mathbf{A}}$ for all external atoms $\&g[\mathbf{y}](\mathbf{x})$ in Π. Thus, $\bar{f}\hat{\Pi}^{\hat{\mathbf{A}}}$ is the same as $\bar{f}\Pi^{\mathbf{A}}$ with replacement atoms in place of external atoms, and with additional guessing rules for replacement atoms. Since $\mathbf{A}'$ is a model of $\mathit{CheckOpt}(\Pi, \mathbf{A})$ it is also a model of $\bar{f}\hat{\Pi}^{\hat{\mathbf{A}}}$. Let $\mathbf{A}'' = \{\mathbf{T}a \in \mathbf{A}' \mid a \in A(\Pi)\} \cup \{\mathbf{F}a \in \mathbf{A}' \mid a \in A(\Pi)\}$. Since $f_{\&g}(\mathbf{A}', \mathbf{y}, \mathbf{x}) = 1$ iff $\mathbf{T}e_{\&g[\mathbf{y}]}(\mathbf{x}) \in \mathbf{A}'$ for all external atoms $\&g[\mathbf{y}](\mathbf{x})$ in Π by assumption, $\mathbf{A}''$ is a model of $\bar{f}\Pi^{\mathbf{A}}$. But then it is also a model of $f\Pi^{\mathbf{A}}$.

Since $\mathbf{A}'$ is an answer set of $\mathit{CheckOpt}(\Pi, \mathbf{A})$, and $\leftarrow a \in \mathit{CheckOpt}(\Pi, \mathbf{A})$ for all $a \in A(\Pi)$ with $\mathbf{T}a \notin \hat{\mathbf{A}}$ (and thus $\mathbf{T}a \notin \mathbf{A}$), we have $\{\mathbf{T}a \in \mathbf{A}'' \mid a \in A(\Pi)\} \subseteq \mathbf{A}$. Finally, due to $\{\leftarrow \text{not } \mathit{smaller}\} \cup \{\mathit{smaller} \leftarrow \text{not } a \mid a \in A(\Pi), \mathbf{T}a \in \hat{\mathbf{A}}\} \in \mathit{CheckOpt}(\Pi, \mathbf{A})$, there is at least one $a \in A(\Pi)$ s.t. $\mathbf{T}a \in \hat{\mathbf{A}}$ (and thus also $\mathbf{T}a \in \mathbf{A}$), but $\mathbf{F}a \in \mathbf{A}'$ (and thus also $\mathbf{F}a \in \mathbf{A}''$). Therefore $\{\mathbf{T}a \in \mathbf{A}'' \mid a \in A(\Pi)\} \subsetneq \mathbf{A}$ is a model of Π, and thus $\mathbf{A}$ is not an answer set of Π.

($\Leftarrow$) If $\mathbf{A}$ is not an answer set of Π, then there is a model $\mathbf{A}''$ of $f\Pi^{\mathbf{A}}$ which is smaller in the positive part, i.e., $\{\mathbf{T}a \in \mathbf{A}''\} \subsetneq \{\mathbf{T}a \in \mathbf{A}\}$.

Let

$$\mathbf{A}' = \kappa(\Pi, \mathbf{A}'') \cup \{\mathbf{T}a' \mid \mathbf{T}a \in \hat{\mathbf{A}}, \mathbf{F}a \in \mathbf{A}''\} \cup \{\mathbf{F}a' \mid \mathbf{T}a \in \hat{\mathbf{A}}, \mathbf{T}a \in \mathbf{A}''\} \cup \{\mathbf{T}\mathit{smaller}\}.$$

We show that $\mathbf{A}'$ is an answer set of program $\mathit{CheckOpt}(\Pi, \mathbf{A})$ such that $f_{\&g}(\mathbf{A}', \mathbf{y}, \mathbf{x}) = 1$ iff $\mathbf{T}e_{\&g[\mathbf{y}]}(\mathbf{x}) \in \mathbf{A}'$ for all external atoms $\&g[\mathbf{y}](\mathbf{x})$ in Π.

Since $\mathbf{A}$ has been extracted from a compatible set $\hat{\mathbf{A}}$ of Π, $f\hat{\Pi}^{\hat{\mathbf{A}}}$ is the same as $f\Pi^{\mathbf{A}}$ with replacement atoms in place of external atoms, and with additional guessing rules for replacement atoms. Since $\mathbf{A}''$ is a model of $f\Pi^{\mathbf{A}}$, and the truth values of all replacement atoms in $\mathbf{A}'$ coincide with the oracle functions by definition of $\kappa(\Pi, \mathbf{A}'')$, and exactly one of e_a or ne_a for each external atom a in Π is set to true (and thus the guessing rules for replacement atoms are satisfied), $\mathbf{A}'$ is a model of $f\hat{\Pi}^{\hat{\mathbf{A}}}$. But then it is also a model of $\bar{f}\hat{\Pi}^{\hat{\mathbf{A}}}$. Since $\{\mathbf{T}a \in \mathbf{A}''\} \subsetneq \{\mathbf{T}a \in \mathbf{A}\}$ and thus also $\{\mathbf{T}a \in \mathbf{A}' \mid a \in A(\Pi)\} \subsetneq \{\mathbf{T}a \in \mathbf{A} \mid a \in A(\Pi)\}$, no constraint

of type $\leftarrow a$ in $\mathit{CheckOpt}(\Pi, \mathbf{A})$ with $a \in A(\Pi), \mathbf{T}a \notin \hat{\mathbf{A}}$ is violated. Moreover, for each a with $\mathbf{T}a \in \hat{\mathbf{A}}$ we have either $\mathbf{T}a \in \mathbf{A}'$ or $\mathbf{T}a' \in \mathbf{A}'$, thus the corresponding rule $a \vee a' \leftarrow$ in $\mathit{CheckOpt}(\Pi, \mathbf{A})$ is satisfied. Finally, since $\mathbf{T}\mathit{smaller} \in \mathbf{A}'$, the rules $\{\mathit{smaller} \leftarrow \mathrm{not}\, a \mid a \in A(\Pi), \mathbf{T}a \in \hat{\mathbf{A}}\}$ are satisfied and the constraint $\leftarrow \mathrm{not}\, \mathit{smaller}$ does not fire. Thus $\mathbf{A}'$ is a model of $\mathit{CheckOpt}(\Pi, \mathbf{A})$.

We show now that $\mathbf{A}'$ is also a subset-minimal model of $\bar{f}\mathit{Check}(\Pi, \mathbf{A})^{\mathbf{A}'}$. Observe that

$$\bar{f}\mathit{CheckOpt}(\Pi, \mathbf{A})^{\mathbf{A}'} = \bar{f}\hat{\Pi}^{\hat{\mathbf{A}}} \cup \{a \vee a' \leftarrow . \mid \mathbf{T}a \in \hat{\mathbf{A}}\} \cup \{\mathit{smaller} \leftarrow \mathrm{not}\, a \mid a \in A(\Pi), \mathbf{T}a \in \hat{\mathbf{A}}\}.$$

However, if an atom $a \in A(\bar{f}\mathit{CheckOpt}(\Pi, \mathbf{A})^{\mathbf{A}'})$ with $\mathbf{T}a \in \mathbf{A}'$ is changed to false, then the interpretation is not a model anymore because the corresponding rule $a \vee a' \leftarrow$ remains unsatisfied since only one a and a' is true in $\mathbf{A}'$ by definition, thus no interpretation which is smaller in the positive part than $\mathbf{A}'$ can be a model of $f\,\mathit{OptCheck}(\Pi, \mathbf{A})^{\mathbf{A}'}$, thus $\mathbf{A}'$ is an answer set of $\mathit{OptCheck}(\Pi, \mathbf{A})$.

Finally, $f_{\&g}(\mathbf{A}', \mathbf{y}, \mathbf{x}) = 1$ iff $\mathbf{T}e_{\&g[\mathbf{y}]}(\mathbf{x}) \in \mathbf{A}'$ for all external atoms $\&g[\mathbf{y}](\mathbf{x})$ in Π by definition of $\kappa(\Pi, \mathbf{A}'')$. □

Our benchmarks in Chapter 5 use this optimized version of the explicit check.

Next, we present a novel FLP check algorithm based on unfounded sets (UFS). Instead of explicitly searching for smaller models of the reduct, we check if the candidate answer set is *unfounded-free* (see below), which implies that it is an answer set [Faber, 2005]. The unfounded set-based check can be realized as a post-check (i.e., it is carried out only after the interpretation has been completed), or also wrt. partial assignments (thus interleaving it with the main search for compatible sets). We realized both, but our benchmarks show that unfounded set checking wrt. partial assignments is counterproductive, roughly because the unfounded set check is too expensive and should be done rarely. We use unfounded sets for logic programs as introduced by Faber (2005) for programs with arbitrary aggregates.

Definition 38 (Unfounded Set)**.** Given a program Π and an assignment $\mathbf{A}$, let U be any set of ordinary ground atoms appearing in Π. Then, U is an *unfounded set for* Π *wrt.* $\mathbf{A}$ if, for each rule r having some atoms from U in the head, at least one of the following conditions holds, where $\mathbf{A} \dot\cup \neg.U = \big(\mathbf{A} \setminus \{\mathbf{T}a \mid a \in U\}\big) \cup \{\mathbf{F}a \mid a \in U\}$:

(i) some literal of $B(r)$ is false wrt. $\mathbf{A}$; or

(ii) some literal of $B(r)$ is false wrt. $\mathbf{A} \dot\cup \neg.U$; or

(iii) some atom of $H(r) \setminus U$ is true wrt. $\mathbf{A}$.

Intuitively, an unfounded set U is a set of atoms for which no rule can be used to justify that any of the atoms in U is true, because all rules are already satisfied independently of U.

Answer sets can then be characterized in terms of unfounded sets (corresponds to Corollary 3 by Faber (2005)).

Definition 39 (Unfounded-Free Interpretations)**.** An interpretation $\mathbf{A}$ of a program Π is *unfounded-free* (in Π) if $\mathbf{A}^{\mathbf{T}} \cap U = \emptyset$, for all unfounded sets U of Π wrt. $\mathbf{A}$.

Theorem 2. *A model* $\mathbf{A}$ *of a program* Π *is an answer set iff it is* unfounded-free *(in* Π*).*

Proof. See Appendix B, page 219.

Example 26. Consider the program $\Pi = \{p \leftarrow \&id[p]()\}$ and $\mathbf{A} = \{\mathbf{T}p\}$. Then $U = \{p\}$ is an unfounded set since U intersects with the head of $p \leftarrow \&id[p]()$ and $\mathbf{A} \dot\cup \neg.U \not\models \&id[p]()$. Therefore $\mathbf{A}$ is not unfounded-free and not an answer set. □

3.2.1 Basic Encoding of the Unfounded Set Search

We realize the search for unfounded sets using nogoods, i.e., for a given Π and an assignment $\mathbf{A}$ we construct a set of nogoods, such that solutions to this set correspond to (potential) unfounded sets. We then use a SAT solver to search for such unfounded sets.

Our encoding of the unfounded set detection is related to the one of Drescher et al. (2008) but respects external atoms. It uses a set $\Gamma_{\Pi,\mathbf{A}} = \Gamma^N_{\Pi,\mathbf{A}} \cup \Gamma^O_{\Pi,\mathbf{A}}$, of nogoods where $\Gamma^N_{\Pi,\mathbf{A}}$ contains all *necessary* constraints and $\Gamma^O_{\Pi,\mathbf{A}}$ are *optional* optimization nogoods that prune irrelevant parts of the search space. The idea is that the set of ordinary atoms which are true in a solution to $\Gamma_{\Pi,\mathbf{A}}$ represents a (potential) unfounded set U of Π wrt. $\mathbf{A}$, while the external replacement atoms encode the truth values of the corresponding external atoms wrt. $\mathbf{A} \dot\cup \neg.U$.

Let $B^+_o(r)$ be the subset of $B^+(r)$ consisting of all ordinary atoms except external replacement atoms, and $B_e(r)$ the subset of $B(r)$ consisting of all external replacement literals. Then, the nogood set $\Gamma_{\Pi,\mathbf{A}}$ is built over atoms $A(\Gamma_{\Pi,\mathbf{A}}) = A(\hat\Pi) \cup \{h_r, l_r \mid r \in \Pi\}$, where h_r, and l_r are new additional atoms for every rule r in Π. The *necessary part* $\Gamma^N_{\Pi,\mathbf{A}} = \big\{\{\mathbf{F}a \mid \mathbf{T}a \in \mathbf{A}\}\big\} \cup \big(\bigcup_{r\in\Pi} \Gamma^R_{r,\mathbf{A}}\big)$ consists of a nogood $\{\mathbf{F}a \mid \mathbf{T}a \in \mathbf{A}\}$, eliminating all unfounded sets that do not intersect with true atoms in $\mathbf{A}$, as well as nogoods $\Gamma^R_{r,\mathbf{A}}$ for every $r \in \Pi$. The latter consist of a *head criterion* $\Gamma^H_{r,\mathbf{A}}$ and a *conditional part* $\Gamma^C_{r,\mathbf{A}}$ for each rule, defined by:

- $\Gamma^R_{r,\mathbf{A}} = \Gamma^H_{r,\mathbf{A}} \cup \Gamma^C_{r,\mathbf{A}}$, where
- $\Gamma^H_{r,\mathbf{A}} = \big\{\{\mathbf{T}h_r\} \cup \{\mathbf{F}h \mid h \in H(r)\}\big\} \cup \big\{\{\mathbf{F}h_r, \mathbf{T}h\} \mid h \in H(r)\big\}$
  encodes that h_r is true for a rule r iff some atom of $H(r)$ is in the unfounded set; and
- $\Gamma^C_{r,\mathbf{A}} = \begin{cases} \big\{\{\mathbf{T}h_r\} \cup \\ \quad \{\mathbf{F}a \mid a \in B^+_o(r), \mathbf{A} \models a\} \cup \{\mathbf{t}a \mid a \in B_e(\hat r)\} \cup \\ \quad \{\mathbf{T}h \mid h \in H(r), \mathbf{A} \models h\}\big\} & \text{if } \mathbf{A} \models B(r), \\ \emptyset & \text{otherwise} \end{cases}$
  encodes that Condition (i), (ii) or (iii) of Definition 38 must hold if h_r is true.

More specifically, for an unfounded set U and a rule r with $H(r) \cap U \neq \emptyset$ (h_r is true) it must not happen that $\mathbf{A} \models B(r)$ (Condition (i) fails), no $a \in B^+_o(r)$ with $\mathbf{A} \models a$ is in the unfounded set and all $a \in B_e(\hat r)$ are true wrt. $\mathbf{A} \dot\cup \neg.U$ (Condition (ii) fails), and all $h \in H(r)$ with $\mathbf{A} \models h$ are in the unfounded set (Condition (iii) fails).

Example 27. Consider $\Pi = \{r_1 \colon\ p \leftarrow \mathit{\&id}[p]()\}$ and the compatible set $\hat{\mathbf{A}} = \{\mathbf{T}p, \mathbf{T}e_{\mathit{\&id}[p]}\}$. The nogood set $\Gamma^N_{\Pi,\mathbf{A}}$ is $\big\{\{\mathbf{T}h_{r_1}, \mathbf{F}p\}, \{\mathbf{F}h_{r_1}, \mathbf{T}p\}, \{\mathbf{T}h_{r_1}, \mathbf{T}e_{\mathit{\&id}[p]}(), \mathbf{T}p\}\big\}$. □

Towards computing unfounded sets, observe that they can be extended to solutions to the set of nogoods $\Gamma_{\Pi,\mathbf{A}}$ over $A(\Gamma_{\Pi,\mathbf{A}})$. Conversely, the solutions to $\Gamma_{\Pi,\mathbf{A}}$ include specific extensions of all unfounded sets, characterized by *induced assignments*: That is, by assigning true to all atoms in U, to all h_r such that $H(r)$ intersects with U, and to all external replacement atoms $e_{\mathit{\&g}[\mathbf{y}]}(\mathbf{x})$ such that $\mathit{\&g}[\mathbf{y}](\mathbf{x})$ is true wrt. $\mathbf{A} \dot\cup \neg.U$, and assigning false to all other atoms in $A(\Gamma_{\Pi,\mathbf{A}})$. More formally, we define:

Definition 40 (Induced Assignment of an Unfounded Set wrt. $\Gamma_{\Pi,\mathbf{A}}$)**.** Let U be an unfounded set of a program Π wrt. assignment $\mathbf{A}$. The *assignment induced by U wrt.* $\Gamma_{\Pi,\mathbf{A}}$, denoted $I_\Gamma(U, \Gamma_{\Pi,\mathbf{A}}, \Pi, \mathbf{A})$, is

$$I_\Gamma(U, \Gamma_{\Pi,\mathbf{A}}, \Pi, \mathbf{A}) = I^0_\Gamma(U, \Pi, \mathbf{A}) \cup \big\{\mathbf{F}a \mid a \in A(\Gamma_{\Pi,\mathbf{A}}), \mathbf{T}a \notin I^0_\Gamma(U, \Pi, \mathbf{A})\big\}, \text{ where}$$
$$\begin{aligned} I^0_\Gamma(U, \Pi, \mathbf{A}) = \ & \big\{\mathbf{T}a \mid a \in U\big\} \cup \big\{\mathbf{T}h_r \mid r \in \Pi, H(r) \cap U \neq \emptyset\big\} \cup \\ & \big\{\mathbf{T}e_{\mathit{\&g}[\mathbf{y}]}(\mathbf{x}) \mid \mathit{\&g}[\mathbf{y}](\mathbf{x}) \in EA(\Pi), \mathbf{A} \dot\cup \neg.U \models \mathit{\&g}[\mathbf{y}](\mathbf{x})\big\}. \end{aligned}$$

For the next result we also require that the optimization part $\Gamma^O_{\Pi,\mathbf{A}}$ is conservative in the sense that, for every unfounded set U of Π wrt. $\mathbf{A}$, it holds that $I_\Gamma(U, \Gamma_{\Pi,\mathbf{A}}, \Pi, \mathbf{A})$ is a solution to $\Gamma^O_{\Pi,\mathbf{A}}$ as well (which is shown for the different optimizations considered subsequently). Then, the solutions to $\Gamma_{\Pi,\mathbf{A}}$ include all assignments induced by unfounded sets of Π wrt. $\mathbf{A}$, but not every solution corresponds to such an induced assignment. Intuitively, this is because it does not necessarily reflect the semantics of external sources.

Proposition 3.6. *Let U be an unfounded set of a program Π wrt. assignment $\mathbf{A}$ such that $\mathbf{A}^\mathbf{T} \cap U \neq \emptyset$. Then $I_\Gamma(U, \Gamma_{\Pi,\mathbf{A}}, \Pi, \mathbf{A})$ is a solution to $\Gamma_{\Pi,\mathbf{A}}$.*

Proof. We prove this by contraposition and show that if $I_\Gamma(U, \Gamma_{\Pi,\mathbf{A}}, \Pi, \mathbf{A})$ is not a solution to $\Gamma_{\Pi,\mathbf{A}}$, then U cannot be an unfounded set.

First observe that the nogoods in $\Gamma^H_{\Pi,\mathbf{A}}$ demand $\mathbf{T}h_r$ to be true for a rule $r \in \Pi$ if and only if some head atom $h \in H(r)$ of this rule is in U. As the conditions in these nogoods are mutually exclusive and therefore consistent, and the truth value of h_r in $I_\Gamma(U, \Gamma_{\Pi,\mathbf{A}}, \Pi, \mathbf{A})$ is defined exactly to this criterion, $\Gamma^C_{r,\mathbf{A}}$ must be involved in a contradiction. Moreover, the nogood $\{\mathbf{F}a \mid \mathbf{T}a \in \mathbf{A}\} \in \Gamma^N_{\Pi,\mathbf{A}}$ eliminates $I_\Gamma(U, \Gamma_{\Pi,\mathbf{A}}, \Pi, \mathbf{A})$ only if U does not intersect with the positive atoms in $\mathbf{A}$. This is no problem because we are only interested in such unfounded sets.

Therefore, if $I_\Gamma(U, \Gamma_{\Pi,\mathbf{A}}, \Pi, \mathbf{A})$ is not a solution to $\Gamma_{\Pi,\mathbf{A}}$, then for some rule $r \in \Pi$ the nogood in $\Gamma^C_{r,\mathbf{A}}$ must be violated. That is, we know the following: $\mathbf{T}h_r \in I_\Gamma(U, \Gamma_{\Pi,\mathbf{A}}, \Pi, \mathbf{A})$ (and therefore $H(r) \cap U \neq \emptyset$), $\mathbf{F}a \in I_\Gamma(U, \Gamma_{\Pi,\mathbf{A}}, \Pi, \mathbf{A})$ for all $a \in B^+_o(r)$, $\mathbf{t}a \in I_\Gamma(U, \Gamma_{\Pi,\mathbf{A}}, \Pi, \mathbf{A})$ for all $a \in B_e(\hat{r})$, and $\mathbf{T}h \in I_\Gamma(U, \Gamma_{\Pi,\mathbf{A}}, \Pi, \mathbf{A})$ for all $h \in H(r)$ with $\mathbf{A} \models h$. Moreover, we have $\Gamma^C_{r,\mathbf{A}} \neq \emptyset$. We now show that this implies that none of the conditions of Definition 38 holds for r wrt. U and $\mathbf{A}$, which contradicts the assumption that U is an unfounded set.

Condition (i) does not hold for r because $\mathbf{A} \models B(r)$ (otherwise $\Gamma^C_{r,\mathbf{A}} = \emptyset$).

Condition (ii) does not hold for r. Suppose to the contrary that it holds. Then there must be some $b \in B(r)$ s.t. $\mathbf{A} \dot\cup \neg.U \not\models b$. Because $\Gamma^C_{r,\mathbf{A}} \neq \emptyset$, we know that $\mathbf{A} \models b$. We make a case distinction on the type of b:

- If b is a positive non-replacement atom, then $\mathbf{F}b \in I_\Gamma(U, \Gamma_{\Pi,\mathbf{A}}, \Pi, \mathbf{A})$ and therefore $b \notin U$. Consequently $\mathbf{A} \dot\cup \neg.U \models b$. Contradiction.
- If b is a negative non-replacement atom, then $\mathbf{A} \models b$ implies $\mathbf{A} \dot\cup \neg.U \models b$. Contradiction.
- If b is a positive or default-negated replacement atom, then $\mathbf{t}b \in I_\Gamma(U, \Gamma_{\Pi,\mathbf{A}}, \Pi, \mathbf{A})$. But this implies, by definition of $I_\Gamma(U, \Gamma_{\Pi,\mathbf{A}}, \Pi, \mathbf{A})$, that $\mathbf{A} \dot\cup \neg.U \models b$. Contradiction.

Condition (iii) does not hold for r because $\mathbf{T}h \in I_\Gamma(U, \Gamma_{\Pi,\mathbf{A}}, \Pi, \mathbf{A})$ and thus, by definition of $I_\Gamma(U, \Gamma_{\Pi,\mathbf{A}}, \Pi, \mathbf{A})$, $h \in U$ for all $h \in H(r)$ with $\mathbf{A} \models h$. Thus $\mathbf{A} \not\models a$ for all $a \in H(r) \setminus U$. □

The next property allows us to find the unfounded sets of Π wrt. $\mathbf{A}$ among all solutions to $\Gamma_{\Pi,\mathbf{A}}$ by using a post-check on the external atoms.

Proposition 3.7. *Let* $\mathbf{S}$ *be a solution to* $\Gamma_{\Pi,\mathbf{A}}$ *such that*

(a) $\mathbf{T}e_{\&g[\mathbf{y}]}(\boldsymbol{x}) \in \mathbf{S}$ *and* $\mathbf{A} \not\models \&g[\boldsymbol{y}](\boldsymbol{x})$ *implies* $\mathbf{A} \dot\cup \neg.U \models \&g[\boldsymbol{y}](\boldsymbol{x})$*; and*

(b) $\mathbf{F}e_{\&g[\mathbf{y}]}(\boldsymbol{x}) \in \mathbf{S}$ *and* $\mathbf{A} \models \&g[\boldsymbol{y}](\boldsymbol{x})$ *implies* $\mathbf{A} \dot\cup \neg.U \not\models \&g[\boldsymbol{y}](\boldsymbol{x})$*,*

where $U = \{a \mid a \in A(\Pi), \mathbf{T}a \in \mathbf{S}\}$*. Then* U *is an unfounded set of* Π *wrt.* $\mathbf{A}$*.*

Proof. Suppose U is not an unfounded set. Then there is an $r \in \Pi$ s.t. $H(r) \cap U \neq \emptyset$ and none of the conditions in Definition 38 is satisfied. We show now that $\mathbf{S}$ cannot be a solution to $\Gamma_{\Pi,\mathbf{A}}$.

Because Condition (i) does not hold, there is a nogood of form

$$N = \{\mathbf{T}h_r\} \cup \{\mathbf{F}a \mid a \in B_o^+(r), \mathbf{A} \models a\} \cup \{\mathbf{t}a \mid a \in B_e(\hat r)\} \cup \{\mathbf{T}h \mid h \in H(r), \mathbf{A} \models h\}$$

in $\Gamma_{\Pi,\mathbf{A}}$.

We now show that $\mathbf{S}$ contains all signed literals of N, i.e., the nogood is violated by $\mathbf{S}$.

Because of $H(r) \cap U \neq \emptyset$, $\mathbf{T}h_r \in \mathbf{S}$ (otherwise a nogood in $\Gamma^H_{r,\mathbf{A}}$ is violated).

As U is not an unfounded set, Condition (ii) in Definition 38 does not hold. Consider all $a \in B_o^+(r)$ s.t. $\mathbf{A} \models a$. Then $a \notin U$, otherwise $A \dot\cup \neg.U \not\models a$ and we have a contradiction with the assumption that Condition (ii) is unsatisfied. But then $\mathbf{F}a \in \mathbf{S}$.

Now consider all $\&g[\mathbf{y}](\mathbf{x}) \in EA(r)$. Then $\mathbf{A} \dot\cup \neg.U \models \&g[\mathbf{y}](\mathbf{x})$ (as Condition (ii) is violated). If $\mathbf{A} \not\models \&g[\mathbf{y}](\mathbf{x})$, then Condition (i) would be satisfied, hence $\mathbf{A} \models \&g[\mathbf{y}](\mathbf{x})$. But then $\mathbf{T}e_{\&g[\mathbf{y}]}(\mathbf{x}) \in \mathbf{S}$, otherwise $\mathbf{A} \dot\cup \neg.U \not\models \&g[\mathbf{y}](\mathbf{x})$ by Condition (b) of this proposition. Next consider all $\mathrm{not}\, \&g[\mathbf{y}](\mathbf{x})$ with $\&g[\mathbf{y}](\mathbf{x}) \in EA(r)$. Then $\mathbf{A} \dot\cup \neg.U \not\models \&g[\mathbf{y}](\mathbf{x})$ (as Condition (ii) is violated). If $\mathbf{A} \models \&g[\mathbf{y}](\mathbf{x})$, then Condition (i) would be satisfied, hence $\mathbf{A} \not\models \&g[\mathbf{y}](\mathbf{x})$. But then $\mathbf{F}e_{\&g[\mathbf{y}]}(\mathbf{x}) \in \mathbf{S}$, otherwise $\mathbf{A} \dot\cup \neg.U \models \&g[\mathbf{y}](\mathbf{x})$ by Condition (a) of this proposition. Therefore, we have $\mathbf{t}a \in \mathbf{S}$ for all $a \in B_e(\hat r)$.

Finally, because Condition (iii) in Definition 38 does not hold, $h \in U$ and therefore also $\mathbf{T}h \in \mathbf{S}$ for all $h \in H(r)$ with $\mathbf{A} \models a$.

This concludes the proof that $\mathbf{S}$ cannot be a solution to $\Gamma_{\Pi,\mathbf{A}}$ satisfying (a) and (b), if U is not an unfounded set. □

Informally, the proposition states that true non-replacement atoms in $\mathbf{S}$ which also appear in Π form an unfounded set, provided that truth of the external replacement atoms $e_{\&g[\mathbf{y}]}(\mathbf{x})$ in $\mathbf{S}$ coincides with the truth of the corresponding $\&g[\mathbf{y}](\mathbf{x})$ wrt. $\mathbf{A} \dot\cup \neg.U$ (as in Definition 40). However, this check is just required if the truth value of $e_{\&g[\mathbf{y}]}(\mathbf{x})$ in $\mathbf{S}$ and of $\&g[\mathbf{y}](\mathbf{x})$ in $\mathbf{A}$ differ. This gives rise to an important optimization for the implementation: external atoms, whose (known) truth value of $\&g[\mathbf{y}](\mathbf{x})$ wrt. $\mathbf{A}$ matches the truth value of $e_{\&g[\mathbf{y}]}(\mathbf{x})$ in $\mathbf{S}$, do not need to be post-checked.

We must further show that for an unfounded set U of a program Π wrt. an interpretation, the induced assignment fulfills the conditions of Proposition 3.7, i.e., no unfounded sets are lost during the post-check.

Proposition 3.8. *Let U be an unfounded set of a program Π wrt. assignment $\mathbf{A}$ such that $\mathbf{A}^\mathbf{T} \cap U \neq \emptyset$. Then $I_\Gamma(U, \Gamma_{\Pi,\mathbf{A}}, \Pi, \mathbf{A})$ fulfills Conditions (a) and (b) of Proposition 3.7.*

Proof. Let $\mathbf{S} = I_\Gamma(U, \Gamma_{\Pi,\mathbf{A}}, \Pi, \mathbf{A})$. If for an external atom $\&g[\mathbf{y}](\mathbf{x})$ in Π we have $\mathbf{T}e_{\&g[\mathbf{y}]}(\mathbf{x}) \in \mathbf{S}$, then by definition of $I_\Gamma(U, \Gamma_{\Pi,\mathbf{A}}, \Pi, \mathbf{A})$ we have $\mathbf{A} \dot\cup \neg.U \models \&g[\mathbf{y}](\mathbf{x})$ (satisfying (a)). If for an external atom $\&g[\mathbf{y}](\mathbf{x})$ in Π we have $\mathbf{F}e_{\&g[\mathbf{y}]}(\mathbf{x}) \in \mathbf{S}$, then by definition of $I_\Gamma(U, \Gamma_{\Pi,\mathbf{A}}, \Pi, \mathbf{A})$ we have $\mathbf{A} \dot\cup \neg.U \not\models \&g[\mathbf{y}](\mathbf{x})$ (satisfying (b)). □

Corollary 3.1. *If $\Gamma_{\Pi,\mathbf{A}}$ has no solution which fulfills the conditions of Proposition 3.7, then $U \cap \mathbf{A}^\mathbf{T} = \emptyset$ for every unfounded set U of Π.*

Proof. If there would be a UFS U of Π wrt. $\mathbf{A}$ which intersects with $\mathbf{A}^\mathbf{T}$, then by Proposition 3.6 $I_\Gamma(U, \Gamma_{\Pi,\mathbf{A}}, \Pi, \mathbf{A})$ would be a solution to $\Gamma_{\Pi,\mathbf{A}}$, and by Proposition 3.8 it would fulfill the conditions of Proposition 3.7. □

Example 28 (ctd.)**.** Reconsider program $\Pi = \{r_1\colon\ p \leftarrow \&id[p]()\}$ from Examples 26 and 27 and the compatible set $\hat{\mathbf{A}} = \{\mathbf{T}p, \mathbf{T}e_{\&id[p]}\}$. The unfounded set search is encoded by the nogood set $\Gamma_{\Pi,\mathbf{A}} = \big\{\{\mathbf{T}h_{r_1}, \mathbf{F}p\}, \{\mathbf{F}h_{r_1}, \mathbf{T}p\}, \{\mathbf{T}h_{r_1}, \mathbf{T}e_{\&id[p]}(), \mathbf{T}p\}\big\}$ and has some solutions $\mathbf{S} \supseteq \{\mathbf{T}h_{r_1}, \mathbf{T}p, \mathbf{F}e_{\&id[p]}()\}$, which correspond to the unfounded set $U = \{p\}$. Here, $\mathbf{F}e_{\&id[p]}()$ represents that $\mathbf{A} \dot\cup \neg.U \not\models \&id[p]()$. □

Note that due to the premises in Conditions (a) and (b) of Proposition 3.7, the post-check is faster if it holds for many external atoms $\&g[\mathbf{y}](\mathbf{x})$ in Π that $\mathbf{A} \models \&g[\mathbf{y}](\mathbf{x})$ implies $\mathbf{T}e_{\&g[\mathbf{y}]}(\mathbf{x}) \in \mathbf{S}$. This can be exploited during the construction of $\mathbf{S}$ as follows: if it is not absolutely necessary to set the truth value of $e_{\&g[\mathbf{y}]}(\mathbf{x})$ differently, then carry over the value from $\&g[\mathbf{y}](\mathbf{x})$ wrt. $\mathbf{A}$. Specifically, this is successful if $e_{\&g[\mathbf{y}]}(\mathbf{x})$ does not occur in $\Gamma_{\Pi,\mathbf{A}}$.

3.2.2 Uniform Encoding of the Unfounded Set Search

The encoding $\Gamma_{\Pi,\mathbf{A}}$ presented above has the disadvantage that it depends on the current assignment $\mathbf{A}$. Therefore it needs to be generated separately for every unfounded set check if the assignment changed (which is very likely). As this causes significant overhead, we present now an advanced encoding which is reusable for any assignment. For this we introduce some additional variables which represent the truth values of the atoms in the current assignment. Before

an unfounded set check, the current assignment is injected by setting the values of these variables to fixed values, which can be done using *assumptions* as supported by modern SAT solvers such as CLASP [CLASP Website, 2014]. Changing assumptions is much easier than changing the encoding, which leads to an additional speedup in some cases, especially for programs which need many unfounded set checks. Moreover, this has the advantage that the solver instance for the unfounded set check can keep running over the whole lifetime of the HEX solver; in contrast to the creation of a separate instance for each unfounded set check, this preserves also learned nogoods.

Our advanced encoding uses a set Ω_Π of nogoods. As before, the idea is that the set of non-replacement atoms of a solution to Ω_Π represents a (potential) unfounded set U of Π wrt. some assignment $\mathbf{A}$, while the external replacement atoms encode the truth values of the corresponding external atoms wrt. $\mathbf{A} \dot\cup \neg.U$. The basic idea of the encoding is similar to the encoding $\Gamma_{\Pi,\mathbf{A}}$. However, unlike $\Gamma_{\Pi,\mathbf{A}}$, the structure of the nogoods in Ω_Π do *not* depend on the current interpretation $\mathbf{A}$, but $\mathbf{A}$ is merely a parameter, which is injected by setting dedicated atoms in the encoding to fixed values. This allows for reusing the same problem encoding for all unfounded set checks also wrt. different assignments. However, as the encoding Ω_Π is conceptually more complex than $\Gamma_{\Pi,\mathbf{A}}$, the initialization is computationally (slightly) more costly, hence the advantages of our new encoding become visible for instances with many compatible sets and thus many unfounded set checks, while it might be counterproductive for very small instances. The development of a heuristics for dynamically choosing between the UFS search encodings is up to future work.

The nogood set Ω_Π is built over atoms

$$\begin{aligned} A(\Omega_\Pi) \;=\; & A(\hat\Pi) \cup \{h_r, l_r \mid r \in \Pi\} \;\cup \\ & \{a_\mathbf{A} \mid a \in A(\hat\Pi)\} \;\cup \\ & \{a_{\mathbf{A}\dot\cup\neg.U}, a_{\mathbf{A}\wedge U}, a_{\overline{\mathbf{A}}\vee U} \mid a \in A(\Pi)\}, \end{aligned}$$

where we have the following fresh atoms which do not occur in $\hat\Pi$:

- h_r and l_r for every rule r in Π
- $a_\mathbf{A}$ and for every ordinary atom $a \in A(\hat\Pi)$ (i.e. ordinary atoms in Π and external replacement atoms)
- $a_{\mathbf{A}\dot\cup\neg.U}, a_{\mathbf{A}\wedge U}, a_{\overline{\mathbf{A}}\vee U}$ for every ordinary atom $a \in A(\Pi)$

The *auxiliary atoms* $a_\mathbf{A}$, $a_{\mathbf{A}\dot\cup\neg.U}$, $a_{\mathbf{A}\wedge U}$, $a_{\overline{\mathbf{A}}\vee U}$ are used to make the encoding reuable for any assignment $\mathbf{A}$. Only during the unfounded set check with respect to a certain assignment, we will temporarily add assumptions to the solver which force certain truth values of the atoms $a_\mathbf{A}$ for all $a \in A(\hat\Pi)$ depending on the current assignment $\mathbf{A}$.

To this end, a *set of assumptions* $\mathcal{A}$ is a consistent set of signed literals. An interpretation $\mathbf{A}$ is a *solution of a set of nogoods* Δ *wrt. a set of assumptions* $\mathcal{A}$ if $\delta \not\subseteq \mathbf{A}$ for all $\delta \in \Delta$ and $\mathcal{A} \subseteq \mathbf{A}$. That is, assumptions fix the truth value of some atoms. Modern ASP and SAT solvers support assumptions natively, which can be easily undone without a complete reset of the reasoner and

recreating the whole problem instance. This is an essential feature for efficiently implementing our improved encoding.

Intuitively, $a_{\mathbf{A}}$ represents the truth value of a in $\mathbf{A}$, $a_{\mathbf{A}\dot\cup\neg.U}$ represents the truth value of a in $\mathbf{A} \dot\cup \neg.U$ (where U is the currently constructed unfounded set), $a_{\mathbf{A}\wedge U}$ represents that a is true in $\mathbf{A}$ and is contained in U, and $a_{\overline{\mathbf{A}}\vee U}$ represents that a is false in $\mathbf{A}$ or it is contained in U. Our encoding Ω_Π is then as follows:

- $\Omega_\Pi = \Omega_\Pi^N \cup \Omega_\Pi^O$ with
- $\Omega_\Pi^N = \{\mathbf{F}a \mid a \in A(\Pi)\} \cup \bigcup_{a\in A(\Pi)} \Omega_a^D \cup \bigcup_{r\in\Pi} \left(\Omega_r^H \cup \Omega_r^C\right)$
  as the necessary part, where
- $\{\mathbf{F}a \mid a \in A(\Pi)\}$
  encodes that we search for a nonempty unfounded set;
- $\Omega_a^D = \begin{cases} \big\{\{\mathbf{F}a_{\mathbf{A}\wedge U}, \mathbf{T}a_{\mathbf{A}}, \mathbf{T}a\}, \{\mathbf{T}a_{\mathbf{A}\wedge U}, \mathbf{F}a_{\mathbf{A}}\}, \{\mathbf{T}a_{\mathbf{A}\wedge U}, \mathbf{F}a\}\big\} \cup \\ \big\{\{\mathbf{F}a_{\overline{\mathbf{A}}\vee U}, \mathbf{F}a_{\mathbf{A}}\}, \{\mathbf{F}a_{\overline{\mathbf{A}}\vee U}, \mathbf{T}a\}, \{\mathbf{T}a_{\overline{\mathbf{A}}\vee U}, \mathbf{T}a_{\mathbf{A}}, \mathbf{F}a\}\big\} \cup \\ \big\{\{\mathbf{T}a_{\mathbf{A}\dot\cup\neg.U}, \mathbf{F}a_{\mathbf{A}}\}, \{\mathbf{T}a_{\mathbf{A}\dot\cup\neg.U}, \mathbf{T}a\}, \{\mathbf{F}a_{\mathbf{A}\dot\cup\neg.U}, \mathbf{T}a_{\mathbf{A}}, \mathbf{F}a\}\big\} \end{cases}$
  encodes that $a_{\mathbf{A}\wedge U}$ is true iff $a_{\mathbf{A}}$ and a are both true, $a_{\overline{\mathbf{A}}\vee U}$ is true iff $a_{\mathbf{A}}$ is false or a is true, and $a_{\mathbf{A}\dot\cup\neg.U}$ is true iff $a_{\mathbf{A}}$ is true and a is false;
- $\Omega_r^H = \big\{\{\mathbf{T}h_r\} \cup \{\mathbf{F}h \mid h \in H(r)\}\big\} \cup \big\{\{\mathbf{F}h_r, \mathbf{T}h\} \mid h \in H(r)\big\}$
  encodes that h_r is true for a rule r iff some atom of $H(r)$ is in the unfounded set;
- $\Omega_r^C = \begin{cases} \big\{\{\mathbf{T}h_r\} \cup & \\ \quad \{\mathbf{T}a_{\mathbf{A}} \mid a \in B^+(\hat r)\} \cup \{\mathbf{F}a_{\mathbf{A}} \mid a \in B^-(\hat r)\} \cup & (i) \\ \quad \{\mathbf{F}a_{\mathbf{A}\wedge U} \mid a \in B_o^+(r)\} \cup \{\mathbf{t}a \mid a \in B_e(\hat r)\} \cup & (ii) \\ \quad \{\mathbf{T}h_{\overline{\mathbf{A}}\vee U} \mid h \in H(r)\}\big\} & (iii) \end{cases}$
  encodes that Condition (i), (ii) or (iii) of Definition 38 must hold if h_r is true.

  More specifically, for an unfounded set U and a rule r with $H(r) \cap U \neq \emptyset$ (h_r is true) it must not happen that $\mathbf{A} \models B(r)$ (Condition (i) fails), no $a \in B_o^+(r)$ with $\mathbf{A} \models a$ is in the unfounded set and all $a \in B_e(\hat r)$ are true wrt. $\mathbf{A} \dot\cup \neg.U$ (Condition (ii) fails), and all $h \in H(r)$ with $\mathbf{A} \models h$ are in the unfounded set (Condition (iii) fails).

Example 29. Reconsider program $\Pi = \{r_1\colon\ p \leftarrow \&id[p]()\}$ from Example 26. The constructed nogood set is

$$\begin{aligned}\Omega_\Pi = \big\{&\{\mathbf{F}p\}, \{\mathbf{F}p_{\mathbf{A}\wedge U}, \mathbf{T}p_{\mathbf{A}}, \mathbf{T}p\}, \{\mathbf{T}p_{\mathbf{A}\wedge U}, \mathbf{F}p_{\mathbf{A}}\}, \{\mathbf{T}p_{\mathbf{A}\wedge U}, \mathbf{F}p\},\\ &\{\mathbf{F}p_{\bar{\mathbf{A}}\vee U}, \mathbf{F}p_{\mathbf{A}}\}, \{\mathbf{F}p_{\bar{\mathbf{A}}\vee U}, \mathbf{T}p\}, \{\mathbf{T}p_{\bar{\mathbf{A}}\vee U}, \mathbf{T}p_{\mathbf{A}}, \mathbf{F}p\},\\ &\{\mathbf{T}p_{\mathbf{A}\dot\cup\neg.U}, \mathbf{F}p_{\mathbf{A}}\}, \{\mathbf{T}p_{\mathbf{A}\dot\cup\neg.U}, \mathbf{T}p\}, \{\mathbf{F}p_{\mathbf{A}\dot\cup\neg.U}, \mathbf{T}p_{\mathbf{A}}, \mathbf{F}p\},\\ &\{\mathbf{T}h_{r_1}, \mathbf{F}p\}, \{\mathbf{F}h_{r_1}, \mathbf{T}p\}, \{\mathbf{T}h_{r_1}, \mathbf{T}e_{\&id[p]}()_{\mathbf{A}}, \mathbf{T}e_{\&id[p]}(), \mathbf{T}p_{\bar{\mathbf{A}}\vee U}\}\big\}.\end{aligned}$$

□

Towards computing unfounded sets, observe that they can be extended to solutions to the set of nogoods Ω_Π over $A(\Omega_\Pi)$. Conversely, the solutions to Ω_Π include specific extensions of all unfounded sets, which are again characterized by induced assignments: that is, by assigning true to all atoms in U, to all h_r such that $H(r)$ intersects with U, and to all external replacement atoms $e_{\&g[\mathbf{y}]}(\mathbf{x})$ such that $\&g[\mathbf{y}](\mathbf{x})$ is true wrt. $\mathbf{A} \dot\cup \neg.U$, appropriate truth values to the auxiliary atoms according to their intuitive meaning described above, and assigning false to all other atoms in $A(\Omega_\Pi)$. More formally, we define:

Definition 41 (Induced Assignment of an Unfounded Set wrt. Ω_Π)**.** Let U be an unfounded set of program Π wrt. $\mathbf{A}$. The *assignment induced by U wrt.* Ω_Π, denoted $I_\Omega(U, \Omega_\Pi, \Pi, \mathbf{A})$, is

$$
\begin{aligned}
I_\Omega(U, \Omega_\Pi, \Pi, \mathbf{A}) &= I^0_\Omega(U, \Pi, \mathbf{A}) \cup \left\{\mathbf{F}a \mid a \in A(\Omega_\Pi), \mathbf{T}a \notin I^0_\Omega(U, \Pi, \mathbf{A})\right\}, \text{ where}\\
I^0_\Omega(U, \Pi, \mathbf{A}) = \; & \left\{\mathbf{T}a \mid a \in U\right\} \cup \left\{\mathbf{T}h_r \mid r \in \Pi, H(r) \cap U \neq \emptyset\right\} \cup\\
& \left\{\mathbf{T}e_{\&g[\mathbf{y}]}(\mathbf{x}) \mid \&g[\mathbf{y}](\mathbf{x}) \in EA(\Pi), \mathbf{A} \dot\cup \neg.U \models \&g[\mathbf{y}](\mathbf{x})\right\} \cup\\
& \left\{\mathbf{T}a_\mathbf{A} \mid a \in A(\Pi), \mathbf{T}a \in \mathbf{A}\right\} \cup\\
& \left\{\mathbf{T}e_{\&g[\mathbf{y}]}(\mathbf{x})_\mathbf{A} \mid \&g[\mathbf{y}](\mathbf{x}) \in EA(\Pi), \mathbf{A} \models \&g[\mathbf{y}](\mathbf{x})\right\} \cup\\
& \left\{\mathbf{T}a_{\mathbf{A}\wedge U} \mid a \in A(\Pi), \mathbf{T}a \in \mathbf{A}, a \in U\right\} \cup\\
& \left\{\mathbf{T}a_{\mathbf{A}\dot\cup\neg.U} \mid a \in A(\Pi), \mathbf{T}a \in \mathbf{A}, a \notin U\right\} \cup\\
& \left\{\mathbf{T}a_{\overline{\mathbf{A}}\vee U} \mid a \in A(\Pi), \mathbf{F}a \in \mathbf{A} \text{ or } a \in U\right\}.
\end{aligned}
$$

Then, the solutions to Ω_Π with assumptions

$$
\begin{aligned}
\mathcal{A}_\mathbf{A} \;=\; & \left\{\mathbf{T}a_\mathbf{A} \mid a \in A(\Pi), \mathbf{T}a \in \mathbf{A}\right\} \cup \left\{\mathbf{F}a_\mathbf{A} \mid a \in A(\Pi), \mathbf{F}a \in \mathbf{A}\right\} \cup\\
& \left\{\mathbf{T}\hat a_\mathbf{A} \mid a \in EA(\Pi), \mathbf{A} \models a\right\} \cup \left\{\mathbf{F}\hat a_\mathbf{A} \mid a \in EA(\Pi), \mathbf{A} \not\models a\right\}
\end{aligned}
$$

include all assignments induced by unfounded sets of Π wrt. $\mathbf{A}$. But as above, not every solution corresponds to such an induced assignment.

As before, we assume that the nogoods in Ω^O_Π are conservative in the sense that for every unfounded set U of Π wrt. $\mathbf{A}$, it holds that $I_\Omega(U, \Gamma_{\Pi,\mathbf{A}}, \Pi, \mathbf{A})$ is a solution to Ω^O_Π as well. We will present concrete optimization nogoods which fulfill this condition in the next section.

Proposition 3.9. *Let U be an unfounded set of a program Π wrt. assignment $\mathbf{A}$ such that $\mathbf{A}^\mathbf{T} \cap U \neq \emptyset$. Then $I_\Omega(U, \Omega_\Pi, \Pi, \mathbf{A})$ is a solution to Ω_Π with assumptions $\mathcal{A}_\mathbf{A}$.*

Proof. We prove this by contraposition and show that if $I_\Omega(U, \Omega_\Pi, \Pi, \mathbf{A})$ is not a solution to Ω_Π with assumptions $\mathcal{A}_\mathbf{A}$, then U cannot be an unfounded set.

First observe that the nogoods in Ω^H_r demand $\mathbf{T}h_r$ to be true for a rule $r \in \Pi$ if and only if some head atom $h \in H(r)$ of this rule is in U. Moreover, the nogoods in Ω^D_a for each $a \in A(\Pi)$ force $a_{\mathbf{A}\dot\cup\neg.U}$ to true if and only if $\mathbf{T}a \in \mathbf{A} \dot\cup \neg.U$, $a_{\mathbf{A}\wedge U}$ to true if and only if $\mathbf{T}a \in \mathbf{A}$ and $a \in U$, and $a_{\overline{\mathbf{A}}\vee U}$ to true if and only if $\mathbf{F}a \in \mathbf{A}$ or $a \in U$. As the truth values of h_r for each $r \in \Pi$, and $a_{\mathbf{A}\dot\cup\neg.U}$, $a_{\mathbf{A}\wedge U}$ and $a_{\overline{\mathbf{A}}\vee U}$ for each $a \in A(\Pi)$ in $I_\Omega(U, \Omega_\Pi, \Pi, \mathbf{A})$ are defined exactly to these criteria, a contradiction must involve Ω^C_r for some $r \in \Pi$. Moreover, the nogood $\{\mathbf{F}a \mid a \in A(\Pi)\}$ eliminates $I_\Omega(U, \Omega_\Pi, \Pi, \mathbf{A})$ only if U does not intersect with the positive atoms in $\mathbf{A}$. This is no problem because we are only interested in such unfounded sets.

Therefore, if $I_\Omega(U, \Omega_\Pi, \Pi, \mathbf{A})$ is not a solution to $\Omega_\Pi, \mathbf{A}$, then for some rule $r \in \Pi$ the nogood in Ω^C_r must be violated. That is, we know the following:

(I) $\mathbf{T}h_r \in I_\Omega(U, \Omega_\Pi, \Pi, \mathbf{A})$ (and therefore $H(r) \cap U \neq \emptyset$);

(II) $\mathbf{T}a_\mathbf{A} \in I_\Omega(U, \Omega_\Pi, \Pi, \mathbf{A})$ for all $a \in B^+(\hat{r})$ and $\mathbf{F}a_\mathbf{A} \in I_\Omega(U, \Omega_\Pi, \Pi, \mathbf{A})$ for all $a \in B^-(\hat{r})$;

(III) $\mathbf{F}a_{\mathbf{A}\wedge U} \in I_\Omega(U, \Omega_\Pi, \Pi, \mathbf{A})$ for all $a \in B_o^+(r)$ and $\mathbf{t}a \in I_\Omega(U, \Omega_\Pi, \Pi, \mathbf{A})$ for all $a \in B_e(\hat{r})$; and

(IV) $\mathbf{T}h_{\overline{\mathbf{A}}\vee U} \in I_\Omega(U, \Omega_\Pi, \Pi, \mathbf{A})$ for all $h \in H(r)$.

We now show that this implies that none of the conditions of Definition 38 holds for r wrt. U and $\mathbf{A}$, which contradicts the assumption that U is an unfounded set (h_r is true in $I_\Omega(U, \Omega_\Pi, \Pi, \mathbf{A})$, which implies $H(r) \cap U \neq \emptyset$).

Condition (i) does not hold for r because of (II), which implies, by definition of our assumptions $\mathcal{A}_\mathbf{A}$, $\mathbf{A} \models B(r)$.

Condition (ii) does not hold for r. Suppose to the contrary that it holds. Then there must be some $b \in B(r)$ s.t. $\mathbf{A} \dot\cup \neg.U \not\models b$. Since Condition (i) is already known to be violated, we can assume that $\mathbf{A} \models b$. We make a case distinction on the type of b:

- If b is a positive non-replacement atom, then $b \in U$ (otherwise $\mathbf{A} \dot\cup \neg.U \models b$). But then we have by definition of $I_\Omega(U, \Omega_\Pi, \Pi, \mathbf{A})$ that $\mathbf{T}b_{\mathbf{A}\wedge U} \in I_\Omega(U, \Omega_\Pi, \Pi, \mathbf{A})$, which contradicts (III).
- If b is a negative non-replacement atom, then $\mathbf{A} \models b$ implies $\mathbf{A} \dot\cup \neg.U \models b$. Contradiction.
- If b is a positive or default-negated replacement atom, then $\mathbf{t}b \in I_\Omega(U, \Omega_\Pi, \Pi, \mathbf{A})$ because of (III). But this implies, by definition of $I_\Omega(U, \Omega_\Pi, \Pi, \mathbf{A})$, that $\mathbf{A} \dot\cup \neg.U \models b$. Contradiction.

Condition (iii) does not hold for r because $\mathbf{T}h_{\overline{\mathbf{A}}\vee U} \in I_\Omega(U, \Omega_\Pi, \Pi, \mathbf{A})$ and thus, by definition of $I_\Omega(U, \Omega_\Pi, \Pi, \mathbf{A})$, $h \in U$ for all $h \in H(r)$ with $\mathbf{A} \models h$. Thus $\mathbf{A} \not\models a$ for all $a \in H(r) \setminus U$. □

The next property allows us to find the unfounded sets of Π wrt. $\mathbf{A}$ among all solutions to Ω_Π wrt. assumptions $\mathcal{A}_\mathbf{A}$ by using a post-check on the external atoms.

Proposition 3.10. *Let* $\mathbf{S}$ *be a solution to* Ω_Π *such that the assumptions* $\mathcal{A}_\mathbf{A}$ *are satisfied and*

(a) $\mathbf{T}e_{\&g[\mathbf{y}]}(\boldsymbol{x}) \in \mathbf{S}$ *and* $\mathbf{A} \not\models \&g[\mathbf{y}](\boldsymbol{x})$ *implies* $\mathbf{A} \dot\cup \neg.U \models \&g[\mathbf{y}](\boldsymbol{x})$*; and*

(b) $\mathbf{F}e_{\&g[\mathbf{y}]}(\boldsymbol{x}) \in \mathbf{S}$ *and* $\mathbf{A} \models \&g[\mathbf{y}](\boldsymbol{x})$ *implies* $\mathbf{A} \dot\cup \neg.U \not\models \&g[\mathbf{y}](\boldsymbol{x})$

where $U = \{a \mid a \in A(\Pi), \mathbf{T}a \in \mathbf{S}\}$*. Then* U *is an unfounded set of* Π *wrt.* $\mathbf{A}$*.*

As for $\Gamma_{\Pi,\mathbf{A}}$, the proposition states that true non-replacement atoms in $\mathbf{S}$ which also appear in Π form an unfounded set, provided that truth of the external replacement atoms $e_{\&g[\mathbf{y}]}(\mathbf{x})$ in $\mathbf{S}$ coincides with the truth of the corresponding $\&g[\mathbf{y}](\mathbf{x})$ wrt. $\mathbf{A} \dot\cup \neg.U$ (as in Definition 40). Again, this check is only required if the truth value of $e_{\&g[\mathbf{y}]}(\mathbf{x})$ in $\mathbf{S}$ and of $\&g[\mathbf{y}](\mathbf{x})$ wrt. $\mathbf{A}$ differ.

Proof. Suppose U is not an unfounded set. Then there is a $r \in \Pi$ s.t. $H(r) \cap U \neq \emptyset$ and none of the conditions in Definition 38 is satisfied. We show now that $\mathbf{S}$ cannot be a solution to Ω_Π s.t. the assumptions $\mathcal{A}_\mathbf{A}$ are satisfied.

For rule r, Ω_Π contains a nogood of form

$$\begin{aligned} N = &\{\mathbf{T}h_r\} \cup \\ &\{\mathbf{T}a_\mathbf{A} \mid a \in B^+(\hat{r})\} \cup \{\mathbf{F}a_\mathbf{A} \mid a \in B^-(\hat{r})\} \cup \\ &\{\mathbf{F}a_{\mathbf{A}\wedge U} \mid a \in B_o^+(r)\} \cup \{\mathbf{t}a \mid a \in B_e(\hat{r})\} \cup \\ &\{\mathbf{T}h_{\overline{\mathbf{A}}\vee U} \mid h \in H(r)\}. \end{aligned}$$

We now show that $\mathbf{S}$ contains all signed literals of N, i.e., the nogood is violated by $\mathbf{S}$.

Because of $H(r) \cap U \neq \emptyset$, $\mathbf{T}h_r \in \mathbf{S}$ (otherwise a nogood in Ω_r^H is violated).

Because U is not an unfounded set, Condition (i) in Definition 38 does not hold. Therefore $\mathbf{A} \models B(r)$. But then our assumptions $\mathcal{A}_\mathbf{A}$ force $\mathbf{T}b_\mathbf{A} \in \mathbf{S}$ for all $b \in B^+(\hat{r})$ and $\mathbf{F}b_\mathbf{A} \in \mathbf{S}$ for all $b \in B^-(\hat{r})$.

As U is not an unfounded set, Condition (ii) in Definition 38 does not hold. Consider all $a \in B_o^+(r)$. Then $\mathbf{A} \models a$ and $a \notin U$. But $a \notin U$ implies $\mathbf{F}a \in \mathbf{S}$. Then nogood $\{\mathbf{T}a_{\mathbf{A}\wedge U}, \mathbf{F}a\}$ implies $\mathbf{F}a_{\mathbf{A}\wedge U}$.

Now consider all $\&g[\mathbf{y}](\mathbf{x}) \in EA(r)$. Then $\mathbf{A} \dot\cup \neg.U \models \&g[\mathbf{y}](\mathbf{x})$ (as Condition (ii) is violated). If $\mathbf{A} \not\models \&g[\mathbf{y}](\mathbf{x})$, then Condition (i) would be satisfied, hence $\mathbf{A} \models \&g[\mathbf{y}](\mathbf{x})$. But then $\mathbf{T}e_{\&g[\mathbf{y}]}(\mathbf{x}) \in \mathbf{S}$, otherwise $\mathbf{A} \dot\cup \neg.U \not\models \&g[\mathbf{y}](\mathbf{x})$ by Condition (b) of this proposition. Next consider all $\mathrm{not}\, \&g[\mathbf{y}](\mathbf{x})$ with $\&g[\mathbf{y}](\mathbf{x}) \in EA(r)$. Then $\mathbf{A} \dot\cup \neg.U \not\models \&g[\mathbf{y}](\mathbf{x})$ (as Condition (ii) is violated). If $\mathbf{A} \models \&g[\mathbf{y}](\mathbf{x})$, then Condition (i) would be satisfied, hence $\mathbf{A} \not\models \&g[\mathbf{y}](\mathbf{x})$. But then $\mathbf{F}e_{\&g[\mathbf{y}]}(\mathbf{x}) \in \mathbf{S}$, otherwise $\mathbf{A} \dot\cup \neg.U \models \&g[\mathbf{y}](\mathbf{x})$ by Condition (a) of this proposition. Therefore, we have $\mathbf{t}a \in \mathbf{S}$ for all $a \in B_e(\hat{r})$.

Finally, because Condition (iii) in Definition 38 does not hold, $h \in U$ and therefore also $\mathbf{T}h \in \mathbf{S}$ for all $h \in H(r)$ with $\mathbf{A} \models a$. That is, for each $h \in H(r)$, either $\mathbf{F}h_\mathbf{A} \in \mathbf{S}$ or $\mathbf{T}h \in \mathbf{S}$. But by the nogoods $\{\mathbf{F}a_{\overline{\mathbf{A}}\vee U}, \mathbf{F}a_\mathbf{A}\}, \{\mathbf{F}a_{\overline{\mathbf{A}}\vee U}, \mathbf{T}a\} \in \Omega_a^D$ both cases imply $\mathbf{T}a_{\overline{\mathbf{A}}\vee U} \in \mathbf{S}$.

This concludes the proof that $\mathbf{S}$ cannot be a solution to Ω_Π satisfying assumptions $\mathcal{A}_\mathbf{A}$ and Conditions (a) and (b), if U is not an unfounded set. □

Again, we must further show that for an unfounded set U of a program Π wrt. an interpretation, the induced assignment fulfills the conditions of Proposition 3.10.

Proposition 3.11. *Let U be an unfounded set of a program Π wrt. assignment $\mathbf{A}$ such that $\mathbf{A}^\mathbf{T} \cap U \neq \emptyset$. Then $I_\Gamma(U, \Gamma_{\Pi,\mathbf{A}}, \Pi, \mathbf{A})$ fulfills Conditions (a) and (b) of Proposition 3.10.*

Proof. Let $\mathbf{S} = I_\Omega(U, \Omega_\Pi, \Pi, \mathbf{A})$. If for an external atom $\&g[\mathbf{y}](\mathbf{x})$ in Π we have $\mathbf{T}e_{\&g[\mathbf{y}]}(\mathbf{x}) \in \mathbf{S}$, then by definition of $I_\Omega(U, \Omega_\Pi, \Pi, \mathbf{A})$ we have $\mathbf{A} \dot\cup \neg.U \models \&g[\mathbf{y}](\mathbf{x})$ (satisfying (a)). If for an external atom $\&g[\mathbf{y}](\mathbf{x})$ in Π we have $\mathbf{F}e_{\&g[\mathbf{y}]}(\mathbf{x}) \in \mathbf{S}$, then by definition of $I_\Omega(U, \Omega_\Pi, \Pi, \mathbf{A})$ we have $\mathbf{A} \dot\cup \neg.U \not\models \&g[\mathbf{y}](\mathbf{x})$ (satisfying (b)). □

Corollary 3.2. *If Ω_Π has no solution which satisfies the assumptions $\mathcal{A}_\mathbf{A}$ and which fulfills the conditions of Proposition 3.10, then $U \cap \mathbf{A}^\mathbf{T} = \emptyset$ for every unfounded set U of Π.*

Proof. If there would be a UFS U of Π wrt. $\mathbf{A}$ which intersects with $\mathbf{A}^{\mathbf{T}}$, then by Proposition 3.9 $I_\Omega(U, \Omega_\Pi, \Pi, \mathbf{A})$ would be a solution to Ω_Π with assumptions $\mathcal{A}_\mathbf{A}$, and by Proposition 3.11 it would fulfill the conditions of Proposition 3.10. □

Example 30 (ctd.)**.** Reconsider program $\Pi = \{r_1\colon\ p \leftarrow \mathit{\&id}[p]()\}$ from Example 28 and the compatible set $\mathbf{A}_2 = \{\mathbf{T}p, \mathbf{T}e_{\mathit{\&id}[p]}\}$.

The nogood set

$$\begin{aligned}\Omega_\Pi = \big\{&\{\mathbf{F}p\}, \{\mathbf{F}p_{\mathbf{A}\wedge U}, \mathbf{T}p_\mathbf{A}, \mathbf{T}p\}, \{\mathbf{T}p_{\mathbf{A}\wedge U}, \mathbf{F}p_\mathbf{A}\}, \{\mathbf{T}p_{\mathbf{A}\wedge U}, \mathbf{F}p\},\\ &\{\mathbf{F}p_{\bar{\mathbf{A}}\vee U}, \mathbf{F}p_\mathbf{A}\}, \{\mathbf{F}p_{\bar{\mathbf{A}}\vee U}, \mathbf{T}p\}, \{\mathbf{T}p_{\bar{\mathbf{A}}\vee U}, \mathbf{T}p_\mathbf{A}, \mathbf{F}p\},\\ &\{\mathbf{T}p_{\mathbf{A}\dot\cup\neg.U}, \mathbf{F}p_\mathbf{A}\}, \{\mathbf{T}p_{\mathbf{A}\dot\cup\neg.U}, \mathbf{T}p\}, \{\mathbf{F}p_{\mathbf{A}\dot\cup\neg.U}, \mathbf{T}p_\mathbf{A}, \mathbf{F}p\},\\ &\{\mathbf{T}h_{r_1}, \mathbf{F}p\}, \{\mathbf{F}h_{r_1}, \mathbf{T}p\}, \{\mathbf{T}h_{r_1}, \mathbf{T}e_{\mathit{\&id}[p]}()_\mathbf{A}, \mathbf{T}e_{\mathit{\&id}[p]}(), \mathbf{T}p_{\bar{\mathbf{A}}\vee U}\}\big\}\end{aligned}$$

with assumptions $\mathcal{A}_{\mathbf{A}_2} = \{\mathbf{T}p_\mathbf{A}\}$ has some solutions $S \supseteq \{\mathbf{T}h_{r_1}, \mathbf{T}p, \mathbf{T}p_\mathbf{A}, \mathbf{F}e_{\mathit{\&id}[p]}, \mathbf{T}p_{\mathbf{A}\wedge U}, \mathbf{T}p_{\bar{\mathbf{A}}\vee U}, \mathbf{F}p_{\mathbf{A}\dot\cup\neg.U}\}$, which correspond to the unfounded set $U = \{p\}$. Here, $\mathbf{F}e_{\mathit{\&id}[p]}()$ represents that $\mathbf{A}_2\dot\cup\, \neg.U \not\models \mathit{\&id}[p]()$. □

We will show in Chapter 5 that this encoding is superior to $\Gamma_{\Pi,\mathbf{A}}$ for many applications. The effect becomes especially visible for programs which require many unfounded set checks, which is roughly the case if there exist many answer sets. Then the reusability of the encoding is valuable. In contrast, for very small programs with few answer sets, the higher costs for generating the encoding sometimes exceed the savings due to reusability.

3.2.3 Optimization and Learning

In this section we first discuss some refinements and optimizations of our encodings of the UFS search. In particular, we add additional nogoods which prune irrelevant parts of the search space. After that, we propose a strategy for learning nogoods from detected unfounded sets, avoiding that the same unfounded set is generated again later.

The following optimizations turned out to be effective in improving the UFS search.

O1: Restricting the UFS Search to Atoms in the Compatible Set. Not all atoms in a program are relevant for the unfounded set search. Formally one can show the following:

Proposition 3.12. *If U is an unfounded set of Π wrt. an interpretation $\mathbf{A}$ and there is an $a \in U$ s.t. $\mathbf{A} \not\models a$, then $U \setminus \{a\}$ is an unfounded set of Π wrt. $\mathbf{A}$.*

Proof. Let $r \in \Pi$ s.t. $H(r) \cap \big(U \setminus \{a\}\big) \neq \emptyset$. We have to show that one of the conditions of Definition 38 holds wrt. $\mathbf{A}$ and $U \setminus \{a\}$.

Because U is an unfounded set of Π wrt. $\mathbf{A}$ and $H(r) \cap \big(U \setminus \{a\}\big) \neq \emptyset$ implies $H(r) \cap U \neq \emptyset$, one of the conditions of Definition 38 holds wrt. $\mathbf{A}$ and U. If this is Condition (i) or (iii), it also holds wrt. $U \setminus \{a\}$ because these condition depend only on r and $\mathbf{A}$. Also if Condition (ii) holds, it also holds wrt. $U \setminus \{a\}$ because $\mathbf{A} \dot\cup \neg.U$ is equivalent to $\mathbf{A} \dot\cup \neg.\big(U \setminus \{a\}\big)$ since $a \notin U$. □

The construction of the nogoods which implement this optimization is simple.

- For the encoding $\Gamma_{\Pi,\mathbf{A}}$ we add the conservative nogood $\{\mathbf{T}a\}$ for each $a \in A(\Pi)$ with $\mathbf{A} \not\models a$.

- For the encoding Ω_Π we add the conservative nogood $\{\mathbf{F}a_\mathbf{A}, \mathbf{T}a\}$ for each $a \in A(\Pi)$.

O2: Avoiding Guesses of External Replacement Atoms. In some situations the truth value of an external replacement atom b in a solution $\mathbf{S}$ to $\Gamma_{\Pi,\mathbf{A}}$ does not matter. That is, both $\big(\mathbf{S} \setminus \{\mathbf{T}b, \mathbf{F}b\}\big) \cup \{\mathbf{T}b\}$ and $\big(\mathbf{S} \setminus \{\mathbf{T}b, \mathbf{F}b\}\big) \cup \{\mathbf{F}b\}$ are solutions to $\Gamma_{\Pi,\mathbf{A}}$ (resp. Ω_Π with assumptions $\mathcal{A}_\mathbf{A}$), which represent the same unfounded set. Then we can set the truth value to an (arbitrary) fixed value instead of inspecting both alternatives. The following provides a sufficient criterion:

Proposition 3.13. *Let b be an external replacement atom, and let $\mathbf{S}$ be a solution to $\Gamma_{\Pi,\mathbf{A}}$ (resp. Ω_Π with assumptions $\mathcal{A}_\mathbf{A}$). If for all rules $r \in \Pi$, such that $\mathbf{A} \models B(r)$ and where $b \in B^+(\hat{r})$ or $b \in B^-(\hat{r})$, either*

(a) for some $a \in B_o^+(r)$ such that $\mathbf{A} \models a$, it holds that $\mathbf{T}a \in \mathbf{S}$; or

(b) for some $a \in H(r)$ such that $\mathbf{A} \models a$, it holds that $\mathbf{F}a \in \mathbf{S}$

then both $\big(\mathbf{S} \setminus \{\mathbf{T}b, \mathbf{F}b\}\big) \cup \{\mathbf{T}b\}$ and $\big(\mathbf{S} \setminus \{\mathbf{T}b, \mathbf{F}b\}\big) \cup \{\mathbf{F}b\}$ are solutions to $\Gamma_{\Pi,\mathbf{A}}$ (resp. Ω_Π with assumptions $\mathcal{A}_\mathbf{A}$).

Proof. Suppose that changing the truth value of b in $\mathbf{S}$ turns the solution to a counterexample of $\Gamma_{\Pi,\mathbf{A}}$ (resp. Ω_Π). Then there must be a violated nogood $N \in \Gamma_{\Pi,\mathbf{A}}$ (resp. $N \in \Omega_\Pi$) containing b, i.e., $\mathbf{T}b \in N$ or $\mathbf{F}b \in N$.

For the encoding $\Gamma_{\Pi,\mathbf{A}}$, this nogood corresponds to a rule with $b \in B^+(\hat{r})$ or $b \in B^-(\hat{r})$ and $\mathbf{A} \models B(r)$, and it contains also the signed literals (1) $\mathbf{F}a$ for all $a \in B_o^+$ with $\mathbf{A} \models a$ and (2) $\mathbf{T}a$ for all $a \in H(r)$ with $\mathbf{A} \models a$. By the precondition of the proposition we have either (a) $\mathbf{T}a \in S$ for some $a \in B_o^+(r)$ with $\mathbf{A} \models a$, or (b) $\mathbf{F}a$ for some $a \in H(r)$ with $\mathbf{A} \models a$. But then the nogood cannot be violated, because (a) contradicts one of (1) and (b) contradicts one of (2).

For the encoding Ω_Π, this nogood also corresponds to a rule r with $b \in B^+(\hat{r})$ or $b \in B^-(\hat{r})$. The nogood contains also the signed literals (1) $\mathbf{T}a_\mathbf{A}$ for all $a \in B^+(\hat{r})$ and $\mathbf{F}a_\mathbf{A}$ for all $a \in B^-(\hat{r})$, (2) $\mathbf{F}a_{\mathbf{A}\wedge U}$ for all $a \in B_o^+$, and (3) $\mathbf{T}h_{\bar{\mathbf{A}}\vee U}$ for all $h \in H(r)$. Because of (1) and since $\mathbf{S}$ is a solution to $\mathcal{A}_\mathbf{A}$, $\mathbf{A} \models B(r)$. Then by the precondition of the proposition we have either (a) $\mathbf{T}a \in \mathbf{S}$ for some $a \in B_o^+(r)$ with $\mathbf{A} \models a$, or (b) $\mathbf{F}a$ for some $a \in H(r)$ with $\mathbf{A} \models a$. But then the nogood cannot be violated, because (a) contradicts one of (2) by definition of $\mathcal{A}_\mathbf{A}$ and $a_{\mathbf{A}\wedge U}$, and (b) contradicts one of (3) by definition of $\mathcal{A}_\mathbf{A}$ and $h_{\bar{\mathbf{A}}\vee U}$. □

This property can be utilized by adding the following additional nogoods. Recall that $A(\Gamma_{\Pi,\mathbf{A}})$ and $A(\Omega_\Pi)$ contain atoms l_r for every $r \in \Pi$. Intuitively, they are used to encode for a solution $\mathbf{S}$ to $\Gamma_{\Pi,\mathbf{A}}$ resp. Ω_Π with assumptions $\mathcal{A}_\mathbf{A}$, whether the truth values of the external atom replacements in $B(r)$ are relevant, or whether they can be set arbitrarily for r. The following nogoods label relevant rules r, forcing l_r to be false iff one of the preconditions in Proposition 3.13 holds.

- For the encoding $\Gamma_{\Pi,\mathbf{A}}$ we add the nogoods:

$$\Gamma^L_{r,\mathbf{A}} = \big\{\{\mathbf{T}l_r, \mathbf{T}a\} \mid a \in B^+_o(r), \mathbf{A} \models a\big\} \cup \big\{\{\mathbf{T}l_r, \mathbf{F}a\} \mid a \in H(r), \mathbf{A} \models a\big\} \cup \big\{\{\mathbf{F}l_r\} \cup \{\mathbf{F}a \mid a \in B^+_o(r), \mathbf{A} \models a\} \cup \{\mathbf{T}a \mid a \in H(r), \mathbf{A} \models a\}\big\}$$

- For the encoding Ω_Π we add the nogoods:

$$\Omega^L_r = \big\{\{\mathbf{T}l_r, \mathbf{T}a, \mathbf{T}a_\mathbf{A}\} \mid a \in B^+_o(r)\big\} \cup \big\{\{\mathbf{T}l_r, \mathbf{F}a, \mathbf{T}a_\mathbf{A}\} \mid a \in H(r)\big\} \cup \big\{\{\mathbf{F}l_r\} \cup \{\mathbf{F}a_{\mathbf{A}\wedge U} \mid a \in B^+_o(r)\} \cup \{\mathbf{T}a_{\bar{\mathbf{A}}\vee U} \mid a \in H(r)\}\big\}$$

These constraints exclusively enforce $\mathbf{T}l_r$ or $\mathbf{F}l_r$. Hence, the truth value of l_r deterministically depends on the other atoms, i.e., the nogoods do not cause additional guessing.

By Proposition 3.13 we can set the truth value of an external replacement atom b arbitrarily, if l_r is false for all r such that $b \in B^+(\hat{r})$ or $b \in B^-(\hat{r})$, and the resulting interpretation will still be a solution to $\Gamma_{\Pi,\mathbf{A}}$ (resp. Ω_Π). However, it must be ensured that changing the truth value of replacement atoms does not harm the satisfaction of the conditions in Proposition 3.7 (resp. Proposition 3.10).

As mentioned after Proposition 3.7, it is advantageous to set the truth value of $e_{\&g[\mathbf{y}]}(\mathbf{x})$ to the one of $\&g[\mathbf{y}](\mathbf{x})$ wrt. $\mathbf{A}$, because this can reduce the number of external atoms that must be checked. Importantly, this also relaxes the antecedence of the conditions in Proposition 3.7 (resp. Proposition 3.10) and guarantees that they are not harmed. The following nogoods enforce a coherent interpretation of the external replacement atoms.

- For the encoding $\Gamma_{\Pi,\mathbf{A}}$ we add the conservative nogoods:

$$\Gamma^F_{r,\mathbf{A}} = \big\{\{\mathbf{F}l_r \mid b \in B^+(\hat{r}) \cup B^-(\hat{r})\} \cup \{\mathbf{F}b\} \mid b \in B_e(\hat{r}), \mathbf{A} \models b\big\} \cup \big\{\{\mathbf{F}l_r \mid b \in B^+(\hat{r}) \cup B^-(\hat{r})\} \cup \{\mathbf{T}b\} \mid b \in B_e(\hat{r}), \mathbf{A} \not\models b\big\}$$

- For the encoding Ω_Π we add the conservative nogoods:

$$\Omega^F_r = \big\{\{\mathbf{F}l_r \mid b \in B^+(\hat{r}) \cup B^-(\hat{r})\} \cup \{\mathbf{T}b_\mathbf{A}, \mathbf{F}b\} \mid b \in B_e(\hat{r})\big\} \cup \big\{\{\mathbf{F}l_r \mid b \in B^+(\hat{r}) \cup B^-(\hat{r})\} \cup \{\mathbf{F}b_\mathbf{A}, \mathbf{T}b\} \mid b \in B_e(\hat{r})\big\}$$

We give now an example for this optimization using our encoding $\Gamma_{\Pi,\mathbf{A}}$.

Example 31. Consider the program $\Pi = \{r_1\colon\ p \leftarrow \&id[p]();\ r_2\colon\ q \leftarrow \&id[q]()\}$, and the compatible set $\hat{\mathbf{A}} = \{\mathbf{T}p, \mathbf{T}q, \mathbf{T}e_{\&id[p]}(), \mathbf{T}e_{\&id[q]}()\}$. The necessary part of the encoding

$$\Gamma_{\Pi,\mathbf{A}} = \big\{\{\mathbf{T}h_{r_1}, \mathbf{F}p\}, \{\mathbf{F}h_{r_1}, \mathbf{T}p\}, \{\mathbf{T}h_{r_1}, \mathbf{T}e_{\&id[p]}(), \mathbf{T}p\}, \{\mathbf{T}h_{r_2}, \mathbf{F}q\}, \{\mathbf{F}h_{r_2}, \mathbf{T}q\}, \{\mathbf{T}h_{r_2}, \mathbf{T}e_{\&id[q]}(), \mathbf{T}q\}\big\}$$

has the solutions $\mathbf{S}_1 \supseteq \{\mathbf{T}h_{r_1}, \mathbf{T}p, \mathbf{F}e_{\&id[p]}(), \mathbf{F}h_{r_2}, \mathbf{F}q, \mathbf{F}e_{\&id[q]}()\}$ and $\mathbf{S}_2 \supseteq \{\mathbf{T}h_{r_1}, \mathbf{T}p, \mathbf{F}e_{\&id[p]}(), \mathbf{F}h_{r_2}, \mathbf{F}q, \mathbf{T}e_{\&id[q]}()\}$ (which represent the same unfounded set $U = \{p\}$). Here, the optimization part for r_2, $\Gamma^L_{r_2,\mathbf{A}} \cup \Gamma^F_{r_2,\mathbf{A}} = \big\{\{\mathbf{T}l_{r_2}, \mathbf{F}q\}, \{\mathbf{F}l_{r_2}, \mathbf{T}q\}, \{\mathbf{F}l_{r_2}, \mathbf{T}e_{\&id[q]}()\}\big\}$, eliminates solutions $\mathbf{S}_2$ for $\Gamma_{\Pi,\mathbf{A}}$. This is beneficial as for solutions $\mathbf{S}_1$ the post-check is easier ($e_{\&id[q]}()$ in $\mathbf{S}_1$ and $\&id[q]()$ in $\mathbf{A}$ have the same truth value). □

Note that, strictly speaking, this optimization is *not* conservative, since for a UFS the induced assignment is not necessarily a solution to the UFS search encodings with this optimization. However, we have shown that there is a related solution for each UFS.

Also note that if this optimization is not used, then for all rules r the atom l_r is in fact not needed and thus unconstrained. To avoid an exponential increase of the number of UFS candidates, these atoms should then be set to a fixed value.

O3: Exchanging Nogoods between UFS and Main Search. Some nogoods learned from external sources during the search for compatible sets can be reused for the UFS search and vice versa. This is because such nogoods are independent of the program resp. SAT instance but depend only on the semantics of the external sources. For this purpose, we first define nogoods which correctly describe the input-output relationship of external atoms.

Definition 42. A nogood of the form $N = \{\mathbf{T}t_1, \ldots, \mathbf{T}t_n, \mathbf{F}f_1, \ldots, \mathbf{F}f_m, \sigma e_{\&g[\mathbf{y}]}(\mathbf{x})\}$, where σ is $\mathbf{T}$ or $\mathbf{F}$, is a *valid input-output relationship*, if for all assignments $\mathbf{A}$, $\mathbf{T}t_i \in \mathbf{A}$, for $1 \leq i \leq n$, and $\mathbf{F}f_i \in \mathbf{A}$, for $1 \leq i \leq m$, implies $\mathbf{A} \models \&g[\mathbf{y}](\mathbf{x})$ if $\sigma = \mathbf{F}$, and $\mathbf{A} \not\models \&g[\mathbf{y}](\mathbf{x})$ if $\sigma = \mathbf{T}$.

Here, the signed literals with atoms t_i for $1 \leq i \leq n$ and f_i for $1 \leq i \leq m$ reflect the relevant true resp. false atoms in the interpretation $\mathbf{A}$, built over predicates which occur in the input list $\mathbf{y}$.

Let N be a nogood which is a valid input-output relationship learned during the main search, i.e., the search for compatible sets of Π, and let $\bar{\mathbf{F}} = \mathbf{T}$ and $\bar{\mathbf{T}} = \mathbf{F}$.

Definition 43 (Nogood Transformation $\mathcal{T}_\Gamma$)**.** For a valid input-output relationship N and an assignment $\mathbf{A}$, the nogood transformation $\mathcal{T}_\Gamma$ is defined as

$$\mathcal{T}_\Gamma(N, \mathbf{A}) = \begin{cases} \emptyset & \text{if } \mathbf{F}t_i \in \mathbf{A} \text{ for some } 1 \leq i \leq n, \\ \big\{\{\mathbf{F}t_1, \ldots, \mathbf{F}t_n\} \cup & \\ \quad \{\mathbf{T}f_i \mid 1 \leq i \leq m, \mathbf{A} \models f_i\} \cup & \\ \quad \{\sigma e_{\&g[\mathbf{y}]}(\mathbf{x})\}\big\} & \text{otherwise.} \end{cases}$$

The next result states that $\mathcal{T}_\Gamma(N, \mathbf{A})$ can be considered, for all valid input-output relationships N wrt. all assignments $\mathbf{A}$, without losing unfounded sets.

Proposition 3.14. *Let N be a valid input-output relationship, and let U be an unfounded set wrt. Π and $\mathbf{A}$. If $\Gamma^O_{\Pi,\mathbf{A}}$ contains only conservative nogoods, then $I_\Gamma(U, \Gamma_{\Pi,\mathbf{A}}, \Pi, \mathbf{A})$ is a solution to $\mathcal{T}_\Gamma(N, \mathbf{A})$ (i.e., also the nogoods $\mathcal{T}_\Gamma(N, \mathbf{A})$ are conservative).*

Proof. If $\mathcal{T}_\Gamma(N, \mathbf{A}) = \emptyset$ then the proposition trivially holds. Otherwise $\mathcal{T}_\Gamma(N, \mathbf{A}) = \{C\}$ and we know that $\mathbf{T}t_i \in \mathbf{A}$ for all $1 \leq i \leq n$. Suppose C is violated. Then $\mathbf{F}t_i \in I_\Gamma(U, \Gamma_{\Pi,\mathbf{A}}, \Pi, \mathbf{A})$ and therefore $t_i \notin U$ for all $1 \leq i \leq n$, and $\mathbf{T}f_i \in I_\Gamma(U, \Gamma_{\Pi,\mathbf{A}}, \Pi, \mathbf{A})$ for all $1 \leq i \leq m$ with $\mathbf{A} \models f_i$, and $\sigma e_{\&g[\mathbf{y}]}(\mathbf{x}) \in I_\Gamma(U, \Gamma_{\Pi,\mathbf{A}}, \Pi, \mathbf{A})$.

But then $\mathbf{A} \dot\cup \neg.U \models t_i$ for all $1 \leq i \leq n$ and $\mathbf{A} \dot\cup \neg.U \not\models f_i$ for all $1 \leq i \leq m$. Because the nogood N is a valid input-output relationship, this implies $\mathbf{A} \dot\cup \neg.U \models \bar\sigma \&g[\mathbf{y}](\mathbf{x})$ iff $\sigma = \mathbf{F}$. Then by definition of $I_\Gamma(U, \Gamma_{\Pi,\mathbf{A}}, \Pi, \mathbf{A})$ we have $\bar\sigma e_{\&g[\mathbf{y}]}(\mathbf{x}) \in I_\Gamma(U, \Gamma_{\Pi,\mathbf{A}}, \Pi, \mathbf{A})$, which contradicts the assumption that $\mathcal{T}_\Gamma(N, \mathbf{A})$ is violated. □

Hence, all valid input-output relationships for external atoms which are learned during the search for compatible sets, can be reused (applying the above transformation) for the unfounded set check. Moreover, during the evaluation of external atoms in the post-check for candidate unfounded sets (solutions to $\Gamma_{\Pi,\mathbf{A}}$), further valid input-output relationships might be learned. These can in turn be used in further unfounded set checks (in transformed form) or directly in the main search.

Example 32 (Set Partitioning)**.** Consider the program Π

$$\begin{aligned} sel(a) &\leftarrow domain(a), \&diff[domain, nsel](a) \\ nsel(a) &\leftarrow domain(a), \&diff[domain, sel](a) \\ domain(a) &\leftarrow \end{aligned}$$

Informally, this program implements a choice from $sel(a)$ and $nsel(a)$. Consider the compatible set $\hat{\mathbf{A}} = \{\mathbf{T}domain(a), \mathbf{T}sel(a), \mathbf{T}e_{\&diff[nsel]}(a)\}$. Suppose the main search learned the input-output relationship $N = \{\mathbf{T}domain(a), \mathbf{F}nsel(a), \mathbf{F}e_{\&diff[nsel]}(a)\}$. Then the transformed nogood is $\mathcal{T}(N,\mathbf{A})=\{\{\mathbf{F}domain(a), \mathbf{F}e_{\&diff[nsel]}(a)\}\}$, which intuitively encodes that, if $domain(a)$ is *not* in the unfounded set U, then $e_{\&diff[nsel]}(a)$ is true in $\mathbf{A} \dot\cup \neg.U$. This is clear because $e_{\&diff[nsel]}(a)$ is true in $\mathbf{A}$ and it can only change its truth value if $domain(a)$ becomes false. □

Finally, an important note is that the optimizations O2 and O3 *can not* be used simultaneously (differently from O1 and O2 resp. O1 and O3), as this can result in contradictions due to (transformed) learned nogoods. We thus disabled O2 in our experiments.

This learning technique can be adopted for the encoding Ω_Π as follows.

Definition 44 (Nogood Transformation $\mathcal{T}_\Omega$)**.** For a valid input-output relationship N, the nogood transformation $\mathcal{T}_\Omega$ is defined as

$$\mathcal{T}_\Omega(N) = \{\{\mathbf{T}t_{1\mathbf{A}}, \mathbf{F}t_1, \ldots, \mathbf{T}t_{n\mathbf{A}}, \mathbf{F}t_n, \mathbf{F}f_{1\mathbf{A}\dot\cup\neg.U}, \ldots, \mathbf{F}f_{m\mathbf{A}\dot\cup\neg.U}, \sigma e_{\&g[\mathbf{y}]}(\mathbf{x})\}\}.$$

Compared to the nogood transformation $\mathcal{T}_\Gamma(N, \mathbf{A})$, the main difference is that $\mathcal{T}_\Omega(N)$ is reusable for any assignment, similar to the definition of our unfounded set detection problem Ω_Π.

The next result states that $\mathcal{T}_\Omega(N)$ can be considered, for all valid input-output relationships N wrt. all assignments $\mathbf{A}$, without losing unfounded sets.

Proposition 3.15. *Let N be a valid input-output relationship, and let U be an unfounded set wrt. Π and $\mathbf{A}$. If Ω^O_Π contains only conservative nogoods, then $I_\Omega(U, \Omega_\Pi, \Pi, \mathbf{A})$ is a solution to $\mathcal{T}_\Omega(N)$ (i.e., also nogoods $\mathcal{T}_\Omega(N)$ are conservative).*

Proof. We know $\mathcal{T}_\Omega(N) = \{C\}$ Suppose C is violated. Then $\mathbf{T}t_{i\mathbf{A}} \in I_\Omega(U, \Omega_\Pi, \Pi, \mathbf{A})$ and therefore $\mathbf{T}t_i \in \mathbf{A}$, $\mathbf{F}t_i \in I_\Omega(U, \Omega_\Pi, \Pi, \mathbf{A})$ for all $1 \leq i \leq n$, $\mathbf{F}f_{iR} \in I_\Omega(U, \Omega_\Pi, \Pi, \mathbf{A})$ and therefore $\mathbf{F}f_i \in \mathbf{A}$ for all $1 \leq i \leq m$, and $\sigma e_{\&g[\mathbf{y}]}(\mathbf{x}) \in I_\Omega(U, \Omega_\Pi, \Pi, \mathbf{A})$.

But then, by definition of $I_\Omega(U, \Omega_\Pi, \Pi, \mathbf{A})$, $\mathbf{T}t_i \in \mathbf{A}$ and $t_i \notin U$ for all $1 \leq i \leq n$, hence $\mathbf{A} \dot\cup \neg.U \models t_i$ for all $1 \leq i \leq n$. Moreover, $\mathbf{A} \dot\cup \neg.U \not\models f_i$ for all $1 \leq i \leq m$. Because nogood

N is a valid input-output relationship, this implies $\mathbf{A} \dot\cup \neg.U \models \&g[\mathbf{y}](\mathbf{x})$ iff $\sigma = \mathbf{F}$. Then by definition of $I_\Omega(U, \Omega_\Pi, \Pi, \mathbf{A})$ we have $\bar\sigma e_{\&g[\mathbf{y}]}(\mathbf{x}) \in I_\Omega(U, \Omega_\Pi, \Pi, \mathbf{A})$, which contradicts the assumption that $\mathcal{T}_\Omega(N)$ is violated. □

Hence, also with encoding Ω_Π all valid input-output relationships for external atoms that are learned during the search for compatible sets can be reused and vice versa.

Example 33 (ctd.)**.** Reconsider the program Π from Example 32. Consider the compatible set $\hat{\mathbf{A}} = \{\mathbf{T}domain(a), \mathbf{T}sel(a), \mathbf{T}e_{\&diff[nsel]}(a)\}$. Suppose the main search has learned the input-output relationship $N = \{\mathbf{T}domain(a),\ \mathbf{F}nsel(a), \mathbf{F}e_{\&diff[nsel]}(a)\}$. Then the transformed nogood is

$$\mathcal{T}_\Omega(N) = \big\{\{\mathbf{T}domain(a)_{\mathbf{A}}, \mathbf{F}domain(a), \mathbf{F}nsel(a)_{\mathbf{A}\dot\cup\neg.U}, \mathbf{F}e_{\&diff[nsel]}(a)\}\big\},$$

which intuitively encodes that, if $domain(a)$ is true in the current assignment but *not* in the unfounded set U, and $nsel(a)$ is false in $\mathbf{A} \dot\cup \neg.U$, then $e_{\&diff[nsel]}(a)$ is true in $\mathbf{A} \dot\cup \neg.U$. This is clear because $e_{\&diff[nsel]}(a)$ is true in $\mathbf{A}$ and it can only change its truth value if $domain(a)$ becomes false. □

The nogood exchange also benefits from the uniform encoding. With the encoding $\Gamma_{\Pi,\mathbf{A}}$ the SAT instance needs to be built from scratch for every unfounded set check. Thus, nogoods learned in the main search need to be transformed and added to the UFS detection problem for every check (otherwise they are lost). With encoding Ω_Π this needs to be done only once because the solver instance for UFS detection keeps running all the time and thus also learned nogoods are kept between multiple unfounded set checks. This also allows us the make use of advanced forgetting heuristics in SAT solvers more effectively.

Learning Nogoods from Unfounded Sets. Until now only detecting unfounded sets has been considered. A strategy to *learn* from detected unfounded sets for the main search for compatible sets is missing. Here we develop such a strategy and call it *unfounded set learning (UFL)*.

Example 34. Consider the program $\Pi = \{p \leftarrow \&id[p]();\ x_1 \vee x_2 \vee \cdots \vee x_k \leftarrow\}$. As we know from Example 26, $\{p\}$ is an unfounded set wrt. $\mathbf{A} = \{\mathbf{T}p, \mathbf{T}e_{\&id}()\}$, regarding just the first rule. However, the same is true for any $\mathbf{A}' \supset \mathbf{A}$ regarding Π, i.e., p must never be true. □

The program in Example 34 has many compatible sets, and half of them (all where p is true) will fail the UFS check for the same reason. We thus develop a strategy for generating additional nogoods to guide the further search for compatible sets in a way, such that the same unfounded sets are not reconsidered.

UFS-Based Learning. For an unfounded set U of Π wrt. $\mathbf{A}$ we define the following set of learned nogoods:

$$L_1(U, \Pi, \mathbf{A}) = \big\{\{\sigma_0, \sigma_1, \ldots, \sigma_j\} \mid \sigma_0 \in \{\mathbf{T}a \mid a \in U\}, \sigma_i \in H_i \text{ for all } 1 \leq i \leq j)\big\},$$

where $H_i = \{\mathbf{T}h \mid h \in H(r_i) \setminus U, \mathbf{A} \models h\} \cup \{\mathbf{F}b \mid b \in B_o^+(r_i), \mathbf{A} \not\models b\}$ and $\{r_1, \ldots, r_j\} = \{r \in \Pi \mid H(r) \cap U \neq \emptyset, U \cap B_o^+(r) = \emptyset\}$ is the set of external rules of Π wrt. U, i.e., all rules

which do not depend on U. Intuitively, the nogoods encode that no atom in the unfounded set must be true, if each rule, that could be used to derive it is already satisfied independently of the unfounded set.

Formally we can show that adding this set of nogoods does not eliminate answer sets of the program:

Proposition 3.16. *If U is an unfounded set of Π wrt. $\mathbf{A}$ and $\mathbf{A} \models \Pi$, then every answer set of Π is a solution to the nogoods in $L_1(U, \Pi, \mathbf{A})$.*

Proof. Suppose there is an answer set $\mathbf{A}'$ of Π which is not a solution to a nogood in $L_1(U, \Pi, \mathbf{A})$ We show that then U is an unfounded set of Π wrt. $\mathbf{A}'$ which intersects with $\mathbf{A}'$, contradicting the assumption that $\mathbf{A}'$ is an answer set.

Let $\{\sigma_0, \sigma_1, \ldots, \sigma_n\}$ be a violated nogood. Let $r \in \Pi$ be a rule such that $H(r) \cap U \neq \emptyset$. We have to show that one of the conditions of Definition 38 holds.

If $B_o^+(r) \cap U \neq \emptyset$, then Condition (ii) holds, therefore we can assume $B_o^+(r) \cap U = \emptyset$. Hence r is an external rule of Π wrt. U. But then there is a σ_i with $1 \leq i \leq n$ such that either (1) $\sigma_i = \mathbf{T}h$ for some $h \in H(r)$ with $h \notin U$ and $\mathbf{A} \models h$, or (2) $\sigma_i = \mathbf{F}b$ for some $b \in B_o^+(r)$ with $\mathbf{A} \not\models b$. Because the nogood is violated by $\mathbf{A}'$ by assumption, we have $\sigma_i \in \mathbf{A}'$. In Case (1) Condition (iii) is satisfied, in Case (2) Condition (i) is satisfied.

Moreover, by definition of L_1 there is an $a \in U$ s.t. $\mathbf{T}a \in \mathbf{A}'$, i.e., $\mathbf{A}'$ intersects with U. □

Example 35. Consider the program Π from Example 34 and suppose we have found the unfounded set $U = \{p\}$ wrt. interpretation $\mathbf{A} = \{\mathbf{T}p, \mathbf{T}x_1\} \cup \{\mathbf{F}x_i \mid 1 < i \leq k\}$. Then the learned nogood $L_2(U, \mathbf{A}, \Pi) = \{\mathbf{T}p\}$ immediately guides the search to the part of the search tree where p is false, i.e., roughly half of the guesses are avoided. □

Reduct-Based Learning. We may also consider a different learning strategy based on the models of $f\Pi^{\mathbf{A}}$ rather than the unfounded set U itself, hinging on the observation that for every unfounded set U, the interpretation $\mathbf{A} \dot\cup \neg.U$ is a model of $f\Pi^{\mathbf{A}}$ (hence $U \neq \emptyset$ refutes $\mathbf{A}$ as a minimal model of $f\Pi^{\mathbf{A}}$), cf. Faber et al. (2011).

We exploit this to construct nogoods from a nonempty UFS U wrt. a model $\mathbf{A}$ as follows. The interpretation $\mathbf{A} \dot\cup \neg.U$ is not only a model of $f\Pi^{\mathbf{A}}$ but a model of *all* programs $\Pi' \subseteq f\Pi^{\mathbf{A}}$. Hence, if an assignment $\mathbf{A}'$ falsifies the bodies of *at least* the same rules of Π as $\mathbf{A}$, and $\mathbf{A'}^{\mathbf{T}} \supset (\mathbf{A} \dot\cup \neg.U)^{\mathbf{T}}$, then $\mathbf{A}'$ cannot be an answer set of Π. This allows for generating the following set of nogoods:

$$L_2(U, \Pi, \mathbf{A}) = \big\{\{\mathbf{T}a \mid a \in \mathbf{A} \dot\cup \neg.U\} \cup \{\sigma_0, \sigma_1, \ldots, \sigma_j\} \\ \mid \sigma_0 \in \{\mathbf{T}a \mid a \in U\}, \sigma_i \in H_i \text{ for all } 1 \leq i \leq j\big\},$$

where $H_i = \{\mathbf{t}a \mid a \in B(\hat{r}), \hat{\mathbf{A}} \not\models a\}$ for all $1 \leq i \leq j$ and $\{r_1, \ldots, r_j \mid r \in \Pi, \mathbf{A} \not\models B(r)\}$ is the set of rules which are *not* in the FLP-reduct of Π wrt. $\mathbf{A}$.

That is, each nogood consists of the positive atoms from the smaller model of the reduct $\mathbf{A} \dot\cup \neg.U$, one unfounded atom σ_0 (i.e. an atom which is true in $\mathbf{A}$ but not in $\mathbf{A} \dot\cup \neg.U$), and one false body literal σ_i ($1 \leq i \leq j$) for each rule of Π with unsatisfied body wrt. $\mathbf{A}$.

Formally, we can show the following proposition:

Proposition 3.17. *If U is an unfounded set of Π wrt. $\mathbf{A}$ and $\mathbf{A} \models \Pi$, then each answer set of Π is a solution to all nogoods in $L_2(U, \Pi, \mathbf{A})$.*

Proof. Suppose there is an answer set $\mathbf{A}'$ of Π which is not a solution to the nogoods in $L_2(U, \Pi, \mathbf{A})$. Let $\{\mathbf{T}a \mid a \in \mathbf{A} \dot\cup \neg.U\} \cup \{\sigma_0, \sigma_1, \ldots, \sigma_n\}$ be a violated nogood. Because $\sigma_i \in \mathbf{A}'$ for all $1 \leq i \leq n$, we know $\mathbf{A}'$ falsifies (at least) all rules falsified by $\mathbf{A}$, thus $f\Pi^{\mathbf{A}'} \subseteq f\Pi^{\mathbf{A}}$. But then $\mathbf{A} \dot\cup \neg.U$ is a model of $\Pi^{\mathbf{A}'}$ because it is a model of $\Pi^{\mathbf{A}}$ by assumption that it is an unfounded set. Moreover, $\mathbf{T}a \in \mathbf{A}'$ for all $a \in \mathbf{A} \dot\cup \neg.U$, and therefore $\mathbf{A}'^{\mathbf{T}} \supseteq (\mathbf{A} \dot\cup \neg.U)^{\mathbf{T}}$. Because $\sigma_0 \in \mathbf{A}'$, we conclude $\mathbf{A}'^{\mathbf{T}} \supsetneq (\mathbf{A} \dot\cup \neg.U)^{\mathbf{T}}$, i.e., $\mathbf{A}'$ is not a subset-minimal model of $\Pi^{\mathbf{A}'}$. □

Example 36. Let $\Pi = \{p \leftarrow \&id[p](); \; q \leftarrow \&id[q]()\}$, where $\&id[a]()$ evaluates to true iff$_{\text{def}}$ a is true. Suppose $\mathbf{A} = \{p, q\}$. Then $U = \{p, q\}$ is an unfounded set wrt. $\mathbf{A}$. In the above construction rule we have $\mathbf{A} \dot\cup \neg.U = \{\mathbf{F}p, \mathbf{F}q\}$, $\sigma_0 \in \{\mathbf{T}p, \mathbf{T}q\}$ and $j = 0$ (because both rule bodies are satisfied wrt. $\mathbf{A}$). The learned nogoods are $\{\mathbf{T}p\}$ and $\{\mathbf{T}q\}$. □

In Example 36, the learned nogoods will immediately guide the search to the interpretation $\{\mathbf{F}p, \mathbf{F}q\}$, which is the only one which becomes an answer set. However, this strategy appeared to be clearly inferior to the UFS-based learning strategy, basically because the nogoods are too specific for the currently detected unfounded set. That is, they do not generalize to other unfounded sets.

3.2.4 Unfounded Set Check wrt. Partial Assignments

In some cases, a detected unfounded set wrt. a partial assignment implies the existence of an unfounded set wrt. any completion of that assignment. Clearly, a search for unfounded sets wrt. incomplete assignments is only useful if we can be sure that detected unfounded sets will remain unfounded for arbitrary completions of the assignment.

Formally, we can show the following:

Proposition 3.18. *Let Π' be a program, let $\mathbf{A}'$ be an assignment which is complete on Π', and let U be an unfounded set of Π' wrt. $\mathbf{A}'$. If $\Pi \supseteq \Pi'$ such that $U \cap H(r) = \emptyset$ for all $r \in \Pi \setminus \Pi'$, then U is an unfounded set of Π wrt. any interpretation $\mathbf{A} \supseteq \mathbf{A}'$.*

Proof. Let Π' be a program, $\mathbf{A}'$ be an assignment which is complete on Π', and U be an unfounded set of Π' wrt. $\mathbf{A}'$. Further let $\Pi \supseteq \Pi'$ and $\mathbf{A} \supseteq \mathbf{A}'$.

We have to show that if $U \cap H(r) \neq \emptyset$ for some $r \in \Pi$, then one of the conditions in Definition 38 holds wrt. $\mathbf{A}$ and U. Let $r \in \Pi$. If $r \in \Pi'$, then one of the conditions holds because U is an unfounded set of Π' wrt. $\mathbf{A}'$ and $\mathbf{A}'$ is complete on Π'. If $r \notin \Pi'$, then $U \cap H(r) = \emptyset$ by assumption. □

Intuitively, the proposition states that an unfounded set of a program wrt. some interpretation will remain an unfounded set if the program is extended by rules which do not derive any of the elements in the unfounded set.

Corollary 3.3. *If some assignment $\mathbf{A}'$ is complete on a subprogram $\Pi' \subseteq \Pi$ and Π' has an unfounded set U wrt. $\mathbf{A}'$ which intersects with $\mathbf{A}'$ and such that $U \cap H(r) = \emptyset$ for all $r \in \Pi \setminus \Pi'$, then no $\mathbf{A} \supseteq \mathbf{A}'$ is an answer-set of Π.*

Proof. By Proposition 3.18, U is also an unfounded set of Π wrt. any $\mathbf{A} \supseteq \mathbf{A}'$. But then by Theorem 2, $\mathbf{A}$ cannot be an answer set of Π. □

This result can be used as follows. For a partial assignments $\mathbf{A}'$ which is complete on a subprogram $\Pi' \subseteq \Pi$, if $\Gamma_{\Pi',\mathbf{A}'} \cup \big\{\{\mathbf{F}a\} \mid r \in \Pi \setminus \Pi', a \in H(r)\big\}$ resp. $\Omega_{\Pi'} \cup \big\{\{\mathbf{F}a\} \mid r \in \Pi \setminus \Pi', a \in H(r)\big\}$ with assumptions $\mathcal{A}_{\mathbf{A}}$ has a solution which passes the post-check, then no completion of $\mathbf{A}'$ can be an answer set of Π.

Example 37. Consider again the program Π from Examples 34 and 35 and suppose we have the partial interpretation $\mathbf{A}' = \{p\}$, i.e., the guess over the x_i for $1 \leq i \leq n$ was not done yet. Nevertheless, we can already make an unfounded set check over the subprogram $\Pi' = \{p \leftarrow \&id[p]()\}$ because $\mathbf{A}'$ is complete over this program. The detected unfounded set $U = \{p\}$ does not intersect with the head of a rule $r \in \Pi \setminus \Pi' = \{x_1 \vee x_2 \vee \ldots x_n \leftarrow\}$. Therefore U is also an unfounded set of Π wrt. arbitrary completions of $\mathbf{A}'$ and we can immediatly backtrack, e.g., by learning $L_2(U, \mathbf{A}', \Pi')$. □

Corollary 3.3 ensures that Part (d) in Algorithm Hex-CDNL does not affect the correctness of the overall algorithm as stated by Proposition 3.2 and Theorem 1.

3.2.5 Deciding the Necessity of the UFS Check

Although the minimality check based on unfounded sets is more efficient than the explicit minimality check, computational costs are still high. Moreover, during evaluation of $\hat{\Pi}$ for computing the compatible set $\hat{\mathbf{A}}$, the ordinary ASP solver in Algorithm Hex-CDNL has already made an unfounded set check, and we can safely assume that it is founded from its perspective. Hence, all remaining unfounded sets which were not discovered by the ordinary ASP solver have to involve external sources, as their behavior is not fully captured by the ASP solver.

In this section we formalize these ideas and define a decision criterion which allows for deciding whether a further UFS check is necessary for a given program. We eventually define a class of programs which does not require an additional unfounded set check. Intuitively, we show that every unfounded set that is not already detected during the construction of $\hat{\mathbf{A}}$ contains input atoms of external atoms which are involved in cycles. If no such input atom exists in the program, then the UFS check is superfluous.

We start with a definition of atom dependency. Note that this definition is different from Definition 20 in Chapter 2 because it captures only positive dependencies; in this subsection we *never* use dependencies according to Chapter 2.

Definition 45 (Positive Atom Dependencies)**.** For a ground program Π, and ground atoms $p(\mathbf{c})$ and $q(\mathbf{d})$, we say that

(i) $p(\mathbf{c})$ *depends (positively)* on $q(\mathbf{d})$, denoted $p(\mathbf{c}) \rightarrow_p q(\mathbf{d})$, if for some rule $r \in \Pi$ we have $p(\mathbf{c}) \in H(r)$ and $q(\mathbf{d}) \in B^+(r)$; and

(ii) $p(\mathbf{c})$ *depends externally* on $q(\mathbf{d})$, denoted $p(\mathbf{c}) \rightarrow_p^e q(\mathbf{d})$, if for some rule $r \in \Pi$ we have $p(\mathbf{c}) \in H(r)$ and there is a $\&g[q_1, \ldots, q_n](\mathbf{e}) \in B^+(r) \cup B^-(r)$ with $q_i = q$ for some $1 \leq i \leq n$.

In the following, we consider *positive atom dependency graphs* G_Π^R for a ground program Π, where the set of vertices is the set of all ground atoms, and the set of edges is given by the binary relation $R = \rightarrow_p \cup \rightarrow_p^e$, whose elements are also called *ordinary edges* and *e-edges*, respectively.

The next definition and lemma allow to restrict our attention to the *core* of an unfounded set, i.e., its most essential part. For our purpose, we can then focus on such cores, disregarding atoms in a cut of G_Π^R which is defined as follows.

Definition 46 (Cut)**.** Let U be an unfounded set of Π wrt. $\mathbf{A}$. A set of atoms $C \subseteq U$ is called a *cut* of G_Π^R, if

(i) $b \not\rightarrow_p^e a$, for all $a \in C$ and $b \in U$ (C has no incoming or internal e-edges); and

(ii) $b \not\rightarrow_p a$ and $a \not\rightarrow_p b$, for all $a \in C$ and $b \in U \setminus C$ (there are no ordinary edges between C and $U \setminus C$).

We first prove that cuts can be removed from unfounded sets and the resulting set is still an unfounded set.

Lemma 3.6 (Unfounded Set Reduction)**.** *Let U be an unfounded set of Π wrt. an interpretation $\mathbf{A}$, and let C be a cut of G_Π^R. Then, $Y = U \setminus C$ is an unfounded set of Π wrt. $\mathbf{A}$.*

Proof. If $Y = \emptyset$, then the result holds trivially. Otherwise, let $r \in \Pi$ with $H(r) \cap Y \neq \emptyset$. We show that one of the conditions in Definition 38 holds. Observe that $H(r) \cap U \neq \emptyset$ because $U \supseteq Y$. Since U is an unfounded set of Π wrt. $\mathbf{A}$, one of the conditions of Definition 38 holds.

If Condition (i) holds, then the condition also holds wrt. Y.

If Condition (ii) holds, let $a \in H(r)$ such that $a \in Y$, and $b \in B(r)$ such that $\mathbf{A} \dot\cup \neg.U \not\models b$. We make a case distinction: either b is an ordinary literal or an external one.

If it is an ordinary default-negated atom $\mathrm{not}\, c$, then $\mathbf{A} \dot\cup \neg.U \not\models b$ implies $\mathbf{T}c \in \mathbf{A}$ and $c \notin U$, and therefore also $\mathbf{A} \dot\cup \neg.Y \not\models b$. So assume b is an ordinary atom. If $b \notin U$ then $\mathbf{A} \not\models b$ and the case for (i) applies, so assume $b \in U$. Because $a \in H(r)$ and $b \in B^+(r)$, we have $a \rightarrow_p b$ and therefore either $a, b \in C$ or $a, b \in Y$ (because there are no ordinary edges between C and Y). But by assumption $a \in Y$, and therefore $b \in Y$, hence $\mathbf{A} \dot\cup \neg.Y \not\models b$.

If b is an external literal, then there is no $q \in U$ with $a \rightarrow_p^e q$ and $q \notin Y$. Otherwise, this would imply $q \in C$ and C would have an incoming e-edge, which contradicts the assumption that C is a cut of G_Π^R. Hence, for all $q \in U$ with $a \rightarrow_p^e q$, also $q \in Y$, and therefore the truth value of b wrt. $\mathbf{A} \dot\cup \neg.U$ and $\mathbf{A} \dot\cup \neg.Y$ is the same. Hence $\mathbf{A} \dot\cup \neg.Y \not\models b$.

If Condition (iii) holds, then also $\mathbf{A} \models h$ for some $h \in H(r) \setminus Y$ because $Y \subseteq U$ and therefore $H(r) \setminus Y \supseteq H(r) \setminus U$. □

Example 38. Consider the following program (visualized in Figure 3.2):

$$\Pi = \{r \leftarrow \&id[r](); \; p \leftarrow \&id[r](); \; p \leftarrow q; \; q \leftarrow p\}$$

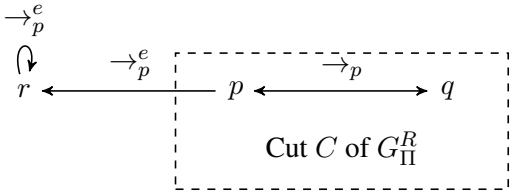


Figure 3.2: Dependencies and Cut of the Program Π from Example 38

Then we have $p \to_p q$, $q \to_p p$, $r \to_p^e r$ and $p \to_p^e r$. Program Π has the unfounded set $U = \{p, q, r\}$ wrt. $\mathbf{A} = \{\mathbf{T}p, \mathbf{T}q, \mathbf{T}r\}$. Observe that $C = \{p, q\}$ is a cut of G^R_Π, and therefore $U \setminus C = \{r\}$ is an unfounded set of Π wrt. $\mathbf{A}$. □

Next we prove that for each unfounded set U of Π, intuitively, either the input to some external atom is unfounded itself, or U is already detected when $\hat{\Pi}$ is evaluated.

Lemma 3.7 (EA-Input Unfoundedness)**.** *Let U be an unfounded set of Π wrt. an assignment $\mathbf{A}$. If G^R_Π has no edge $x \to_p^e y$ such that $x, y \in U$, then U is an unfounded set of $\hat{\Pi}$ wrt. the compatible set $\hat{\mathbf{A}}$ corresponding to $\mathbf{A}$.*

Proof. If $U = \emptyset$, then the result holds trivially. Otherwise, let $\hat{r} \in \hat{\Pi}$ such that $H(\hat{r}) \cap U \neq \emptyset$. Let $a \in H(\hat{r}) \cap U$. Observe that $\hat{r}$ cannot be an external atom guessing rule because U contains only ordinary atoms. We show that one of the conditions in Definition 38 holds for $\hat{r}$ wrt. $\hat{\mathbf{A}}$.

Because $\hat{r}$ is no external atom guessing rule, there is a corresponding rule $r \in \Pi$ containing external atoms in place of replacement atoms. Because U is an unfounded set of Π and $H(r) = H(\hat{r})$, one of the conditions of Definition 38 holds.

If Condition (i) holds, let $b \in B(r)$ such that $\mathbf{A} \not\models b$ and $\hat{b}$ the corresponding literal in $B(\hat{b})$ (which is the same if b is ordinary and the corresponding replacement literal if b is external). Then also $\hat{\mathbf{A}} \not\models \hat{b}$ because $\hat{\mathbf{A}}$ is compatible.

If Condition (ii) holds, let $b \in B(r)$ such that $\mathbf{A} \not\models b$. We make a case distinction: either b is ordinary or external.

If b is ordinary, then $b \in B(\hat{r})$ and $\hat{\mathbf{A}} \dot\cup \neg.U \not\models b$ holds because $\mathbf{A}$ and $\hat{\mathbf{A}}$ are equivalent for ordinary atoms.

If b is an external atom or default-negated external atom, then no atom $p(\mathbf{c}) \in U$ is input to it, i.e. p is not a predicate input parameter of b; otherwise we had $a \to_p^e p(\mathbf{c})$, contradicting our assumption that U has no internal e-edges. But then $\mathbf{A} \dot\cup \neg.U \not\models b$ implies $\mathbf{A} \not\models b$ because the truth value of b in $\mathbf{A} \dot\cup \neg.U$ and $\mathbf{A}$ is the same. Therefore we can apply the case for (i).

If Condition (iii) holds, then also $\hat{\mathbf{A}} \models h$ for some $h \in H(\hat{r}) \setminus U$ because $H(r) = H(\hat{r})$ contains only ordinary atoms and $\mathbf{A}$ is equivalent to $\hat{\mathbf{A}}$ for ordinary atoms. □

Example 39. Reconsider the program Π from Example 38. Then the unfounded set $U' = \{p, q\}$ wrt. $\mathbf{A}' = \{\mathbf{T}p, \mathbf{T}q, \mathbf{F}r\}$ is already detected when

$$\hat{\Pi} = \{e_{\&id[r]}() \vee ne_{\&id[r]}() \leftarrow;\ r \leftarrow e_{\&id[r]}();\ p \leftarrow e_{\&id[r]}();\ p \leftarrow q;\ q \leftarrow p\}$$

is evaluated by the ordinary ASP solver because $p \rightarrow_p^e q \notin R$ and $q \rightarrow_p^e p \notin R$. In contrast, the unfounded set $U'' = \{p, q, r\}$ wrt. $\mathbf{A}'' = \{\mathbf{T}p, \mathbf{T}q, \mathbf{T}r\}$ is *not* detected by the ordinary ASP solver because $p, r \in U''$ and $p \rightarrow_p^e r$[2]. □

The essential property of unfounded sets of Π wrt. $\mathbf{A}$, that are not recognized during the evaluation of $\hat{\Pi}$, is cyclic dependencies including input atoms of some external atom. Towards a formal characterization of a class of programs without this property, i.e., that do not require additional UFS checks, we define cycles as follows.

Definition 47 (Cycle)**.** A *cycle* wrt. a binary relation $\circ$ is a sequence $C = c_0, c_1, \ldots, c_n, c_{n+1}$ of elements with $n \geq 0$, such that $(c_i, c_{i+1}) \in \circ$ for all $0 \leq i \leq n$ and $c_0 = c_{n+1}$. We say that a set S *contains a cycle* wrt. $\circ$, if there is a cycle $C = c_0, c_1, \ldots, c_n, c_{n+1}$ wrt. $\circ$ such that $c_i \in S$ for all $0 \leq i \leq n + 1$.

The following proposition states, intuitively, that each unfounded set U of Π wrt. $\mathbf{A}$, which contains no cycle through the input atoms to some external atom, has a corresponding unfounded set U' of $\hat{\Pi}$ wrt. $\hat{\mathbf{A}}$. That is, the unfoundedness is already detected when $\hat{\Pi}$ is evaluated.

Let $\rightarrow_p^d = \rightarrow_p \cup \leftarrow_p \cup \rightarrow_p^e$, where $\leftarrow_p$ is the inverse of $\rightarrow_p$, i.e. $\leftarrow_p = \{(x, y) \mid (y, x) \in \rightarrow_p\}$. A cycle $c_0, c_1, \ldots, c_n, c_{n+1}$ wrt. $\rightarrow_p^d$ is called an *e-cycle*, if it contains e-edges, i.e., if $(c_i, c_{i+1}) \in \rightarrow_p^e$ for some $0 \leq i \leq n$. We say that a set S *contains e-edges*, if there are $x, y \in S$ such that $(x, y) \in \rightarrow_p^e$.

Proposition 3.19 (Relevance of e-cycles)**.** *Let $U \neq \emptyset$ be an unfounded set of Π wrt. an interpretation $\mathbf{A}$ such that $\mathbf{A}^{\mathbf{T}}$ does not contain any e-cycle wrt. $\rightarrow_p^d$. Then, there exists a nonempty unfounded set of $\hat{\Pi}$ wrt. $\hat{\mathbf{A}}$.*

Proof. We define the *reachable set* $R(a)$ from some atom a as

$$R(a) = \{b \mid (a, b) \in \{\rightarrow_p \cup \leftarrow_p\}^*\},$$

where $\{\rightarrow_p \cup \leftarrow_p\}^*$ is the reflexive and transitive closure of $\rightarrow_p \cup \leftarrow_p$, i.e., $R(a)$ is the set of atoms $b \in U$ reachable from a using edges from $\rightarrow_p \cup \leftarrow_p$ only but no e-edges.

We first assume that U contains at least one e-edge, i.e. there are $x, y \in U$ such that $x \rightarrow_p^e y$. Now we show that there is a $u \in U$ with outgoing e-edge (i.e. $u \rightarrow_p^e v$ for some $v \in U$), but such that $R(u)$ has no incoming e-edges (i.e. for all $v \in R(u)$ and $b \in U$, $b \not\rightarrow_p^e v$ holds). Suppose to the contrary that for all a with outgoing e-edges, the reachable set $R(a)$ has an incoming e-edge. We now construct an e-cycle wrt. $\rightarrow_p^d$, which contradicts our assumption. Start with an arbitrary node with an outgoing e-edge $c_0 \in U$ and let p_0 be the (possibly empty) path (wrt. $\rightarrow_p \cup \leftarrow_p$) from c_0 to the node $d_0 \in R(c_0)$ such that d_0 has an incoming e-edge, i.e. there is a c_1 such that $c_1 \rightarrow_p^e d_0$. Note that $c_1 \notin R(c_0)$: whenever $x \rightarrow_p^e y$ for $x, y \in U$, then there is no path from x to y wrt. $\rightarrow_p \cup \leftarrow_p$, because otherwise we would immediately have an e-cycle wrt. $\rightarrow_p^d$.

By assumption, also some node d_1 in $R(c_1)$ has an incoming e-edge (from some node $c_2 \notin R(c_1)$). Let p_1 be the path from c_1 to d_1, etc. By iteration we can construct the concatenation

[2]Out formal results only imply that it is not *necessarily* detected. However, it is easy to verify that U'' is indeed not an unfounded set of $\hat{\Pi}$ wrt. $\hat{\mathbf{A}}'' = \mathbf{A}'' \cup \{\mathbf{T}e_{\&id[r]}()\}$.

of the paths $p_0, (d_0, c_1), p_1, (d_1, c_2), p_2, \ldots, p_i, (d_i, c_{i+1}), \ldots$, where the p_i from c_i to d_i are the paths within reachable sets, and the (d_i, c_{i+1}) are the e-edges between reachable sets. However, as U is finite, some nodes on this path must be equal, i.e., a subsequence of the constructed sequence represents an e-cycle (in reverse order).

This proves that there is a node u with outgoing e-edge but such that $R(u)$ has no incoming e-edges. We next show that $R(u)$ is a cut of G^R_Π. Condition (i) is immediately satisfied by selection of u. Condition (ii) is shown as follows. Let $u' \in R(u)$ and $v' \in U \setminus R(u)$. We have to show that $u' \not\to_p v'$ and $v' \not\to_p u'$. Suppose, towards a contradiction, that $u' \to_p v'$. Because of $u' \in R(u)$, there is a path from u to u' wrt. $\to_p \cup \leftarrow_p$. But if $u' \to_p v'$, then there would also be a path from u to v' wrt. $\to_p \cup \leftarrow_p$ and v' would be in $R(u)$, a contradiction. Analogously, $v' \to_p u'$ would also imply that there is a path from u to v' because there is a path from u to u', again a contradiction.

Therefore, $R(u) \subseteq U$ is a cut of G^R_Π, and by Lemma 3.6, it follows that $U \setminus R(u)$ is an unfounded set. Observe that $U \setminus R(u)$ contains one e-edge less than U because u has an outgoing e-edge and is removed from the unfounded set. Further observe that $U \setminus R(u) \neq \emptyset$ because there is a $w \in U$ such that $u \to^e_p w$ but $w \notin R(u)$. By iterating this argument, the number of e-edges in the unfounded set can be reduced to zero in a nonempty core.

Eventually, or if the unfounded set did not contain any e-edges already at the beginning, Lemma 3.7 applies, proving that the remaining set is an unfounded set of $\hat{\Pi}$. □

Corollary 3.4. *If there is no e-cycle wrt. $\to^d_p$ and $\hat{\Pi}$ has no nonempty unfounded set wrt. $\hat{\mathbf{A}}$, then $\mathbf{A}$ is unfounded-free for Π.*

Proof. Suppose there is an unfounded set U of Π wrt. $\mathbf{A}$. Then it contains no e-cycle because there is no e-cycle wrt. $\to^d_p$. Then by Proposition 3.19 there is an unfounded set of $\hat{\Pi}$ wrt. $\hat{\mathbf{A}}$, which contradicts our assumption. □

This corollary can be used as follows to increase performance of an evaluation algorithm: if there is no cycle wrt. $\to^d_p$ containing e-edges, then an explicit unfounded set check is not necessary because the unfounded set check made during evaluation of $\hat{\Pi}$ suffices. Note that this test can be done efficiently (in fact in linear time, similar to deciding stratifiability of an ordinary logic program). Moreover, in practice one can abstract from $\to^d_p$ by using analogous relations on the level of predicates instead of atoms. Clearly, if there is no e-cycle in the predicate dependency graph, then there can also be no e-cycle in the atom dependency graph. Hence, the predicate dependency graph can be used to decide whether the unfounded set check can be skipped. In our implementation the check is done on the atom level.

Example 40. The program $\Pi = \{out(X) \leftarrow \&diff[set1, set2](X)\} \cup F$ does not require an unfounded set check for any set of facts F because there is no e-cycle wrt. $\to^d_p$, where *diff* computes the set difference of the extensions of $set1$ and $set2$.

Also $\Pi = \{str(Z) \leftarrow dom(Z), str(X), str(Y), \text{not } \&concat[X, Y](Z)\}$ does not need such a check; there is a cycle over an external atom, but no e-cycle wrt. $\to^d_p$. □

Note that Corollary 3.4 amounts to a *static* analysis of e-cycles in the program, whereas Proposition 3.19 has a *dynamic* view, i.e., it takes also the current unfounded set (and thus the

assignment) into account. The direct application of Proposition 3.19 possibly eliminates more unnecessary unfounded set checks than Corollary 3.4 because e-cycles in the program may be broken with respect to the assignment. However, in order to apply Proposition 3.19 directly, the existence of e-cycles has to be decided before every unfounded set check, and the computational overhead might easily exceed the savings due to avoided unfounded set checks, whereas the static approach requires only one such check. Therefore we have decided to apply the decision criterion in form of Corollary 3.4. A closer analysis of the effects of direct application of Proposition 3.19 is up to future work.

Moreover, the following proposition states that, intuitively, if $\hat{\Pi}$ has no unfounded sets wrt. $\hat{\mathbf{A}}$, then any unfounded set U of Π wrt. $\mathbf{A}$ must contain an atom which is involved in a cycle wrt. $\rightarrow_p^d$ that has an e-edge.

Definition 48 (Cyclic Input Atoms)**.** For a program Π, an atom a is a *cyclic input atom*, if there is an atom b such that $b \rightarrow_p^e a$ and there is a path from a to b wrt. $\rightarrow_p^d$.

Let $CA(\Pi)$ denote the set of all cyclic input atoms of program Π.

Proposition 3.20 (Unfoundedness of Cyclic Input Atom)**.** *Let $U \neq \emptyset$ be an unfounded set of Π wrt. $\mathbf{A}$ such that U does not contain cyclic input atoms. Then, $\hat{\Pi}$ has a nonempty unfounded set wrt. $\hat{\mathbf{A}}$.*

Proof. If U contains no cyclic input atoms, then all cycles wrt. $\rightarrow_p^d$ containing e-edges in the atom dependency graph of Π are broken, i.e., U does not contain an e-cycle wrt. $\rightarrow_p^d$. Then by Proposition 3.19 there is an unfounded set of $\hat{\Pi}$ wrt. $\hat{\mathbf{A}}$. □

Proposition 3.20 allows for generating the additional nogood $\{\mathbf{F}a \mid a \in CA(\Pi)\}$ and adding it to $\Gamma_{\Pi,\mathbf{A}}$. Again, considering predicates instead of atoms is possible to reduce the overhead introduced by the dependency graph.

3.2.6 Program Decomposition

The usefulness of the decision criterion can be increased by decomposing the program into components such that the criterion can be applied component-wise. This allows for restricting the unfounded set check to components with e-cycles, whereas e-cycle-free components can be ignored in the check.

Related to our splitting set technique is the work by Drescher et al. (2008), where a similar program decomposition is used, yet for ordinary programs only. While we consider e-cycles, which are specific for HEX-programs, the interest of Drescher et al. (2008) is with head-cycles with respect to disjunctive rule heads. In fact, our implementation may be regarded as an extension of their work since the evaluation of $\hat{\Pi}$ follows their principles of performing UFS checks in case of head-cycles. Note that our splitting is also different from the well-known splitting technique [Lifschitz and Turner, 1994] as we consider only positive dependencies for ordinary atoms.

Let $Comp$ be a partitioning of the ordinary atoms $A(\Pi)$ of Π into subset-maximal strongly connected components wrt. $\rightarrow_p \cup \rightarrow_p^e$. We define for each partition $C \in Comp$ the subprogram Π_C *associated with* C as $\Pi_C = \{r \in \Pi \mid H(r) \cap C \neq \emptyset\}$.

We next show that if a program has an unfounded set U wrt. $\mathbf{A}$, then $U \cap C$ is an unfounded set wrt. $\mathbf{A}$ for the subprogram Π_C associated with some strongly connected component C.

Proposition 3.21. *Let $U \neq \emptyset$ be an unfounded set of Π wrt. $\mathbf{A}$. Then, for some Π_C with $C \in Comp$ it holds that $U \cap C$ is a nonempty unfounded set of Π_C wrt. $\mathbf{A}$.*

Proof. Let U be a nonempty unfounded set of Π wrt. $\mathbf{A}$. Because $Comp$ is a decomposition of $A(\Pi)$ into strongly connected components, the *component dependency graph*

$$\left\langle Comp, \{(C_1, C_2) \mid C_1, C_2 \in Comp, \exists a_1 \in C_1, a_2 \in C_2 : (a_1, a_2) \in \rightarrow_p \cup \rightarrow_p^e\}\right\rangle$$

is acyclic. Following the hierarchical component dependency graph from the nodes without predecessor components downwards, we can find a 'first' component which has a nonempty intersection with U, i.e., there exists a component $C \in Comp$ such that $C \cap U \neq \emptyset$ but $C' \cap U = \emptyset$ for all transitive predecessor components C' of C.

We show that $U \cap C$ is an unfounded set of Π_C wrt. $\mathbf{A}$. Let $r \in \Pi_C$ be a rule such that $H(r) \cap (U \cap C) \neq \emptyset$. We have to show that one of the conditions of Definition 38 holds for r wrt. $\mathbf{A}$ and $U \cap C$.

Because U is an unfounded set of Π wrt. $\mathbf{A}$ and $H(r) \cap (U \cap C) \neq \emptyset$ implies $H(r) \cap U \neq \emptyset$, we know that one of the conditions holds for r wrt. $\mathbf{A}$ and U. If this is Condition (i) or (iii), then it trivially holds also wrt. $\mathbf{A}$ and $U \cap C$ because these conditions depend only on the assignment $\mathbf{A}$, but not on the unfounded set U.

If it is Condition (ii), then $\mathbf{A} \dot\cup \neg.U \not\models b$ for some (ordinary or external) body literal $b \in B(r)$. We show next that the truth value of all literals in $B(r)$ is the same in $\mathbf{A} \dot\cup \neg.U$ and $\mathbf{A} \dot\cup \neg.(U \cap C)$, which proves that Condition (ii) holds also wrt. $\mathbf{A}$ and $U \cap C$.

If $b = \mathrm{not}\, a$ for some atom a, then $\mathbf{T}a \in \mathbf{A}$ and $a \notin U$ and consequently $a \notin U \cap C$, hence $\mathbf{A} \dot\cup \neg.(U \cap C) \not\models b$. If b is an ordinary atom, then either $\mathbf{F}b \in \mathbf{A}$, which implies immediately that $\mathbf{A} \dot\cup \neg.(U \cap C) \not\models b$, or $b \in U$. But in the latter case b is either in a predecessor component C' of C or in C itself (since $h \rightarrow_p b$ for all $h \in H(r)$). But since $U \cap C' = \emptyset$ for all predecessor components of C, we know $b \in C$ and therefore $b \in (U \cap C)$, which implies $\mathbf{A} \dot\cup \neg.(U \cap C) \not\models b$.

If b is a positive or default-negated external atom, then all input atoms a to b are either in a predecessor component C' of C or in C itself (since $h \rightarrow_p^e a$ for all $h \in H(r)$). We show with a similar argument as before that the truth value of each input atom a is the same wrt. $\mathbf{A} \dot\cup \neg.U$ and $\mathbf{A} \dot\cup \neg.(U \cap C)$: if $\mathbf{A} \dot\cup \neg.U \models a$, then $\mathbf{T}a \in \mathbf{A}$ and $a \notin U$, hence $a \notin (U \cap C)$ and therefore $\mathbf{A} \dot\cup \neg.(U \cap C) \models a$. If $\mathbf{A} \dot\cup \neg.U \not\models a$, then either $\mathbf{F}a \in \mathbf{A}$, which immediately implies $\mathbf{A} \dot\cup \neg.(U \cap C) \not\models a$, or $a \in U$. But in the latter case a must be in C because $U \cap C' = \emptyset$ for all predecessor components C' of C. Therefore $a \in (U \cap C)$ and consequently $\mathbf{A} \dot\cup \neg.(U \cap C) \not\models a$. Because all input atoms a have the same truth value wrt. $\mathbf{A} \dot\cup \neg.U$ and $\mathbf{A} \dot\cup \neg.(U \cap C)$, the same holds also for the positive or default-negated external atom b itself. □

This proposition states that a search for unfounded sets can be done independently for the subprograms Π_C for all $C \in Comp$. If there is an unfounded set of Π wrt. an assignment, then there is also one of at least one program component wrt. this assignment. However, we know by Corollary 3.4 that programs Π without e-cycles cannot contain unfounded sets, which are not

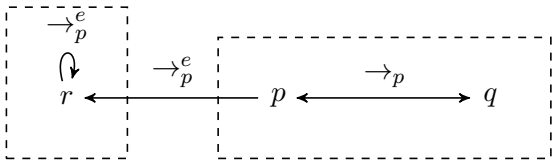


Figure 3.3: Decomposition of the Program from Example 41

already detected when $\hat{\Pi}$ is solved. If we apply this proposition to the subprograms Π_C, we can safely ignore e-cycle-free program components.

Example 41. Reconsider the program Π from Example 38. Then $Comp$ contains the components $C_1 = \{p, q\}$ and $C_2 = \{r\}$ and we have $\Pi_{C_1} = \{p \leftarrow \&id[r](); \ p \leftarrow q; \ q \leftarrow p\}$ and $\Pi_{C_2} = \{r \leftarrow \&id[r]()\}$ (see Figure 3.3). By Proposition 3.21, each unfounded set of Π wrt. some assignment can also detected in one of the components. Consider e.g. $U = \{p, q, r\}$ wrt. $\mathbf{A} = \{\mathbf{T}p, \mathbf{T}q, \mathbf{T}r\}$. Then $U \cap \{r\} = \{r\}$ is also an unfounded set of Π_{C_2} wrt. $\mathbf{A}$.

By separate application of Corollary 3.4 to the components, we can conclude that there can be no unfounded sets over Π_{C_1} that are not already detected when $\hat{\Pi}$ is evaluated (because it has no e-cycles). Hence, the additional unfounded set check is only necessary for Π_{C_2}. Indeed, the only unfounded set which is not detected when $\hat{\Pi}$ is evaluated is $\{r\}$ of Π_{C_2} wrt. any interpretation $\mathbf{A} \supseteq \{\mathbf{T}r\}$. □

Finally, one can also show that splitting, i.e., the component-wise check for foundedness, does not lead to spurious unfounded sets.

Proposition 3.22. *If U is an unfounded set of Π_C wrt. $\mathbf{A}$ such that $U \subseteq C$, then U is an unfounded set of Π wrt. $\mathbf{A}$.*

Proof. If $U = \emptyset$, then the result holds trivially. By definition of Π_C we have $H(r) \cap C = \emptyset$ for all $r \in \Pi \setminus \Pi_C$. By the precondition of the proposition we have $U \subseteq C$. But then $H(r) \cap U = \emptyset$ for all $r \in \Pi \setminus \Pi_C$ and U is an unfounded set of Π wrt. $\mathbf{A}$. □

More generally, unfounded set checks may also be performed over program components larger than single strongly connected components, but we leave this extension for future work.

3.2.7 Minimality Checking Algorithm

We now summarize the results from the previous subsections algorithmically and integrate them into Algorithm GuessAndCheckHexEvaluation. Intuitively, the idea is to construct the nogood set for unfounded set detection (using one of the two encodings) and to enumerate its solutions until either a solution passes the post-check or all solutions have been exhausted. In the former case an unfounded set has been found, in the latter it is proven that there does not exist an unfounded set. The procedure is formalized in Algorithm FLPCheck and summarized in Figure 3.4.

Algorithm FLPCheck

Input: A program Π, a compatible set $\hat{\mathbf{A}}$, a set of nogoods ∇ of Π
Output: *true* if $\mathbf{A}$ is an answer set of Π and *false* otherwise, learned nogoods added to ∇

(a) **for** $C \in Comp$ **do**
(b) **if** *there is an e-cycle of* Π_C *wrt.* $\rightarrow_p^d$ **then**
if *encoding* Γ **then**
SAT instance is $\Gamma_{\Pi_C,\mathbf{A}}$
Let $\mathcal{T}(N)$ be $\mathcal{T}_\Gamma(N)$
if *encoding* Ω **then**
SAT instance is Ω_{Π_C} with assumptions $\mathcal{A}_\mathbf{A}$
Let $\mathcal{T}(N)$ be $\mathcal{T}_\Omega(N)$
(c) **while** *SAT instance has more solutions* **do**
Let $\mathbf{S}$ be the next solution of the SAT instance
Let U be the unfounded set candidate encoded by $\mathbf{S}$
$isUFS \leftarrow true$
(d) **for** *all external atoms* $\&g[\mathbf{y}](\boldsymbol{x})$ *in* Π_C **do**
Evaluate $\&g[\mathbf{y}]$
$\nabla \leftarrow \nabla \cup \Lambda(\&g[\mathbf{y}], \mathbf{A})$
Add $\mathcal{T}(N)$ to the SAT instance for all $N \in \Lambda(\&g[\mathbf{y}], \mathbf{A})$
if $\mathbf{T}e_{\&g[\mathbf{y}]}(\boldsymbol{x}) \in \mathbf{S}$, $\mathbf{A} \not\models \&g[\mathbf{y}](\boldsymbol{x})$ *and* $\mathbf{A} \dot\cup \neg.U \not\models \&g[\mathbf{y}](\boldsymbol{x})$ **then**
$isUFS \leftarrow false$
if $\mathbf{F}e_{\&g[\mathbf{y}]}(\boldsymbol{x}) \in \mathbf{S}$, $\mathbf{A} \models \&g[\mathbf{y}](\boldsymbol{x})$ *and* $\mathbf{A} \dot\cup \neg.U \models \&g[\mathbf{y}](\boldsymbol{x})$ **then**
$isUFS \leftarrow false$
(e) **if** $isUFS$ **then**
Let $N \in L_1(U, \Pi_C, \mathbf{A})$ be a nogood learned from the UFS
$\nabla \leftarrow \nabla \cup \{N\}$
return *false*
return $true$

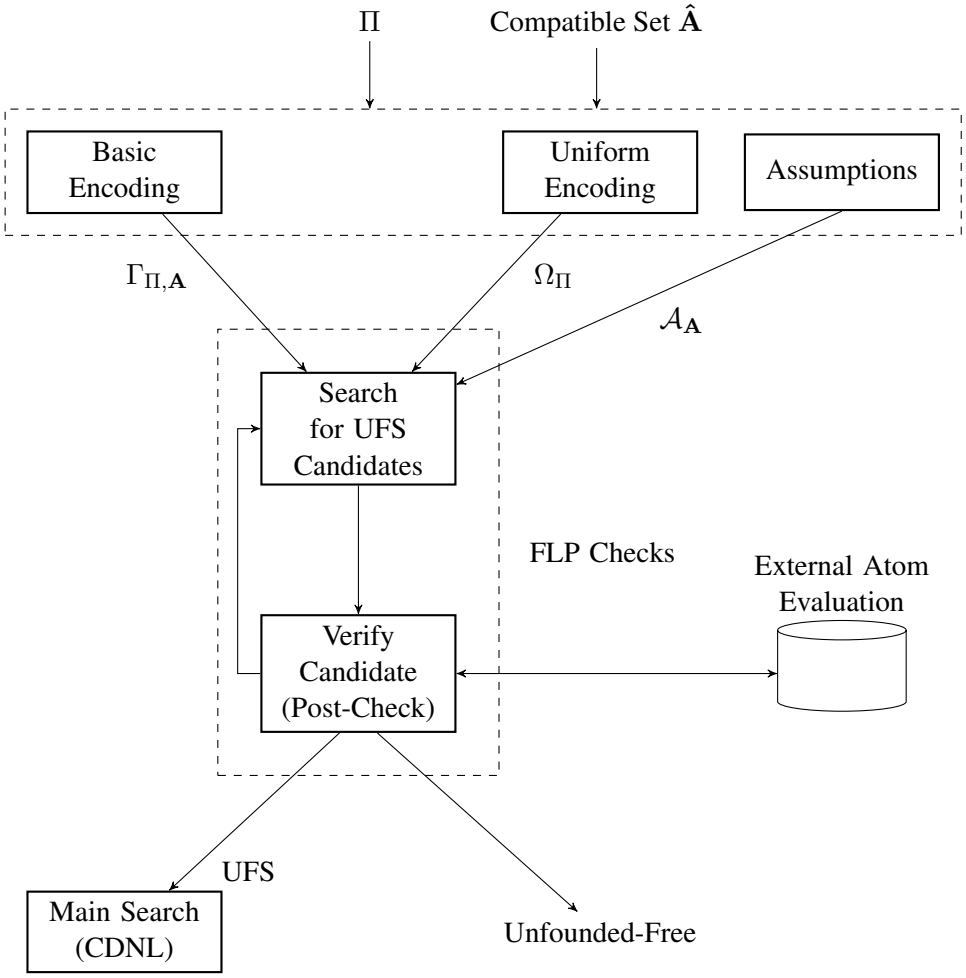


Figure 3.4: Visualization of Algorithm FLPCheck

One can show that it identifies all answer sets among the compatible sets of a program. This property is essential for the correctness and completeness of Algorithm GuessAndCheckHexEvaluation as stated by Theorem 1. Thus, the collection of all algorithms developed in this section is sound and complete for the HEX-semantics for arbitrary ground HEX-programs.

Theorem 3 (Soundness and Completeness of Algorithm FLPCheck)**.** *For every program* Π *(i) Algorithm FLPCheck returns* $true$ *if and only if the restriction* $\mathbf{A}$ *of* $\hat{\mathbf{A}}$ *to atoms* $A(\Pi)$ *is an answer set of* Π*. (ii) All answer sets of* Π *are solutions to all nogoods added to* ∇ *by Algorithm FLPCheck.*

Proof. (i) The loop at (a) enumerates all subset-maximal strongly connected components of the ordinary atoms of Π wrt. $\rightarrow_p \cup \rightarrow_p^e$. By Theorem 2, $\mathbf{A}$ is an answer set of Π if and only if it is unfounded-free. By Propositions 3.21 and 3.22, there is a nonempty unfounded set of some Π_C wrt. $\mathbf{A}$ if and only if there a nonempty unfounded set of Π wrt. $\mathbf{A}$. Thus, the proposition is

proved if we can prove that the algorithm returns $false$ if and only if there is an unfounded set of Π_C wrt. $\mathbf{A}$.

In Part (b), components without e-cycles are skipped. If there is a nonempty unfounded set U of some Π_C wrt. $\mathbf{A}$ s.t. no e-cycle of Π_C wrt. $\rightarrow_p^d$ exists, then by Proposition 3.19 there is also a nonempty unfounded set U' of $\hat{\Pi}_C$ wrt. $\hat{\mathbf{A}} = \kappa(\Pi, \mathbf{A})$. However, U' is then also an unfounded set of $\hat{\Pi}$ wrt. $\hat{\mathbf{A}}$ because $U' \cap H(\hat{r}) = \emptyset$ for all $\hat{r} \in \hat{\Pi} \setminus \hat{\Pi}_C$. Because $\hat{\mathbf{A}}$ is a compatible set of Π by assumption, it is an answer set of $\hat{\Pi}$. Thus $\hat{\mathbf{A}}$ is unfounded-free wrt. $\hat{\Pi}$, i.e., U' cannot exist. Thus, for components without e-cycles, no unfounded set can be detected and they can be ignored.

If there are e-cycles of Π_C wrt. $\rightarrow_p^d$, then we choose an encoding Γ or Ω and setup the according SAT instance. The loop at (c) enumerates all solutions $\mathbf{S}$ to $\Gamma_{\Pi,\mathbf{A}}$, which encode candidate unfounded sets and include all unfounded sets of Π wrt. $\mathbf{A}$ by Proposition 3.6 resp. 3.9. The loop at (d) evaluates all external atoms, and adds the learned valid input-output relationships to the main search (via ∇) and in transformed form to the UFS search. This is conservative by Propositions 3.14 and 3.15. The loop further implements the post-check formalized by Propositions 3.7 and 3.10. Thus, $isUFS$ is set to $true$ and the if at (e) returns $false$ if and only if the post-check is passed for some solution $\mathbf{S}$, i.e., an unfounded set of Π wrt. $\mathbf{A}$ exists; otherwise $true$ is returned after the loop at (c).

(ii) The algorithm adds only a nogood $N \in L_1(U, \Pi_C, \mathbf{A})$ if U is an unfounded set of Π_C wrt. $\mathbf{A}$. The claim follows then from Proposition 3.16. □

3.3 Wellfounded Evaluation Algorithm for Monotonic Ground HEX-Programs

For monotonic programs the evaluation is much simpler, both from a conceptual and from a computational point of view. To this end, we introduce monotonic HEX-programs as follows.

Definition 49. A HEX-program Π is *monotonic*, if

(i) for any $a, b \in A(\Pi)$, $a \rightarrow_n^e b$ or $a \rightarrow_n b$ in the atom dependency graph $ADG(\Pi)$ implies that $b \leftarrow . \in \Pi$, i.e., b is a fact; and

(ii) Π does not contain disjunctions.

The intuition behind this definition is that there is no nondeterminism in the program, once the facts have been fixed.

Example 42. The program $\Pi' = \{s(a);\ s(Y) \leftarrow s(X), \&concat[X, a](Y), limit(Y)\}$ from Example 9 is monotonic. □

The evaluation of such programs is carried out by Algorithm WellfoundedHexEvaluation. This algorithm was sketched by Schindlauer (2006), but not formalized. It starts with an assignment consisting of all $\mathbf{T}a$ for all facts $a \leftarrow$ in the program. Then the assignment is iteratively expanded by adding the (single) head atoms of all rules whose bodies are satisfied by $\mathbf{A}$, if the

unassigned atoms are assumed to be false. As the program is finite, this procedure will eventually reach a fixpoint, i.e., the assignment is not changed anymore, or a constraint fires. In the latter case there does not exist an answer set of Π, otherwise the fixpoint, extended by $\mathbf{F}a$ for all atoms a which were not derived, is the unique answer set.

Algorithm WellfoundedHexEvaluation

Input: A monotonic ground HEX-program Π
Output: All answer sets of Π

$\mathbf{A} \leftarrow \emptyset$
`// iteratively expand the assignment by all atoms derived by rules with satisfied bodies`
while $\mathbf{A}$ *changed* **do**
  $\mathbf{A} \leftarrow \mathbf{A} \cup \left\{\mathbf{T}a \mid r \in \Pi, a \in H(r), \mathbf{A} \cup \{\mathbf{F}a \mid a \in A(\Pi), \mathbf{T}a \notin \mathbf{A}\} \models B(r)\right\}$
  if $\mathbf{A} \cup \{\mathbf{F}a \mid a \in A(\Pi), \mathbf{T}a \notin \mathbf{A}\} \models B(r)$ *for constraint* $r \in \Pi$ **then**
    return $\emptyset$
`// add false literals for all atoms which were not derived`
return $\left\{\mathbf{A} \cup \{\mathbf{F}a \mid a \in A(\Pi), \mathbf{T}a \notin \mathbf{A}\}\right\}$

One can formally show that this algorithm is sound and complete wrt. the HEX-semantics.

Proposition 3.23. *If Algorithm WellfoundedHexEvaluation returns (i) a set containing one interpretation, then it is the unique answer set of* Π*; (ii)* $\emptyset$*, then there is no answer set of* Π*.*

Proof. (i) We first show that, if the algorithm returns a set containing one interpretation, then this interpretation is an answer set. Suppose Algorithm WellfoundedHexEvaluation returns an interpretation $\mathbf{A}$. Then all rules in Π are satisfied, because otherwise for some rule $r \in \Pi$, the body $B(r)$ is satisfied and the head $H(r)$ is not. But this is impossible, because in this case the loop would not have terminated and $a \in H(r)$ would have been added as positive literal to $\mathbf{A}$. Moreover, $\mathbf{A}$ is also subset-minimal wrt. $f\Pi^{\mathbf{A}}$ because for any interpretation $\mathbf{A}' < \mathbf{A}$, at least one rule body $B(r)$ for $r \in \Pi$ is satisfied by $\mathbf{A}'$ but such that the corresponding head $H(r)$ is not. Otherwise the atom $a \in H(r)$ would not have been added to $\mathbf{A}$. As $\mathbf{A}' \models B(r)$ implies $\mathbf{A} \models B(r)$ by monotonicity of our program, this means that we have also $r \in f\Pi^{\mathbf{A}}$. Hence, $\mathbf{A}'$ is not a model of the reduct of Π wrt. $\mathbf{A}$.

Moreover, $\mathbf{A}$ must be contained in any answer set $\mathbf{A}'$ of Π since otherwise some rule would be violated (a rule which derives some atom that is true in $\mathbf{A}$ but false in $\mathbf{A}'$). But then by minimality of answer sets we can conclude that $\mathbf{A}$ is the only answer set.

(ii) Now we show that, if the algorithm returns $\emptyset$, then there does not exist an answer set of Π. Suppose the algorithm returns $\emptyset$. Then for the intermediate result $\mathbf{A}$ of our algorithm just before $\emptyset$ is returned, a constraint in Π is violated. Since Π is monotonic, the only way of satisfying the constraint is to change some signed literals in $\mathbf{A}$ from true to false (note that the constraint cannot contain any $\mathrm{not}\, a$ in its body such that a could become true in a later iteration, since this contradicts monotonicity of the program). But then the resulting assignment $\mathbf{A}'$ keeps at least one rule unsatisfied, because otherwise no a such that $\mathbf{T}a \in \mathbf{A}, \mathbf{F}a \in \mathbf{A}'$ would have

been set to true by our algorithm. Hence, neither $\mathbf{A}$, nor any smaller or incomparable assignment can be an answer set of Π. By monotonicity of the constraint, also no larger assignment can be an answer set, i.e., Π has no answer set. □

Corollary 3.5. *Algorithm WellfoundedHexEvaluation returns all answer sets of a monotonic* HEX*-program* Π.

Proof. If the algorithms returns an interpretation which is not an answer set we have a contradiction with Proposition 3.23 (i). If it misses to return an answer set we have a contradiction with Proposition 3.23 (ii). □

This allows for using Algorithm WellfoundedHexEvaluation in place of EvalGroundHexProgram in EvaluateExtendedPreGroundable in Chapter 2. It is easy to see that this algorithm is polynomial (modulo complexity of external atom evaluations) because the number of iterations is bounded by $A(\Pi)$ and each iteration is bounded by the total number of body atoms in the program (if implemented naively). Thus, the algorithm is usually more efficient than Algorithm GuessAndCheckHexEvaluation.

3.4 Related Work and Summary

The chapter is concluded with a discussion of related work. We then give a summary and an outlook on future work.

3.4.1 Related Work

The basic idea of our conflict-driven algorithm is related to *constraint ASP solving*, which is an extension of ASP programs by constraint atoms (comparisons of terms, e.g. $X \leq 5$), and global constraints such as domain restrictions of constraint variables. Algorithms for constraint ASP solving have been presented by Gebser et al. (2009) and by Ostrowski and Schaub (2012) and are realized in the CLINGCON system. External atom evaluation in our algorithm can superficially be regarded as constraint propagation. However, while both Gebser et al. (2009) and Ostrowski and Schaub (2012) consider a particular application, we deal with a more abstract interface to external sources. An important difference between CLINGCON and external behavior learning (EBL) is that the constraint solver is seen as a black box, whereas we exploit known properties of external sources. Moreover, we support *user-defined learning*, i.e., customization of the default construction of conflict clauses to incorporate knowledge about the sources, as discussed in Section 3.1.1. Another difference is the construction of conflict clauses. Constraint ASP has special constraint atoms, which may be contradictory, e.g. $\mathbf{T}(X > 10)$ and $\mathbf{T}(X = 5)$. The learned clauses are sets of constraint literals, which are kept as small as possible. In our algorithm we have usually no conflicts between ground external atoms as output atoms are mostly independent of each other (excepting e.g. functional sources). Instead, we have a strong relationship between the input and the output. This is reflected by conflict clauses which usually consist of (relevant) input atoms and the negation of one output atom. As in constraint ASP solving, the key for efficiency is keeping conflict clauses small.

Unfounded set checking has been established as a fruitful approach in ASP solving. Foundedness is besides grounding one of the main differences between ASP and SAT solving. Historically, different kinds of unfounded set checks with different complexities have been developed for various program classes. Normal logic programs without external sources require already an unfounded set check which runs, however, in polynomial time and is frequently realized using *source pointers* [Simons et al., 2002]. Intuitively, the reasoner stores for each atom a pointer to a rule which possibly supports this atom. The list of source pointers is updated during propagation. If at some point there is no supporting rule for an atom, then it can be concluded that this atom must be false. The approach has then been extended to disjunctive logic programs. Related to our work is the one of Koch et al. (2003), which reduces stable model checking for disjunctive logic programs to unsatisfiability testing of CNFs, which, like answer set checking from FLP-reducts, is co-NP-complete [Faber et al., 2011]. The approach of Koch et al. (2003) was then extended to conflict-driven learning and unfounded set checking by Drescher et al. (2008). Here, two instances of the reasoner generate and check answer set candidates. As a further extension we considered unfounded set checking for disjunctive logic programs with external atoms. In our setting, we need in addition to respect the semantics of external sources, thus the results there do not carry over immediately. External sources prevent encoding the whole unfounded set search in a SAT instance since their semantics is in general not known a priori. Thus, we developed a SAT encoding combined with a post-check. The technique of Drescher et al. (2008) was recently, in parallel to our work, refined by exploiting assumptions such that the encoding of the unfounded set search does not need to be adapted to the current assignment [Gebser et al., 2013]. This is related to our uniform encoding of the unfounded set search, but still restricted to disjunctive ASP *without* external sources. From a complexity point of view, the difference between ordinary disjunctive programs and FLP programs with external atoms is that co-NP-hardness holds for the latter already for Horn programs with nonmonotonic external atoms that are decidable in polynomial time. For computationally harder external atoms, the complexity might increase relative to an oracle for the external function [Faber et al., 2011]. However, the results from this thesis do still apply in such cases.

Moreover, Drescher et al. (2008) also use a splitting technique related to our program decomposition, yet for ordinary programs only. While we consider e-cycles, which are specific for HEX-programs, the interest of Drescher et al. (2008) is with head-cycles with respect to disjunctive rule heads. In fact, our technique may be regarded as an extension of the work by Drescher et al. (2008), since the evaluation of $\hat{\Pi}$ follows their principles of performing UFS checks in case of head-cycles. Note that our splitting is different from the well-known splitting technique [Lifschitz and Turner, 1994] as we consider only positive dependencies for ordinary atoms.

The unfounded set check presented in this work is needed for the FLP semantics, but other semantics may not need on such a check. For instance, Shen (2011), Shen and Wang (2011) and Shen et al. (2014) present a semantics where unfounded set checking is essentially replaced by a fixpoint iteration which, intuitively, tests if a model candidate reproduces itself. This might be more efficient in some cases.

3.4.2 Summary and Future Work

In this chapter we have first introduced a novel guess and check evaluation algorithm for ground HEX-programs. It is related to conflict-driven disjunctive ASP solving [Drescher et al., 2008], but extends the techniques to programs with external atoms. In contrast to the previous translation approach, the new algorithms respect external atoms as first-class citizens.

Whenever the algorithm calls external sources, which is not only done after a model candidate has been created but possibly also during model construction, it possibly learns additional nogoods from the call. While the algorithm is designed in a generic form which uses learned nogoods abstractly defined by *learning functions*, we have also specified concrete learning functions for external atoms with frequent properties, such as functionality and monotonicity. We have shown that adding these nogoods to the solver is correct in the sense that it does not eliminate compatible sets. Hence, the nogoods help restricting the search space by exploiting part of the known input-output behavior of external atoms.

Answer sets are subset-minimal. Checking the minimality is in general (in presence of disjunctive rule heads and/or nonmonotonic external atoms) a co-NP-complete task and requires special attention. We have designed a minimality check based on the concept of unfounded sets [Faber, 2005]. The search for unfounded sets is realized as a separate search problem which is encoded as SAT instance, for which we discussed two encodings. That is, the solutions to the SAT instance contain representations of all unfounded sets, but not all solutions are such representations. However, the unfounded sets can be identified among all solutions by a relatively simple post-check. We have then shown several optimizations of the basic minimality check and tightly coupled the minimality check and the search for unfounded sets by *nogood exchanging*, i.e., nogoods learned in one search problem can be reused for the other one.

We have shown a decision criterion which allows for skipping the entire minimality check for certain practically relevant program classes. The fundamental idea is exploiting the absence of cycles over external atoms. This criterion can not only be applied on the overall HEX-program, but also on program components wrt. a program decomposition introduced in this chapter.

Finally, we have provided an alternative algorithm for programs (or program components in our model-building framework) which are monotonic. This algorithm has lower computational costs as it is based fixpoint iteration instead of guessing, which runs in polynomial time.

Empirical benchmark results are postponed to Chapter 5. They will show that the new algorithms lead to significantly better runtimes, where the gain is potentially even exponential.

We now discuss some starting points for future work. The identification of further properties for informed learning is an important topic. Another issue is the development of heuristics for several purposes. First, our algorithm can perform unfounded set checks already during the search. Second, we have introduced two encodings for unfounded set checking and observe in Chapter 5 that each of them might be more efficient in some cases. A heuristics for dynamically choosing between the two encodings might be subject to future work. Third, our algorithm evaluates external atoms whenever their input is complete. However, this is only one possible strategy. It is also possible to delay external atom evaluation although the input is already complete, which may be advantageous for external sources with high computational costs. On the other hand, it might also be useful in some cases to evaluate external atoms already with partial input (e.g., for monotonic external atoms), since this could derive further nogoods which can

already falsify the partial assignments. Thus, the development of a heuristics for deciding when to evaluate external atoms is also an interesting point for future work.

Chapter 4

Grounding and Domain Expansion

In this chapter we consider programs with variables and appropriate grounding algorithms, i.e., transformations of programs with variables into propositional programs. While efficient grounding algorithms for ordinary ASP programs already exist, the presence of external atoms calls for new grounding techniques. In particular, *value invention* is a special challenge, i.e., programs with external sources which return constants that do not show up in the original program. While naive support of value invention leads to programs with infinite groundings and answer sets in general, suitable safety conditions can be used to restrict the use of value invention such that this is avoided. Traditionally, the notion of *strong domain-expansion safety* as by Eiter et al. (2006a) (recapitulated in Chapter 2) was used. However, this notion is unnecessarily restrictive because it prevents value invention in many cases although the program *can* be finitely grounded.

After recapitulating the model-building framework for HEX-programs in Section 4.1, relaxing the safety conditions is thus a main concern in this chapter and will be addressed in Section 4.2. This will lead to a new class of programs which is in contrast to strongly domain-expansion safe HEX-programs from Chapter 2 and will be called *liberally domain-expansion safe* HEX-*programs*.

Based on this new class we then develop a new grounding algorithm in Section 4.3. While the traditional grounding algorithm cannot directly process arbitrary strongly domain-expansion safe programs but relies on a decomposition into extended pre-groundable HEX-fragments, as already briefly discussed in Chapter 2, the new algorithm will be able to ground arbitrary liberally and strongly domain-expansion safe HEX-programs as the latter are a strict generalization of liberally domain-expansion safe programs. While, program decomposition is not necessary anymore, it still can be useful in some cases. This gives the designer of evaluation heuristics for the model-building framework more freedom. The new algorithm is integrated into the model-building framework, which is recapitulated in Section 4.1, in Section 4.4.

Finally, we will develop a new evaluation heuristics for the model-building framework which aims at two opposite goals. The program shall be split as rarely as possible because this is

advantageous for the evaluation algorithms from Chapter 3. However, in some cases splits are of great importance for our new grounding algorithm from Section 4.3 for efficiency reasons. Thus, the new heuristics splits the program whenever this is advantageous for grounding purposes (although not necessary), but not more often.

4.1 The Model-Building Framework for HEX-Programs

The evaluation of HEX-programs is traditionally based on a *model-building framework* introduced by Eiter et al. (2011a) and described in more detail by Schüller (2012). The idea is to split the non-ground program into (possibly overlapping) smaller program components, called *evaluation units* or *units* in short, where each evaluation unit is an extended pre-groundable HEX-program as described in Chapter 2. The decomposition is achieved by application of a generalized version of the *Splitting Theorem* [Lifschitz and Turner, 1994] and the *Global Splitting Theorem* [Schindlauer, 2006]. For this purpose, a *dependency graph* between non-ground rules of the program is constructed, which is in contrast to former evaluation techniques that used dependencies between atoms instead of rules [Eiter et al., 2006b; Schindlauer, 2006].

The decomposition of the overall program into evaluation units is done for two reasons. First, this may increase efficiency in some cases, as observed by Schüller (2012). And second, the decomposition is sometimes even *necessary* because the actual evaluation in Algorithm EvaluateExtendedPreGroundable (see Chapter 2) can only handle extended pre-groundable HEX-programs. Thus, if the input program is not extended pre-groundable, the framework must split it such that each unit becomes extended pre-groundable. It was shown by Schüller (2012) that such a splitting exists for every strongly domain-expansion safe program. In later subsections of this chapter, we will develop a more advanced algorithm which can handle a larger class of programs directly. This gives the framework more freedom in the decision whether units are split or not.

The work by Eiter et al. (2011a) and Schüller (2012) focuses on the evaluation framework as a whole and uses black-box ASP solvers for evaluating the single evaluation unit, i.e., for implementing Algorithm EvaluateExtendedPreGroundable. In this sense, a macroscopic perspective is chosen. In contrast, this thesis puts the focus on the evaluation of the units and therefore has a microscopic point of view. Nevertheless we describe the main aspects of the evaluation framework to provide a complete picture of the evaluation methods.

Our running example in this subsection will be the following.

Example 43. Let Π be the following ground program with facts $employee(a)$, $employee(b)$, $employee(c)$ and $qualification(c)$:

$$
\begin{aligned}
r_1\colon&\quad team1(a) \vee team1(b) \leftarrow\\
r_2\colon&\quad team1(b) \vee team1(c) \leftarrow\\
r_3\colon&\quad team2(X) \leftarrow \&diff[employee, team1](X)\\
r_4\colon&\quad team1a(X) \leftarrow \&diff[team1, qualification](X)\\
r_5\colon&\quad team1b(X) \leftarrow team1(X), qualification(X)\\
r_6\colon&\quad bonus(X) \leftarrow team2(X)\\
r_7\colon&\quad bonus(X) \leftarrow team1b(X)
\end{aligned}
$$

Intuitively, the program considers a company with employees defined using predicate $employee$, some of which have a certain $qualification$. The program forms then two teams $team1$ and $team2$ such that certain restrictions concerning the assignment of employees to $team1$, encoded by r_1 and r_2, are satisfied. By r_3, everyone who is not in $team1$ shall be in $team2$. Then $team1$ is further divided into two sub-teams $team1a$ and $team1b$, where $team1b$ shall consist of all employees who have the $qualification$ (r_5); the others are assigned to $team1a$ (r_4). Finally, all employees working in $team2$ or in $team1b$ shall be eligible for a $bonus$ (r_6 and r_7). □

4.1.1 Formalization of the Model-Building Framework

The model-building framework is based on a notion of rule dependencies, which is in contrast to the previous approach [Schindlauer, 2006] based on the atom dependency graph $ADG(\Pi)$. This has the advantage that many theorems by Schüller (2012) become simpler. Moreover, the new framework is also more flexible because it abstractly uses *evaluation graphs* which define the ordering of program evaluation, while the former approach hard-coded this in the algorithms.

Definition 50 (Rule Dependencies)**.** Let Π be a program with rules $r, s \in \Pi$. We denote by $r \rightarrow_m s$ (resp. $r \rightarrow_n s$) that r *depends monotonically* (resp. *depends nonmonotonically*) on s. We define:

(i) If $a \in B^+(r), b \in H(s)$ and $a \sim b$, then $r \rightarrow_m s$.

(ii) If $a \in B^-(r), b \in H(s)$ and $a \sim b$, then $r \rightarrow_n s$.

(iii) If $a \in H(r), b \in H(s)$ and $a \sim b$, then both $r \rightarrow_m s$ and $s \rightarrow_m r$.

(iv) If $a \in B(r)$ is an external atom of form $\&g[\mathbf{Y}](\mathbf{X})$ where $\mathbf{Y} = Y_1, \ldots, Y_n$, the input $Y_i = p$ for $1 \leq i \leq ar_{\text{I}}(\&g)$ has $type(\&g, i) = \mathbf{pred}$, and $b \in H(s)$ is an atom of form $p(\mathbf{Z})$, then

- $r \rightarrow_m s$ if $\&g$ is monotonic (in all predicate parameters) and $a \in B^+(r)$; and
- $r \rightarrow_n s$ otherwise.

Note that the dependency in Condition (iv) is considered monotonic only if the external atom is monotonic in *all* parameters. This is because the former formalization (and implementation) of HEX did not distinguish between different parameters, i.e., there was only a global monotonicity attribute of each external predicate. In this thesis we have a more fine-grained approach and use a separate monotonicity attribute for each predicate parameter, which allows for using a more liberal definition of rule dependencies (monotonic dependencies are usually advantageous over nonmonotonic ones). However, as this thesis does not focus on the evaluation framework, we stick with the former definition at this point and leave the formalization of the relaxed notion for future work.

Example 44 (ctd.)**.** For the program from Example 43 we have the following dependencies, which are visualized in Figure 4.1:

- $r_1 \rightarrow_m r_2$ and $r_2 \rightarrow_m r_1$ by (iii)

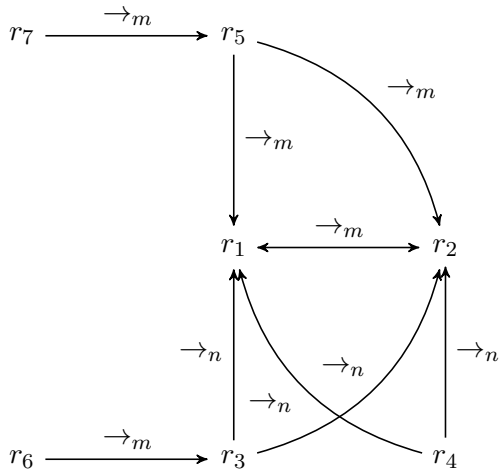


Figure 4.1: Rule Dependencies of the Program from Examples 43 and 44

- $r_3 \rightarrow_n r_1$ and $r_3 \rightarrow_n r_2$ by (iv)
- $r_4 \rightarrow_n r_1$ and $r_4 \rightarrow_n r_2$ by (iv)
- $r_5 \rightarrow_m r_1$ and $r_5 \rightarrow_m r_2$ by (i)
- $r_6 \rightarrow_m r_3$ by (i)
- $r_7 \rightarrow_m r_5$ by (i)

□

The dependency graph is used to construct the *evaluation graph* which controls the overall evaluation of the program. The evaluation graph is composed of extended pre-groundable HEX-programs as nodes, which are called *evaluation units* in this context. The edges of the evaluation graph connect the evaluation units acyclically and are derived from the dependency relation between rules. More formally, we introduce the following concepts [Schüller, 2012].

Definition 51 ((Evaluation) Unit)**.** An *(evaluation) unit* is an extended pre-groundable HEX-program.

Definition 52 (Evaluation Graph)**.** An *evaluation graph* $\mathcal{E} = \langle V, E \rangle$ of a program Π is a directed acyclic graph; vertices V are evaluation units and $\mathcal{E}$ has the following properties:

(a) $\bigcup_{u \in V} u = \Pi$, i.e., every rule $r \in \Pi$ is contained in at least one unit;

(b) for every non-constraint $r \in \Pi$, it holds that $\big|\{u \in V \mid r \in u\}\big| = 1$, i.e., r is contained in exactly one unit;

(c) for each nonmonotonic dependency $r \rightarrow_n s$ between rules $r, s \in \Pi$ and for all $u \in V$ with $r \in u$ and $v \in V$ with $s \in v$ s.t. $u \neq v$, there exists an edge $(u, v) \in E$, i.e., nonmonotonic dependencies between rules have corresponding edges everywhere in $\mathcal{E}$; and

(d) for each monotonic dependency $r \rightarrow_m s$ between rules $r, s \in \Pi$, there exists one $u \in V$ with $r \in u$ such that E contains all edges (u, v) with $v \in V$, $s \in v$ and $v \neq u$, i.e., there is (at least) one unit in $\mathcal{E}$ where all monotonic dependencies from r to other rules have corresponding outgoing edges in $\mathcal{E}$.

We denote by $preds_{\mathcal{E}}(u)$ the predecessors of unit u in $\mathcal{E} = \langle V, E \rangle$, i.e., $preds_{\mathcal{E}}(u) = \{v \in V \mid (u, v) \in E\}$. For units u, w we write $u < w$ if there exists a path from u to w in $\mathcal{E}$ and $u \leq w$ if $u < w$ or $u = w$. Moreover, for a unit $u \in V$ let $u^{<} = \bigcup_{w \in U, u < w} w$ and $u^{\leq} = u^{<} \cup \{u\}$.

Informally, the edges of $\mathcal{E}$ cover the rule dependencies in the sense that if $r \in v$ depends on $s \in w$ with $w \neq v \in V$, then there must be an edge from v to w in E. For the sake of simplicity of the formal results, it is advantageous to introduce an empty *final evaluation unit* u_{final} which depends on all other units, as shown in the following example.

Example 45 (ctd.)**.** A valid evaluation graph for the program in Example 43 is $\mathcal{E} = \langle V, E \rangle$ with

$$V = \{u_1 = \{r_1, r_2\}, u_2 = \{r_3\}, u_3 = \{r_4\}, u_4 = \{r_5\}, u_5 = \{r_6, r_7\}, u_{final} = \emptyset\}$$
$$E = \{(u_2, u_1), (u_3, u_1), (u_4, u_1), (u_5, u_2), (u_5, u_4),$$
$$(u_{final}, u_1), (u_{final}, u_2), (u_{final}, u_3), (u_{final}, u_4), (u_{final}, u_5)\}$$

as visualized in Figure 4.2 (where u_{final} is omitted). □

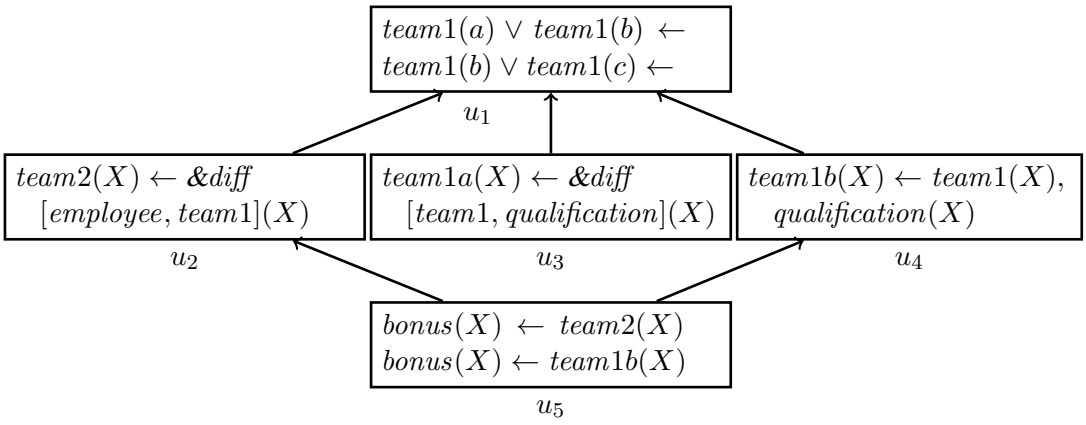


Figure 4.2: Evaluation Graph of Example 45 without u_{final}

Note that there exist in general multiple valid evaluation graphs for a given program. However, it was shown in Proposition 16 by Schüller (2012) that every strongly domain-expansion safe HEX-program has at least one evaluation graph, which is crucial for the applicability of the framework in the general setting. The construction of a concrete evaluation graph is the job of so-called *evaluation heuristics* which can be plugged into the framework. The heuristics may have significant influence on efficiency [Eiter et al., 2011a] and we will develop a new heuristics in Section 4.5.

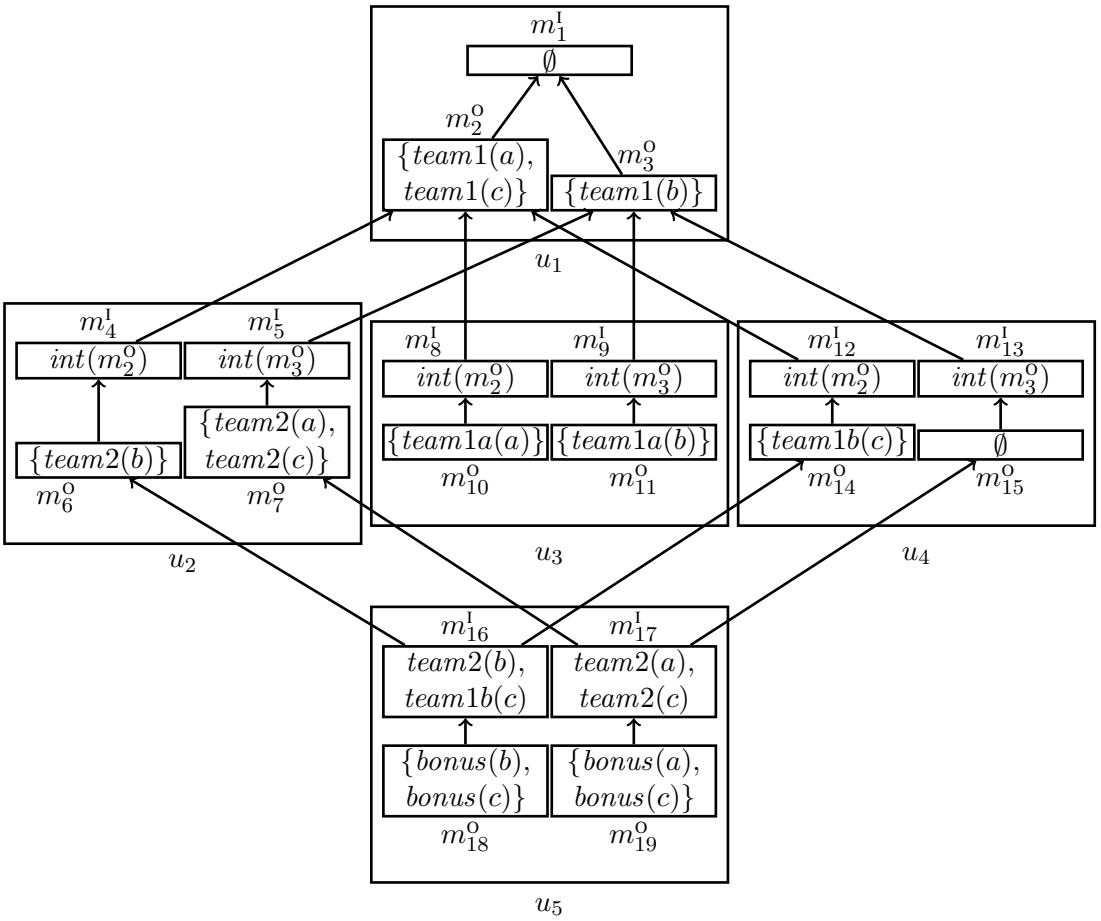


Figure 4.3: Answer Set Graph of Example 48

An important concept for the model building process is that of *first ancestor intersection units*. This will allow us in the following to decide whether the output models of multiple predecessor units origin from the same ancestor.

Definition 53 (First Ancestor Intersection Unit (FAI))**.** For an evaluation graph $\mathcal{E} = \langle V, E \rangle$ and distinct units $v, w \in V$, we say that w *is a first ancestor intersection unit (FAI) of* v if there exist paths $p_1 \neq p_2$ from v to w in E such that p_1 and p_2 share no nodes apart from v and w. We denote by $fai(v)$ the set of all FAIs of a unit v.

Example 46 (ctd.)**.** In the evaluation graph from Example 44, u_1 is the only FAI of u_5 because there exist two paths u_5, u_2, u_1 and u_5, u_4, u_1 from u_5 to u_1, but no other. □

Evaluation is then based on an *answer set graph*, which interrelates models at evaluation units in the evaluation graph. Each unit u has assigned a set of *input models* $i\text{-}ints(u)$ and a set of *output models* $o\text{-}ints(u)$.

Definition 54 (Interpretation Structure)**.** An *interpretation structure* for an evaluation graph $\mathcal{E} = \langle V, E\rangle$ is a labeled directed acyclic graph $\mathcal{I} = \langle M, F, unit, type, int\rangle$ where each node $M \subseteq \mathcal{I}_{id}$ is from a countable set $\mathcal{I}_{id}$ of identifiers, e.g., from $\mathbb{N}$, and $unit\colon\ M \to V$, $type\colon\ M \to \{\text{I}, \text{O}\}$ and $int\colon\ M \to 2^{HB_\Pi}$ are total node labeling functions.

Given a unit $u \in V$ of an evaluation graph $\mathcal{E} = \langle V, E\rangle$, we denote for an interpretation structure $\mathcal{I}$ by $i\text{-}ints_\mathcal{I}(u) = \{m \in M \mid unit(m) = u \text{ and } type(m) = \text{I}\}$ the *input interpretations*, and by $o\text{-}ints_\mathcal{I}(u) = \{m \in M \mid unit(m) = u \text{ and } type(m) = \text{O}\}$ the *output interpretations* at unit u, respectively. Given vertex $m \in M$, we further denote by

$$int^+(m) = int(m) \cup \bigcup \{int(m') \mid m' \in M \text{ and } m' \text{ is reachable from } m \text{ in } \mathcal{I}\}$$

the *expanded interpretation* of m.

An interpretation structure is called *interpretation graph*, if the edge relation satisfies some further properties [Schüller, 2012].

Definition 55 (Interpretation Graph)**.** An *interpretation graph* $\mathcal{I} = \{M, F, unit, type, int\}$ for an evaluation graph $\mathcal{E} = \langle V, E\rangle$ is an interpretation structure which fulfills for every $u \in V$ the following properties:

(IG-I) *I-connectedness:* for every $m \in o\text{-}ints_\mathcal{I}(u)$ the structure contains exactly one outgoing edge $(m, m') \in F$ and $m' \in i\text{-}ints_\mathcal{I}(u)$ is an i-interpretation at unit u;

(IG-O) *O-connectedness:* for every $m \in i\text{-}ints_\mathcal{I}(u)$ and for every predecessor unit $u_i \in preds_\mathcal{E}(u)$ of u, there is exactly one outgoing edge $(m, m_i) \in F$ and $m_i \in o\text{-}ints_\mathcal{I}(u_i)$ (every m_i is an o-interpretation at the respective unit u_i);

(IG-F) *FAI intersection*[1]*:* for every $m \in i\text{-}ints_\mathcal{I}(u)$, let $\mathcal{I}'$ be the subgraph of $\mathcal{I}$ reachable from m, and let $\mathcal{E}'$ be the subgraph of $\mathcal{E}$ reachable from u. Then $\mathcal{I}'$ contains exactly one o-interpretation at each evaluation unit of $\mathcal{E}'$; and

(IG-U) *Uniqueness:* for each pair of distinct vertices $m_1, m_2 \in M, m_1 \neq m_2$ with $unit(m_1) = unit(m_2) = u$ the expanded interpretation of m_1 and m_2 differs, formally $int^+(m_1) \neq int^+(m_2)$.

Example 47 (ctd.)**.** Figure 4.3 shows an interpretation graph for the evaluation graph of Example 44. Actually it shows an answer set graph, which is a special interpretation graph and is introduced next. □

It is intended that an output model m^O of a unit u results from the corresponding input model m^I, if Algorithm EvaluateExtendedPreGroundable is called for the program u augmented by m^I interpreted as facts, and corresponds to an answer set of $u^{\leq}$. However, the definition

[1]This property is called *FAI intersection* because it implies that for any units u and $v \in fai(u)$, all paths in the interpretation graph from some $m \in i\text{-}ints_\mathcal{I}(u)$ to an output model $m' \in o\text{-}ints_\mathcal{I}(v)$ share the same m', i.e., the paths 'intersect' at FAI units. In fact, in order to check the property for a unit u, it suffices to consider all $v \in fai(u)$ because other ancestor units between u and v are not reachable via multiple paths, thus the property cannot be violated (cf. the proof of Proposition 17 by Schüller (2012)).

of an interpretation graph does not refer to the HEX-programs in the evaluation units. Thus, it is not yet guaranteed that the output interpretations of the final evaluation unit are really the intended answer sets of the program. This requires the further notion of an *answer set graph* [Schüller, 2012].

Definition 56 (Answer Set Graph)**.** Given an evaluation graph $\mathcal{E} = \langle V, E \rangle$, an *answer set graph* is an interpretation graph $\mathcal{I}$ for $\mathcal{E}$ such that for each unit $u \in V$ it holds that

(i) every expanded input interpretation in $i\text{-}ints_{\mathcal{I}}(u)$ is an answer set of $u^{<}$, i.e., $int^{+}(m) \in \mathcal{AS}(u^{<})$ for all $m \in i\text{-}ints_{\mathcal{I}}(u)$;

(ii) every expanded output interpretation in $i\text{-}ints_{\mathcal{I}}(u)$ is an answer set of $u^{\leq}$, i.e., $int^{+}(m) \in \mathcal{AS}(u^{\leq})$ for all $m \in o\text{-}ints_{\mathcal{I}}(u)$; and

(iii) every input interpretation at u is the union of the output interpretations it depends on, i.e., $int(m) = \bigcup_{(m,m_i)\in F} int(m_i)$.

4.1.2 Using the Framework for Model Building

For the description of the evaluation algorithm we need to introduce the concept of *joins*. This will allow us to decide which combinations of output models of predecessor units serve as an input model to a successor unit.

Definition 57 (Join)**.** Let $\mathcal{I} = \langle M, F, unit, type, int \rangle$ be an interpretation graph for an evaluation graph $\mathcal{E} = \langle V, E \rangle$. Let $u \in V$ be an evaluation unit and $u_1, \ldots, u_k$ be all units on which u depends. Let $m_i \in o\text{-}ints(u_i)$ for $1 \leq i \leq k$ be output models of predecessor units of u.

Then the *join* $m_1 \bowtie \cdots \bowtie m_k = \bigcup_{1\leq i\leq k} m_i$ is defined if for each $u' \in fai(u)$ there exists exactly one model $m' \in o\text{-}ints(u')$ reachable from some model m_i, $1 \leq i \leq k$, and undefined otherwise.

Intuitively, the concept ensures that only those combinations of output models form an input model to an evaluation unit, which result from one common ancestor model in the model graph.

During program evaluation, Algorithm BuildAnswerSets starts from an empty answer set graph and expands it to the final answer set graph as follows. The algorithm iteratively selects an evaluation unit u such that all direct predecessors $u_1, \ldots, u_k$ have already been processed. Then the algorithm computes in the first step all input interpretations in Parts (a) and (b) (which is the empty interpretation for units without predecessors) and in a second step all output interpretations of u in Part (d). Both steps can be described in terms of updates to the answer set graph. The input interpretations of u_{final} correspond then to the answer sets of Π (cf. Part (c)), which is formalized by the following theorem.

Theorem 4 (Theorem 15 by Schüller (2012))**.** *Given an evaluation graph $\mathcal{E} = (V, E)$ of a program Π, BuildAnswerSets returns $\mathcal{AS}(\Pi)$.*

This is demonstrated with a final example.

Algorithm BuildAnswerSets

Input: Evaluation graph $\mathcal{E} = (V, E)$ for a HEX-program Π with a unit u_{final} that depends on all other units in V

Output: All answer sets of Π

$M \leftarrow \emptyset, F \leftarrow \emptyset, unit \leftarrow \emptyset, type \leftarrow \emptyset, int \leftarrow \emptyset, U \leftarrow V$
while $U \neq \emptyset$ **do**
  Choose $u \in U$ s.t. $preds_{\mathcal{E}}(u) \cap U = \emptyset$
  Let $\{u_1, \ldots, u_k\} = preds_{\mathcal{E}}(u)$
(a) **if** $k = 0$ **then**
    $m \leftarrow max(M) + 1$
    $M \leftarrow M \cup \{m\}$
    $unit(m) \leftarrow u, type(m) \leftarrow \text{I}, int(m) \leftarrow \emptyset$
(b) **else**
    for $m_1 \in o\text{-}ints(u_1), \ldots, m_k \in o\text{-}ints(u_k)$ **do**
      if $J = m_1 \bowtie \cdots \bowtie m_k$ *is defined* **then**
        $m \leftarrow max(M) + 1$
        $M \leftarrow M \cup \{m\}$
        $F \leftarrow F \cup \big\{(m, m_i) \mid 1 \leq i \leq k\big\}$
        $unit(m) \leftarrow u, type(m) \leftarrow \text{I}, int(m) \leftarrow J$
(c) **if** $u = u_{final}$ **then**
    return $i\text{-}ints(u_{final})$
(d) **for** $m' \in i\text{-}ints(u)$ **do**
    $O \leftarrow \text{EvaluateExtendedPreGroundable}\big(u, int(m')\big)$
    for $o \in O$ **do**
      $m \leftarrow max(M) + 1$
      $M \leftarrow M \cup \{m\}$
      $F \leftarrow F \cup \big\{(m, m') \mid 1 \leq i \leq k\big\}$
      $unit(m) \leftarrow u, type(m) \leftarrow \text{O}, int(m) \leftarrow o$
  $U \leftarrow U \setminus \{u\}$

Example 48 (ctd.)**.** We now describe the construction of the answer set graph as depicted in Figure 4.3 (without final unit u_{final}). The evaluation graph in Example 44 has a single unit u_1 without predecessors. Its only input model is the empty one, i.e., $i\text{-}ints(u_1) = \{m_1^{\mathrm{I}} = \emptyset\}$.

The algorithm chooses u_1 and computes the set of output models for input model $\emptyset$, which is $o\text{-}ints(u_1) = \{m_2^{\mathrm{O}} = \{team1(a), team1(c)\}, m_3^{\mathrm{O}} = \{team1(b)\}\}$.

In the next step, one of the components u_2, u_3 or u_4 can be chosen for evaluation because for each of them the single predecessor unit u_1 has already been processed. For u_2 and input model $m_4^{\mathrm{I}} = m_2^{\mathrm{O}} = \{team1(a), team1(c)\}$, the unique output model is $m_6^{\mathrm{O}} = \{team2(b)\}$, and for input model $m_5^{\mathrm{I}} = m_3^{\mathrm{O}} = \{team1(b)\}$, the unique output model is $m_7^{\mathrm{O}} = \{team2(a), team2(c)\}$. For u_3 and input model $m_8^{\mathrm{I}} = m_2^{\mathrm{O}} = \{team1(a), team1(c)\}$, the unique output model is $m_{10}^{\mathrm{O}} = \{team1a(a)\}$, and for input model $m_9^{\mathrm{I}} = m_3^{\mathrm{O}} = \{team1(b)\}$, the unique output model is $m_{11}^{\mathrm{O}} = \{team1a(b)\}$. For u_4 and input model $m_{12}^{\mathrm{I}} = m_2^{\mathrm{O}} = \{team1(a), team1(c)\}$, the unique output model is $m_{14}^{\mathrm{O}} = \{team1b(c)\}$, and for input model $m_{13}^{\mathrm{I}} = m_3^{\mathrm{O}} = \{team1(b)\}$, the unique output model is $m_{15}^{\mathrm{O}} = \emptyset$.

Then the algorithm chooses u_5 for evaluation. The first step is the computation of the input models of u_5. Because u_5 has two predecessor units u_2 and u_4 and each of them has two output models m_6^{O}, m_7^{O} resp. m_{14}^{O}, m_{15}^{O}, there are four possible combinations. However, only the joins $m_{16}^{\mathrm{I}} = m_6^{\mathrm{O}} \bowtie m_{14}^{\mathrm{O}} = \{team2(b), team1b(c)\}$ and $m_{17}^{\mathrm{I}} = m_7^{\mathrm{O}} \bowtie m_{15}^{\mathrm{O}} = \{team2(a), team2(c)\}$ are defined, because for the common ancestor unit u_1 of u_5, there is exactly one output model $m_2^{\mathrm{O}} \in o\text{-}ints(u_1)$ reachable from $m_6^{\mathrm{O}}, m_{14}^{\mathrm{O}}$ resp. $m_3^{\mathrm{O}} \in o\text{-}ints(u_1)$ from $m_7^{\mathrm{O}}, m_{15}^{\mathrm{O}}$. In contrast, from $m_6^{\mathrm{O}}, m_{15}^{\mathrm{O}}$ and $m_7^{\mathrm{O}}, m_{14}^{\mathrm{O}}$, both output models $m_2^{\mathrm{O}}, m_3^{\mathrm{O}}$ of u_1 are reachable. In the second step, the output models of u_5 are determined: for m_{16}^{I} the unique output model is $m_{18}^{\mathrm{O}} = \{bonus(b), bonus(c)\}$ and for m_{17}^{I} the unique output model is $m_{19}^{\mathrm{O}} = \{bonus(a), bonus(c)\}$.

Finally, unit u_{final} is chosen for evaluation. Actually, as this is the final unit and contains no rules, only the input models need to be determined. We have 5 units with 2 output models each, thus we have 2^5 possible combinations. However, only $m_{20}^{\mathrm{I}} = m_2^{\mathrm{O}} \bowtie m_6^{\mathrm{O}} \bowtie m_{10}^{\mathrm{O}} \bowtie m_{14}^{\mathrm{O}} \bowtie m_{18}^{\mathrm{O}} = \{team_1(a), team_1(c), team_{1a}(a), team_{1b}(c), team_2(b), bonus(b), bonus(c)\}$ (with the single reachable model m_2^{O} at the common ancestor unit u_1) and $m_{21}^{\mathrm{I}} = m_3^{\mathrm{O}} \bowtie m_7^{\mathrm{O}} \bowtie m_{11}^{\mathrm{O}} \bowtie m_{15}^{\mathrm{O}} \bowtie m_{19}^{\mathrm{O}} = \{team_1(b), team_{1a}(b), team_2(a), team_2(c), bonus(a), bonus(c)\}$ (with the single reachable model m_3^{O} at the common ancestor unit u_1) are defined. These models are the answer sets of the program. □

4.2 Liberal Safety Criteria for HEX-Programs

In this section we want to relax the traditional safety criteria as formalized by strong domain-expansion safety in Definition 23. We start with an example to demonstrate that some programs which are not strongly domain-expansion safe are still finitely groundable, i.e., a finite subset of the grounding suffices to compute the answer sets.

Example 49. Consider the program Π:

$$\Pi = \begin{Bmatrix} r_1 \colon t(a); & r_3 \colon s(Y) \leftarrow t(X), \&concat[X, a](Y) \\ r_2 \colon dom(aa); & r_4 \colon t(X) \leftarrow s(X), dom(X) \end{Bmatrix}$$

where $\&concat[X, a](Y)$ returns in Y the string in X with a appended, has an infinite grounding. Note that this program is not strongly domain-expansion safe due to the external atom in rule r_3. However, only rules using a and aa are relevant for program evaluation because the cycle is 'broken' by $dom(X)$ in r_4. □

Note that external sources are largely black boxes to the reasoner. Thus the set of relevant constants for grounding might be *intuitively* clear, but not *formally*. Predetermining is in general not possible. We call a program *finitely restrictable* if a finite portion of the grounding of the program is sufficient to preserve all answer sets, i.e., a finite grounding has the same answer sets (wrt. the true atoms) as the complete grounding. This is ensured by additional safety criteria. As the example demonstrates, strong domain-expansion safety as by Definition 23 is unnecessarily restrictive.

Our overall objective in this section is thus to introduce a more liberal notion of safety that still ensures finite restrictability. However, rather than to merely generalize an existing notion, we aim for a generic notion at a conceptual level that may incorporate besides syntactic also semantic information about sources. We will introduce a new notion of *liberal domain-expansion safety* which incorporates both syntactic and semantic properties of the program at hand. In the following, *domain-expansion safety* refers to liberal domain-expansion safety, unless we explicitly say strong domain-expansion safety. Compared to the latter, this gives us a larger class of programs which are guaranteed to have a finite grounding that preserves all answer sets. Unlike strong domain-expansion safety, liberal domain-expansion safety is not a property of entire atoms but of *attributes*, i.e., pairs of predicates and argument positions. Intuitively, an attribute is domain-expansion safe, if the number of different terms in an answer-set preserving grounding (i.e., in a grounding which has the same answer sets if restricted to the positive atoms as the original program) is finite. A program is domain-expansion safe, if all its attributes are domain-expansion safe.

4.2.1 Liberally Domain-Expansion Safe HEX-Programs

Our notion of *liberal domain-expansion safety (de-safety)* is designed in an extensible fashion, i.e., such that several safety criteria can be easily integrated. For this we parameterize our definition of domain-expansion safety by a *term bounding function (TBF)*, which identifies variables in a rule that are ensured to have only finitely many instantiations in the answer set preserving grounding. Finiteness of the overall grounding follows then from the properties of TBFs. Concrete syntactic and semantic properties are realized in our definitions of concrete TBFs (cf. Section 4.2.2).

Definition 58 (Attributes)**.** For an ordinary predicate $p \in \mathcal{P}$, let $p\restriction i$ be the *i-th attribute of p* for all $1 \leq i \leq ar(p)$. For an external predicate $\&g \in \mathcal{X}$ with input list $\mathbf{Y}$ in rule r, let $\&g[\mathbf{Y}]_r\restriction_T i$ with $T \in \{\text{I}, \text{O}\}$ be the *i-th input resp. output attribute of $\&g[\mathbf{Y}]$ in r* for all $1 \leq i \leq ar_T(\&g)$.

For a program Π, the *range* of an attribute is, intuitively, the set of ground terms which occur in the position of the attribute. Formally:

Definition 59 (Range)**.** For an attribute $p{\restriction}i$ we define $range(p{\restriction}i, \Pi) = \{t_i \mid p(t_1, \ldots, t_{ar(p)}) \in A(\Pi)\}$; for an attribute $\&g[\mathbf{Y}]_r{\restriction}_T i$ we define $range(\&g[\mathbf{Y}]_r{\restriction}_T i, \Pi) = \{x_i^T \mid \&g[\mathbf{x}^{\mathrm{I}}](\mathbf{x}^{\mathrm{O}}) \in EA(\Pi)\}$, where $\mathbf{x}^T = x_1^T, \ldots, x_{ar_T(\&g)}^T$.

Example 50. Some attributes of the program Π from Example 49 are $t{\restriction}1$, $\&concat[X, a]_{r_3}{\restriction}_{\mathrm{I}}2$ and $\&concat[X, a]_{r_3}{\restriction}_{\mathrm{O}}1$. We further have $range(t{\restriction}1, \Pi) = \{a\}$. □

Definition 60 (Grounding Operator G_Π)**.** We use the following monotone operator to compute by fixpoint iteration a finite subset of $grnd_{\mathcal{C}}(\Pi)$ for a program Π:

$$G_\Pi(\Pi') = \bigcup_{r \in \Pi} \{r\theta \mid \exists \mathbf{A} \subseteq \mathcal{A}(\Pi'), \mathbf{A} \not\models \bot, \mathbf{A} \models B^+(r\theta)\},$$

where $\mathcal{A}(\Pi') = \{\mathbf{T}a, \mathbf{F}a \mid a \in A(\Pi')\} \setminus \{\mathbf{F}a \mid a \leftarrow . \in \Pi\}$ and $r\theta$ is the instance of r under variable substitution $\theta\colon\ \mathcal{V} \to \mathcal{C}$.

Note that in this definition, $\mathbf{A}$ might be partial, but by convention we assume that all atoms which are not explicitly assigned to true are false. Moreover, ranges are defined also for non-ground programs.

That is, G_Π takes a ground program Π' as input and returns all rules from $grnd_{\mathcal{C}}(\Pi)$ whose positive body is satisfied by some assignment over the atoms of Π'. Intuitively, the operator iteratively extends the grounding by new rules if they are possibly relevant for the evaluation, where relevance is in terms of satisfaction of the positive rule body by some assignment constructible over the atoms which are possibly derivable so far.

Obviously, the least fixpoint $G_\Pi^\infty(\emptyset)$ of this operator is a subset of $grnd_{\mathcal{C}}(\Pi)$; we will show that it is finite if Π is domain-expansion safe according to our new notion. Moreover, we will show that this grounding preserves all answer sets because all rule instances which are not added have unsatisfied bodies anyway.

Example 51. Consider the following program Π:

$$r_1\colon s(a); \quad r_2\colon dom(ax); \quad r_3\colon dom(axx)$$
$$r_4\colon s(Y) \leftarrow s(X), \&concat[X, x](Y), dom(Y)$$

The least fixpoint of G_Π is the following ground program:

$$r_1'\colon\ s(a); \quad r_2'\colon\ dom(ax); \quad r_3'\colon\ dom(axx)$$
$$r_4'\colon\ s(ax) \leftarrow s(a), \&concat[a, x](ax), dom(ax)$$
$$r_5'\colon\ s(axx) \leftarrow s(ax), \&concat[ax, x](axx), dom(axx)$$

Rule r_4' is added in the first iteration and rule r_5' in the second. □

Towards a formal definition of domain-expansion safety, we introduce the following notions.

Definition 61 (Bounded Terms)**.** A term in a rule is *bounded*, if the number of substitutions in $G_\Pi^\infty(\emptyset)$ for this term is finite.

Boundedness of terms is abstractly formalized using *term bounding functions (TBFs)*. That is, a TBF declares terms as bounded.

Definition 62 (Term Bounding Function (TBF))**.** A *term bounding function* $b(\Pi, r, S, B)$ maps a program Π, a rule $r \in \Pi$, a set S of domain-expansion safe attributes, and a set B of bounded terms in r to an enlarged set of bounded terms $b(\Pi, r, S, B) \supseteq B$, such that all $t \in b(\Pi, r, S, B)$ have finitely many substitutions in $G^{\infty}_{\Pi}(\emptyset)$ if

(i) the attributes S have a finite range in $G^{\infty}_{\Pi}(\emptyset)$; and

(ii) each term in $terms(r) \cap B$ has finitely many substitutions in $G^{\infty}_{\Pi}(\emptyset)$.

Intuitively, a TBF gets a set of already bounded terms and a set of already domain-expansion safe attributes. The TBF then derives under these preconditions further terms which are also bounded.

Our concept derives domain-expansion safety of attributes and programs from the boundedness of variables according to a TBF. We use a mutually inductive definition that starts from the empty set of domain-expansion safe attributes $S_0(\Pi)$ and then derives in each step $n \geq 1$, first the set $B_n(r, \Pi, b)$ of bounded terms for all rules r, and then an enlarged set $S_n(\Pi)$ of domain-expansion safe attributes. The set of domain-expansion safe attributes in step $n+1$ thus depends on the TBF, which in turn depends on the domain-expansion safe attributes from step n.

Definition 63 ((Liberal) Domain-Expansion Safety)**.** Let b be a TBF. The set of *bounded terms* $B_n(r, \Pi, b)$ in a rule $r \in \Pi$ in step $n \geq 1$ is defined as

$$B_n(r, \Pi, b) = \bigcup_{j \geq 0} B_{n,j}(r, \Pi, b),$$

where $B_{n,0}(r, \Pi, b) = \emptyset$ and for $j \geq 0$,

$$B_{n,j+1}(r, \Pi, b) = b(\Pi, r, S_{n-1}(\Pi), B_{n,j}).$$

The set of *domain-expansion safe attributes* $S_\infty(\Pi) = \bigcup_{i \geq 0} S_i(\Pi)$ of a program Π is iteratively constructed, where $S_0(\Pi) = \emptyset$ and for $S_{n+1}(\Pi)$ with $n \geq 0$ we have

- $p\restriction i \in S_{n+1}(\Pi)$ if for each $r \in \Pi$ and atom $p(t_1, \ldots, t_{ar(p)}) \in H(r)$, $t_i \in B_{n+1}(r, \Pi, b)$, i.e., t_i is *bounded*;
- $\&g[\mathbf{Y}]_r\restriction_{\mathrm{I}} i \in S_{n+1}(\Pi)$ if each Y_i is a *bounded* variable, or Y_i is a predicate input parameter p and $p\restriction 1, \ldots, p\restriction ar(p) \in S_n(\Pi)$; and
- $\&g[\mathbf{Y}]_r\restriction_{\mathrm{O}} i \in S_{n+1}(\Pi)$ if r contains an external atom $\&g[\mathbf{Y}](\mathbf{X})$ such that either X_i is bounded or $\&g[\mathbf{Y}]_r\restriction_{\mathrm{I}} 1, \ldots, \&g[\mathbf{Y}]_r\restriction_{\mathrm{I}} ar_{\mathrm{I}}(\&g) \in S_n(\Pi)$.

A program Π is *(liberally) domain-expansion safe*, if it is safe and all its attributes are domain-expansion safe.

An example is delayed until we have introduced concrete TBFs in Section 4.2.2. However, the intuition is as follows. In each step, the TBF first derives terms which are bounded, exploiting

e.g. syntactic or semantic criteria. This possibly makes additional attributes domain-expansion safe, which may trigger in turn further terms to become bounded in the next step.

One can show that $S_\infty(\Pi)$ is finite, thus the inductive definition can be used for computing $S_\infty(\Pi)$: the iteration can be aborted after finitely many steps. We first note this and other desired properties.

Proposition 4.1. *The set $S_\infty(\Pi)$ is finite.*

Proof. The sets $\mathcal{P}$, $\mathcal{X}$ and Π are finite and each ordinary and external predicate has a finite (input and output) arity. Therefore there exists only a finite number of attributes. □

Moreover, domain-expansion safe attributes have a finite range in $G^\infty_\Pi(\emptyset)$.

Proposition 4.2. *For every TBF b and $n \geq 0$, if $\alpha \in S_n(\Pi)$, then the range of α in $G^\infty_\Pi(\emptyset)$ is finite.*

Proof. We prove this by induction on n.

For $n = 0$ we have $S_0(\Pi) = \emptyset$ and the proposition holds trivially.

For the induction step $n \mapsto n+1$, assume that the attributes in $S_n(\Pi)$ are domain-expansion safe (outer induction hypothesis). We first show that for each rule r and term $t \in B_{n+1}(r, \Pi, b)$, the set of ground instances of r in $G^\infty_\Pi(\emptyset)$ contains only finitely many different substitutions for t. We consider $B_{n+1,j}(r, \Pi, b)$ and again prove this by induction on j. For $j = 0$ we have $B_{n+1,0}(r, \Pi, b) = \emptyset$ and the proposition holds trivially. For the induction step $j \mapsto j+1$, assume that the terms in $B_{n+1,j}(r, \Pi, b)$ are bounded (inner induction hypothesis). Let $t \in B_{n+1,j+1}(r, \Pi, b)$. If $t \in B_{n+1,j}(r, \Pi)$ then the claim follows from the inner induction hypothesis. Otherwise t is added in step $j+1$. By the outer induction hypothesis all attributes in $S_n(\Pi)$ have a finite range in $G^\infty_\Pi(\emptyset)$. By the inner induction hypothesis there are only finitely many substitutions for all terms $t \in B_{n+1,j}(r, \Pi, b)$ in $G^\infty_\Pi(\emptyset)$. This fulfills the conditions of TBFs (Definition 62). Since b is a TBF, this implies that there are also only finitely many substitutions for all $t \in b(r, S_n(\Pi), B_{n+1,j})$. This proves the inner induction statement and, by definition of $B_n(r, \Pi, b)$, also that for each $t \in B_{n+1}(r, \Pi, b)$ the set of ground instances of r in $G^\infty_\Pi(\emptyset)$ contains only finitely many different substitutions for t.

If $p{\restriction}i \in S_{n+1}(\Pi)$, then for each rule $r \in \Pi$ and atom $p(t_1, \ldots, t_{ar(p)}) \in H(r)$ we have $t_i \in B_{n+1}(r, \Pi, b)$. As we have shown, this means that there are only finitely many different substitutions for t_i in the ground instances of r in the set $G^\infty_\Pi(\emptyset)$. As there are also only finitely many different rules in Π, and the number of substitutions for the term t_i in the head of r is finite, this implies that also the set $\{t_i \mid p(t_1, \ldots, t_{ar(p)}) \in A(G^\infty_\Pi(\emptyset))\}$ is finite.

If $\&g[\mathbf{Y}]_r{\restriction}_\mathrm{I} i \in S_{n+1}(\Pi)$, then the i-th input parameter is either of type constant and Y_i is a constant or a variable, or it is of type predicate. If it is of type constant and Y_i is a constant, then there exists only one ground instance. If it is of type constant and Y_i is a variable, then $Y_i \in B_{n+1}(r, \Pi, b)$, for which we have shown that there are only finitely many different substitutions for Y. If it is of type predicate input parameter p, then the range of all attributes $p{\restriction}1, \ldots, p{\restriction}ar(p)$ in $G^\infty_\Pi(\emptyset)$ is finite by the (outer) induction hypothesis.

If $\&g[\mathbf{Y}]_r{\restriction}_\mathrm{O} i \in S_{n+1}(\Pi)$, then either $\&g[\mathbf{Y}]_r{\restriction}_\mathrm{I} 1, \ldots, \&g[\mathbf{Y}]_r{\restriction}_\mathrm{I} ar_\mathrm{I}(\&g) \in S_n(\Pi)$, or r contains an external atom $\&g[\mathbf{Y}](\mathbf{X})$ s.t. Y_i is bounded. If $\&g[\mathbf{Y}]_r{\restriction}_\mathrm{I} 1, \ldots, \&g[\mathbf{Y}]_r{\restriction}_\mathrm{I} ar_\mathrm{I}(\&g) \in$

$S_n(\Pi)$, then the range of all input parameters in $G_\Pi^\infty(\emptyset)$ is finite by the (outer) induction hypothesis. But then there exist only finitely many oracle calls to $\&g$. As each such call can introduce only finitely many new values, also the range of each output parameter in $G_\Pi^\infty(\emptyset)$ is finite. If r contains an external atom $\&g[\mathbf{Y}](\mathbf{X})$ such that Y_i is bounded, then only finitely many substitutions for $\&g[\mathbf{Y}]_r\restriction_{\mathrm{O}} i$ can satisfy the rule body, thus $G_\Pi^\infty(\Pi)$ will also contain only finitely many values for $\&g[\mathbf{Y}]_r\restriction_{\mathrm{O}} i$. Thus, the (outer) induction hypothesis holds for $n+1$, which proves the statement. □

Corollary 4.1. *If $\alpha \in S_\infty(\Pi)$, then $range\big(\alpha, G_\Pi^\infty(\emptyset)\big)$ is finite.*

Proof. If $a \in S_\infty$ then $a \in S_n$ for some $n \geq 0$ and the claim follows from Proposition 4.2. □

From this result if follows that also $G_\Pi^\infty(\emptyset)$ is finite.

Corollary 4.2. *If Π is a domain-expansion safe program, then $G_\Pi^\infty(\emptyset)$ is finite.*

Proof. Since Π is domain-expansion safe by assumption, $a \in S_\infty(\Pi)$ for all attributes a of Π. Then by Corollary 4.1, the range of all attributes of Π in $G_\Pi^\infty(\emptyset)$ is finite. But then there exists also only a finite number of ground atoms in $G_\Pi^\infty(\emptyset)$. Since the original non-ground program Π is finite, this implies that also the grounding is finite. □

It follows from these propositions that $S_\infty(\Pi)$ is also finitely constructible. Note that they hold independently of a concrete TBF, which is because the properties of TBFs are sufficiently strong.

4.2.2 Concrete Term Bounding Functions

We now introduce concrete term bounding functions that exploit syntactic and semantic properties of external atoms to guarantee boundedness of variables. Consequently this ensures also finiteness of the ground program given by $G_\Pi^\infty(\emptyset)$.

Syntactic Criteria

We first identify syntactic properties that can be exploited for our purposes.

Definition 64 (Syntactic Term Bounding Function)**.** We define $b_{syn}(\Pi, r, S, B)$ such that we have $t \in b_{syn}(\Pi, r, S, B)$ if

(i) t is a constant in r; or

(ii) there is an ordinary atom $q(s_1, \ldots, s_\ell) \in B^+(r)$ s.t. $t = s_j$, for some $1 \leq j \leq \ell$ and $q\restriction j \in S$; or

(iii) for some external atom $\&g[\mathbf{Y}](\mathbf{X}) \in B^+(r)$, we have that $t = X_i$ for some $X_i \in \mathbf{X}$, and for each $Y_i \in \mathbf{Y}$,

$$\begin{cases} Y_i \in B, & \text{if } type(\&g, i) = \mathbf{const}, \\ Y_i\restriction 1, \ldots, Y_i\restriction ar(Y_i) \in S, & \text{if } type(\&g, i) = \mathbf{pred}. \end{cases}$$

Intuitively, Case (i) defines a constant as bounded because it is never substituted by other terms in the grounding. In Case (ii) the precondition that an attribute $q\restriction j$ for some $1 \leq j \leq ar(q)$ is domain-expansion safe, and thus has a finite range in $G^{\infty}_{\Pi}(\emptyset)$, implies that the term at this attribute is bounded. Case (iii) essentially states that if the input to an external atom is finite, then also its output is finite.

Lemma 4.1. *Function $b_{syn}(\Pi, r, S, B)$ is a TBF.*

Proof. If t is in the output of $b_{syn}(\Pi, r, S, B)$, then one of the conditions holds.

If Condition (i) holds, then t is a constant, hence there is only one ground instance.

If Condition (ii) holds, then t must also occur as value for $q\restriction j$, which has a finite range by assumption.

If Condition (iii) holds, then t is output of an external atom such that there are only finitely many substitutions of its constant inputs and the attributes of all predicate inputs have a finite range by assumption. Thus there are only finitely many different oracle calls with finite output each. □

Example 52 (ctd.)**.** Consider program Π from Example 51. We get the set $S_1(\Pi) = \{dom\restriction 1, \&cat[X, x]_{r_4}\restriction_{\mathrm{I}} 2\}$, as $B_1(r_2, \Pi, b_{syn}) = \{ax\}$, $B_1(r_3, \Pi, b_{syn}) = \{axx\}$ and $B_1(r_4, \Pi, b_{syn}) = \{x\}$ (by (i) in Definition 64), i.e., the derived terms in all rules that have $dom\restriction 1$ in their head are known to be bounded. In the next iteration, we get $B_2(r_4, \Pi, b_{syn}) = \{Y\}$ (by (ii) in Definition 64) as $dom\restriction 1$ is already known to be de-safe. Since we also have $B_2(r_1, \Pi, b_{syn}) = \{a\}$, the terms derived by r_1 and r_4 are bounded, hence $s\restriction 1 \in S_2(\Pi)$. Moreover, $\&cat[X, x]_{r_4}\restriction_{\mathrm{O}} 1 \in S_2(\Pi)$ because Y is bounded. The third iteration yields that attribute $\&cat[X, x]_{r_4}\restriction_{\mathrm{I}} 1 \in S_3(\Pi)$ because $X \in B_3(r_4, \Pi, b_{syn})$ due to (ii) in Definition 64. Thus, all attributes are de-safe. □

Semantic Properties

We now define a TBF exploiting meta-information about external sources in three properties.

The first property is based on *malign cycles* in *positive attribute dependency graphs*, which are the source of any infinite value invention. Intuitively, the *positive attribute dependency graph* $G_A(\Pi)$ has as nodes the attributes of Π and its edges model the information flow between the attributes. For instance, if for rule r we have $p(\mathbf{X}) \in H(r)$ and $q(\mathbf{Y}) \in B^+(r)$ s.t. $X_i = Y_j$ for some $X_i \in \mathbf{X}$ and $Y_j \in \mathbf{Y}$, then we have a flow from $q\restriction j$ to $p\restriction i$. Formally:

Definition 65 (Positive Attribute Dependency Graph)**.** The *positive attribute dependency graph* $G_A(\Pi) = \langle Attr, E\rangle$ of a program Π has as nodes $Attr$ the set of all attributes in Π and as edges the least set E such that for all $r \in \Pi$:

- If $p(\mathbf{X}) \in H(r)$, $q(\mathbf{Y}) \in B^+(r)$ and for some i, j we have that $X_i = Y_j$ is a variable, then $(q\restriction j, p\restriction i) \in E$.
- If $\&g[\mathbf{Y}](\mathbf{X}) \in B^+(r)$, $p(\mathbf{Z}) \in B^+(r)$ and for some i, j we have that $Z_i = Y_j$ and $type(\&g, i) = \mathbf{const}$, then $(p\restriction i, \&g[\mathbf{Y}]_r\restriction_{\mathrm{I}} j) \in E$.
- If $\&g[\mathbf{Y}](\mathbf{X}) \in B^+(r)$, $\&h[\mathbf{U}](\mathbf{V}) \in B^+(r)$ and for some i, j we have that $U_i = X_j$ and $type(\&h, i) = \mathbf{const}$, then $(\&g[\mathbf{Y}]_r\restriction_{\mathrm{O}} j, \&h[\mathbf{U}]_r\restriction_{\mathrm{I}} i) \in E$.

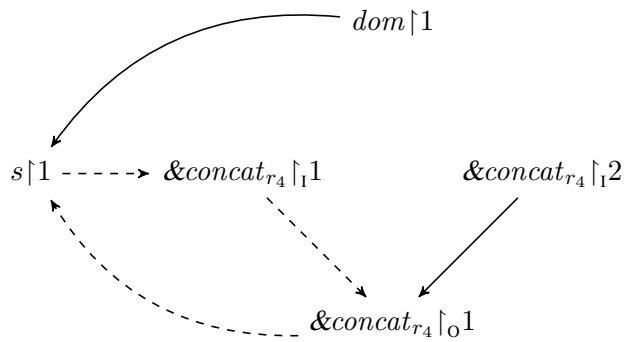


Figure 4.4: Positive Attribute Dependency Graph $G_A(\Pi)$ of the Program Π from Example 51 with a Cycle (dashed)

- If $\&g[\mathbf{Y}](\mathbf{X}) \in B^+(r)$, then $(\&g[\mathbf{Y}]_r\restriction_{\mathrm{I}} i, \&g[\mathbf{Y}]_r\restriction_{\mathrm{O}} j) \in E$ for all $1 \leq i \leq ar_{\mathrm{I}}(\&g)$ and $1 \leq j \leq ar_{\mathrm{O}}(\&g)$.
- If $p(\mathbf{X}) \in H(r)$, $\&g[\mathbf{Y}](\mathbf{Z}) \in B^+(r)$ and for some i, j we have that $X_i = Z_j$ is a variable, then $(\&g[\mathbf{X}]_r\restriction_{\mathrm{O}} j, p\restriction i) \in E$.
- If $\&g[\mathbf{Y}](\mathbf{X}) \in B^+(r)$ such that $p = Y_i$ and $type(\&g, i) = \mathbf{pred}$, then $(p\restriction k, \&g[\mathbf{Y}]_r\restriction_{\mathrm{I}} i) \in E$ for all $1 \leq k \leq ar(p)$.

Example 53 (ctd.)**.** The positive attribute dependency graph of the program from Example 51 is as shown in Figure 4.4. □

Graph $G_A(\Pi)$ models the direct information flow in Π, while its transitive closure models the indirect information flow.

Definition 66 (Benign and Malign Cycles)**.** A cycle K in $G_A(\Pi)$ is *benign* wrt. a set S of safe attributes, if there exists a well-ordering $\leq_{\mathcal{C}}$ of $\mathcal{C}$, s.t. for every $\&g[\mathbf{Y}]_r\restriction_{\mathrm{O}} j \notin S$ in the cycle, $f_{\&g}(\mathbf{A}, y_1, \ldots, y_m, t_1, \ldots t_n) = 0$, whenever

- for some input parameter $1 \leq i \leq ar_{\mathrm{I}}(\&g)$, $type(\&g, i) = \mathbf{pred}$, $\&g[\mathbf{Y}]_r\restriction_{\mathrm{I}} i \notin S$ is in the cycle K, $(s_1, \ldots, s_{ar(y_i)}) \in ext(\mathbf{A}, y_i)$, and $t_j \not\leq_{\mathcal{C}} s_k$ for some $1 \leq k \leq ar(y_i)$; or
- some y_i for $1 \leq i \leq ar_{\mathrm{I}}(\&g)$ is a constant input parameter, $\&g[\mathbf{Y}]_r\restriction_{\mathrm{I}} i \notin S$ is in K, and $t_j \not\leq_{\mathcal{C}} y_i$.

A cycle in $G_A(\Pi)$ is called *malign* wrt. S if it is not benign.

Intuitively, a cycle is benign if external atoms never deliver larger values wrt. to their yet unsafe cyclic input. As there is a least element, this ensures a finite grounding.

Example 54 (ctd.)**.** The cycle (dashed) in $G_A(\Pi)$ of Π from Example 51 (see Figure 4.4) is malign wrt. $S = \emptyset$ because there does not exist a well-ordering as required by Definition 66. Intuitively, this is because the external atom infinitely extends the string. However, it is benign wrt. $S = \{s\restriction 1\}$. □

Three other properties involve meta-information which directly ensures that an output attribute of an external source is finite.

Definition 67 (Finite Domain)**.** An external predicate $\&g \in \mathcal{X}$ has the *finite domain property wrt. output* $i \in \{1, \ldots, ar_{\mathrm{O}}(\&g)\}$, if $\{x_i \mid \mathbf{y} \in (\mathcal{P} \cup \mathcal{C})^{ar_{\mathrm{I}}(\&g)}, \mathbf{x} \in \mathcal{C}^{ar_{\mathrm{O}}(\&g)}, f_{\&g}(\mathbf{A}, \mathbf{y}, \mathbf{x}) = 1\}$ is finite for all assignments $\mathbf{A}$.

Here, the provider of the external source explicitly states that the output at a certain position in the output tuple is finite. This is perhaps the most direct way to ensure boundedness of the respective term.

Example 55. An external atom $\&md5[S](Y)$ computing the MD5 hash value Y of a string S is finite domain wrt. the (single) output element, as its domain is finite (yet very large). □

A relaxed notion of finiteness allows for open domains, but forbids constants in the output of an external source which do not already appear in the extension of the respective input predicate parameter.

Definition 68 (Relative Finite Domain)**.** An external predicate $\&g \in \mathcal{X}$ has the *relative finite domain property wrt. output argument* $i \in \{1, \ldots, ar_{\mathrm{O}}(\&g)\}$ *and predicate input argument* $j \in 1, \ldots, ar_{\mathrm{I}}(\&g)$, if $\{x_i \mid \mathbf{y} \in (\mathcal{P} \cup \mathcal{C})^{ar_{\mathrm{I}}(\&g)}, \mathbf{x} \in \mathcal{C}^{ar_{\mathrm{O}}(\&g)}, f_{\&g}(\mathbf{A}, \mathbf{y}, \mathbf{x}) = 1\} \subseteq \{c \in \mathbf{c} \mid \mathbf{c} \in ext(\mathbf{A}, y_j)\}$ is finite for all assignments $\mathbf{A}$.

Example 56. An external atom $\&diff[dom, set](Y)$ has the relative finite domain property wrt. output argument 1 and predicate input argument 1 because each constant c must already occur in the extension of dom wrt. $\mathbf{A}$ if $f_{\&g}(\mathbf{A}, dom, set, c) = 1$. □

While the previous properties conclude that an output term of an external atom is bounded if there are only finitely many different input constants and interpretations, we now reverse the direction. An external atom may have the property that only a finite number of different inputs can yield a certain output, which is formalized as follows.

Definition 69 (Finite Fiber)**.** An external predicate $\&g \in \mathcal{X}$ has the *finite fiber property*, if $\{\mathbf{y} \mid \mathbf{y} \in (\mathcal{P} \cup \mathcal{C})^{ar_{\mathrm{I}}(\&g)}, f_{\&g}(\mathbf{A}, \mathbf{y}, \mathbf{x}) = 1\}$ is finite for every $\mathbf{A}$ and $\mathbf{x} \in \mathcal{C}^{ar_{\mathrm{O}}(\&g)}$.

Example 57. Let $\&square[X](S)$ be an external atom that computes the square S of the number X. Then for some given S, there are at most two distinct values for X. □

The four properties above lead to the following TBF.

Definition 70 (Semantic Term Bounding Function)**.** We define $b_{sem}(\Pi, r, S, B)$ such that we have $t \in b_{sem}(\Pi, r, S, B)$ if

(i) t is captured by some attribute α in $B^+(r)$ that is not reachable from malign cycles in $G_A(\Pi)$ wrt. S, i.e., if $\alpha = p{\restriction}i$ then $t = t_i$ for some body atom $p(t_1, \ldots, t_\ell) \in B^+(r)$, and if $\alpha = \&g[\mathbf{Y}]_r{\restriction}_T i$ then $t = Y_i^T$ for some $\&g[\mathbf{Y}^{\mathrm{I}}](\mathbf{Y}^{\mathrm{O}}) \in B^+(r)$ where $\mathbf{Y}^T = X_1^T, \ldots, Y_{ar(\&g)}^T$; or

(ii) $t = X_i$ for some $\&g[\mathbf{Y}](\mathbf{X}) \in B^+(r)$, where $\&g$ has the *finite domain property in* i; or

(iii) $t = X_i$ for some $\&g[\mathbf{Y}](\mathbf{X}) \in B^+(r)$, where $\&g$ has the *relative finite domain property in output argument* i *and predicate input argument* j and $Y_j\restriction k \in S$ for all $1 \leq k \leq ar(Y_j)$; or

(iv) $t \in \mathbf{Y}$ for some $\&g[\mathbf{Y}](\mathbf{X}) \in B^+(r)$, where $X \in B$ for every $X \in \mathbf{X}$ and $\&g$ has the *finite fiber property*.

This TBF is directly motivated by the introduced properties.

Lemma 4.2. *Function* $b_{sem}(\Pi, r, S, B)$ *is a TBF.*

Proof. If t is in the output of $b_{sem}(\Pi, r, S, B)$, then one of the conditions holds.

If Condition (i) holds, then there is no information flow from malign cycles wrt. S to t. However, such cycles are the only source of infinite groundings: the attributes in S have a finite domain by assumption. For the remaining attributes in the cycle, the well-ordering guarantees that only finitely many different values can be produced in the cycle.

If Condition (ii) holds, then the claim follows immediately from finiteness of the domain of the respective external atom.

If Condition (iii) holds, then the external atom cannot introduce new constants. Because the set of constants in the extension of the respective input parameter Y_j is finite by assumption that $Y_j\restriction k \in S$ for all $1 \leq k \leq ar(Y_j)$, it follows that also the set of constants in the output of the external atom is finite.

If Condition (iv) holds, then there are only finitely many different substitutions for t because the output of the respective external atom is bound by the precondition of TBFs and the finite fiber ensures that there are only finitely many different inputs for each output. □

4.2.3 Combination of Term Bounding Functions

The concept of liberal domain-expansion safety based on term bounding functions can be fruitfully exploited for easy extensions. In particular, multiple term bounding functions may be combined in order to further relax the syntactic and semantic criteria.

The following proposition allows us to construct TBFs modularly from multiple TBFs.

Proposition 4.3. *If* $b_i(\Pi, r, S, B)$, $1 \leq i \leq \ell$, *are TBFs, then the union of the term bounding functions*

$$b(\Pi, r, S, B) = \bigcup_{1 \leq i \leq \ell} b_i(\Pi, r, S, B)$$

is also a TBF.

Proof. For $t \in b(\Pi, r, S, B)$, $t \in b_i(\Pi, r, S, B)$ for some $1 \leq i \leq \ell$. Then there are only finitely many substitutations for t in $G^{\infty}_{\Pi}(\emptyset)$ because b_i is a TBF. □

In particular, a TBF which exploits syntactic and semantic properties simultaneously is

$$b_{synsem}(\Pi, r, S, B) = b_{syn}(\Pi, r, S, B) \cup b_{sem}(\Pi, r, S, B),$$

which we will use subsequently.

4.2.4 Finite Restrictability

We now make use of the results from above to show that domain-expansion safe programs are finitely restrictable in an effective manner. Recall that $\equiv^{pos}$ denotes equivalence of the answer sets in their positive parts.

Theorem 5 (Finite Restrictability of Domain-Expansion Safe Programs)**.** *Let* Π *be a domain-expansion safe program. Then* Π *is finitely restrictable and* $G_\Pi^\infty(\emptyset) \equiv^{pos} \Pi$.

Proof. We construct the grounding $grnd_C(\Pi)$ as the least fixpoint $G_\Pi^\infty(\emptyset)$ of the grounding operator $G_\Pi(X)$, which is known to be finite by Corollary 4.2. The set C is then implicitly given by the set of constants appearing in $grnd_C(\Pi)$. It remains to show that indeed $grnd_C(\Pi) \equiv^{pos} grnd_{\mathcal{C}}(\Pi)$. To make the proof reusable for Proposition 4.5, we will show the more general proposition $grnd_C(\Pi) \equiv^{pos} grnd_{C'}(\Pi)$ for any $C' \supseteq C$.

($\Rightarrow$) Suppose $\mathbf{A} \in \mathcal{AS}\big(grnd_C(\Pi)\big)$. Let $\mathbf{A}' = \mathbf{A} \cup \big\{\mathbf{F}a \mid a \in A\big(grnd_{C'}(\Pi)\big), \mathbf{T}a \notin \mathbf{A}\big\}$, i.e., the completion of $\mathbf{A}$ to all atoms in $grnd_{C'}(\Pi)$ by setting all additional atoms to false. Then $\{\mathbf{T}a \in \mathbf{A}\} = \{\mathbf{T}a \in \mathbf{A}'\}$. We show now that $\mathbf{A}'$ is an answer set of $grnd_{C'}(\Pi)$. First observe that $\mathbf{A}' \not\models B^+(r)$ for all $r \in grnd_{C'}(\Pi) \setminus grnd_C(\Pi)$; otherwise $r \in G_\Pi\big(grnd_C(\Pi)\big)$, which contradicts the assumption that $grnd_C(\Pi)$ is the least fixpoint of $G_\Pi(\emptyset)$. Hence, $\mathbf{A}' \models grnd_{C'}(\Pi)$. Moreover $fgrnd_C(\Pi)^{\mathbf{A}} = fgrnd_{C'}(\Pi)^{\mathbf{A}'}$, hence $\mathbf{A}'$ is a subset-minimal model of the FLP-reduct of $grnd_{C'}(\Pi)$ iff $\mathbf{A}$ is a subset-minimal model of the FLP-reduct of $grnd_C(\Pi)$, which is the case because $\mathbf{A} \in \mathcal{AS}\big(grnd_C(\Pi)\big)$. Therefore $\mathbf{A}' \in \mathcal{AS}\big(grnd_{C'}(\Pi)\big)$.

($\Leftarrow$) Now suppose $\mathbf{A} \in \mathcal{AS}\big(grnd_{C'}(\Pi)\big)$. Then $\mathbf{A}' = \mathbf{A} \cap \big\{\mathbf{T}a, \mathbf{F}a \mid a \in A\big(grnd_C(\Pi)\big)\big\}$ is a model of $grnd_C(\Pi)$. Let $\mathbf{A}'' = \mathbf{A}' \cup \{\mathbf{F}a \mid a \in A\big(grnd_{C'}(\Pi)\big), \mathbf{T}a \notin \mathbf{A}'\}$, i.e., the completion of $\mathbf{A}'$ to all atoms in $grnd_{C'}(\Pi)$ by setting all additional atoms to false. Then $\mathbf{A}'' \not\models B^+(r)$ for all $r \in grnd_{C'}(\Pi) \setminus grnd_C(\Pi)$; otherwise $r \in G_\Pi\big(grnd_C(\Pi)\big)$, which contradicts the assumption that $grnd_C(\Pi)$ is the least fixpoint of $G_\Pi(\emptyset)$. Therefore, $\mathbf{A}'' \models grnd_{C'}(\Pi)$. But this implies that $\mathbf{A} = \mathbf{A}''$: by construction of $\mathbf{A}''$ we have $\mathbf{A}''^{\mathbf{T}} \subseteq \mathbf{A}^{\mathbf{T}}$, and $\mathbf{A}''^{\mathbf{T}} \subsetneq \mathbf{A}^{\mathbf{T}}$ would imply that $\mathbf{A}$ is not subset-minimal, which contradicts the assumption that $\mathbf{A} \in \mathcal{AS}\big(grnd_{C'}(\Pi)\big)$. Moreover, $fgrnd_C(\Pi)^{\mathbf{A}'} = fgrnd_{C'}(\Pi)^{\mathbf{A}''}$. Hence $\mathbf{A}'$ is a subset-minimal model of the FLP-reduct of $grnd_C(\Pi)$ iff $\mathbf{A}''$ is a subset-minimal model of the FLP-reduct of $grnd_{C'}(\Pi)$, which is the case because $\mathbf{A}'' \in \mathcal{AS}\big(grnd_{C'}(\Pi)\big)$. Therefore $\mathbf{A}' \in \mathcal{AS}\big(grnd_C(\Pi)\big)$. The observation $\{\mathbf{T}a \in \mathbf{A}'\} = \{\mathbf{T}a \in \mathbf{A}''\}$ concludes the proof. □

This proposition holds independently of a concrete term bounding function. However, functions that are too liberal are excluded by the preconditions in the definition of TBFs.

The operator G_Π is exponential in the number of ground atoms as it considers all assignments $\mathbf{A} \subseteq \mathcal{A}(\Pi')$ in every step. As this compromises efficiency, a better alternative is:

$$R_\Pi(\Pi') = \bigcup_{r \in \Pi} \big\{r\theta \mid \{\mathbf{T}a \mid a \in A(\Pi')\} \models B^+(r\theta)\big\}.$$

Intuitively, instead of enumerating exponentially many assignments it simply maximizes the output of external atoms by setting all input atoms to true, which is possible due to monotonicity.

Proposition 4.4. *Let Π be a domain-expansion safe program s.t. each nonmonotonic predicate input parameter to external atoms occurs only in facts. Then $G_\Pi^\infty(\emptyset) = R_\Pi^\infty(\emptyset)$.*

Proof. It suffices to show for any intermediate result X that we have $\mathbf{A} \models B^+(r)$ for some $\mathbf{A} \subseteq \{\mathbf{T}a, \mathbf{F}a \mid a \in A(X)\} \setminus \{\mathbf{F}a \mid a \leftarrow . \in \Pi\}$ if and only if $\mathbf{A'} \models B^+(r)$ for $\mathbf{A'} = \{\mathbf{T}a \mid a \in A(X)\}$.

($\Rightarrow$) $B^+(r)$ contains only positive ordinary atoms a, hence $\mathbf{A} \models a$ implies $\mathbf{A'} \models a$. For external atoms e, $\mathbf{A} \models e$ but $\mathbf{A'} \not\models e$ is only possible if for some input atom a to e over a nonmonotonic predicate parameter we have $\mathbf{F}a \in \mathbf{A}$ but $\mathbf{T}a \in \mathbf{A'}$. But by assumption, nonmonotonic predicate parameters do only occur in facts, hence $\mathbf{F}a \in \mathbf{A}$ is impossible.

($\Leftarrow$) Trivial. □

Thus, for such programs we may compute a sufficient finite subset of $grnd_\mathcal{C}(\Pi)$ using instead G_Π the more efficient R_Π.

Example 58 (ctd.)**.** In Example 51, $\&concat[X, x](Y)$ is monotonic, hence we can use R_Π for restricted grounding. □

The operator can also be optimized in a different way. External atoms that are not relevant for domain-expansion safety can be removed from the fixpoint iteration without affecting correctness of the grounding. For each $r \in \Pi$, let $\bar{r} = H \leftarrow B$ be any rule such that $r = H \leftarrow b_1, \ldots, b_h, B$ where $b_1, \ldots, b_h \in EA(r)$ and $var(r) = var(\bar{r})$, where $var(r)$ denotes the set of variables from $\mathcal{V}$ appearing in rule r. That is, $\bar{r}$ results from r by possibly dropping external atoms from $B^+(r)$ but such that $\bar{r}$ contains all variables of r, and let $\bar{\Pi} = \{\bar{r} \mid r \in \Pi\}$ by a liberally domain-expansion safe program.

We then define the following monotone operator:

$$Q_\Pi(\Pi') = \bigcup_{r \in \Pi} \{r\theta \mid \exists \mathbf{A} \subseteq \mathcal{A}(\Pi'), \mathbf{A} \not\models \bot, \mathbf{A} \models B^+(\bar{r}\theta)\}.$$

The intuition is that removing atoms from rule bodies makes rule applicability (possibly) more frequent, which may result in a larger (but still finite) grounding. As this grounding is a superset of the one computed by $G_\Pi(\Pi')$, it is still answer set preserving.

Proposition 4.5. *For every program Π, $Q_\Pi^\infty(\emptyset) \subseteq grnd_\mathcal{C}(\Pi)$ is finite and $Q_\Pi^\infty(\emptyset) \equiv^{pos} G_\Pi^\infty(\emptyset)$.*

Proof. Clearly, satisfaction of rule body $B^+(r)$ implies satisfaction of the corresponding $B^+(\bar{r})$. Thus $Q_\Pi^\infty(\emptyset) \supseteq G_\Pi^\infty(\emptyset)$. It follows then from the proof of Theorem 5 that $G_\Pi^\infty(\emptyset) \equiv^{pos} Q_\Pi^\infty(\emptyset)$.

Moreover, since $\bar{\Pi}$ is still domain-expansion safe by assumption, $G_{\bar{\Pi}}^\infty(\emptyset)$ is still finite by Theorem 5. But then also $Q_\Pi^\infty(\emptyset)$ is finite because the only difference between $G_{\bar{\Pi}}^\infty(\emptyset)$ and $Q_\Pi^\infty(\emptyset)$ is that the latter may contain additional external atoms in the rule bodies. □

Example 59 (ctd.)**.** In the program in Example 51, the external atom $\&concat[X, x](Y)$ is not needed to establish domain-expansion safety, hence we might drop it during fixpoint iteration. □

The *combination* of the optimizations is especially valuable. One can first eliminate external atoms with nonmonotonic input other than facts and check then rule body satisfaction as in R_{Π}. If an external atom b is strongly safe wrt. the according rule and the program, then it is very often (as in Example 51) not necessary for establishing domain-expansion safety and b is a candidate for being removed. That is, the traditional strong safety criterion is now used as a *weak criterion*, which is not strictly necessary but may help to reduce grounding time. We will build upon this idea when designing our new grounding algorithm in Section 4.3.

4.2.5 Applications

Pushdown Automaton. As a demonstration of our relaxed notion of safety, we model a *pushdown automaton* in a HEX-program, which can be of use if context-free languages must be parsed under further constraints that cannot be easily expressed in the production rules; the HEX-program may be extended accordingly, where the declarative nature of HEX is versatile for parsing and constraint checking in parallel as opposed to a generate-and-filter approach.

For instance, consider RNA sequences over the alphabet $\{a, g, c, u\}$ and suppose we want to accept all sequences ww' such that w' is the complementary string of w, where (a, u) and (g, c) are complementary pairs. Because complementary strings within one sequence influence the secondary structure of an RNA molecule, this duality is important for its proper function [Zuker and Sankoff, 1984]. This language is easily expressed by the production rules

$$\{S \to aSu, S \to uSa, S \to gSc, S \to cSg, S \to \epsilon\}$$

with start symbol S. Now suppose that we want to check in addition the occurrence of certain subsequences, e.g., because they have a known function. A concrete example would be a *promoter sequence* which identifies the location where a new gene starts and might be used to distinguish between coding and non-coding sequences. Modeling this in the production rules makes the grammar much more complex. Moreover, we might want to keep the grammar independent of concrete subsequences but import them from a database. Then it might be useful to model the basic language as automaton in a logic program and check side conditions by additional constraints.

Recall that a pushdown automaton is a finite-state machine with an additional stack; following Sipser (2012), this is formalized as a tuple $(Q, \Sigma, \Gamma, \delta, q_0, Z, F)$, where

- Q is a finite set of states;
- Σ is a finite input alphabet;
- Γ is a finite stack alphabet;
- $\delta \subseteq Q \times \big(\Sigma \cup \{\epsilon\}\big) \times \Gamma \times Q \times \Gamma^*$ is the transition relation;

- $q_0 \in Q$ is the initial state;
- $Z \in \Gamma$ is the initial stack symbol; and
- $F \subseteq Q$ is the set of final states.

The transition relation maps the current state, an input symbol and the topmost stack symbol to a successor state and a finite word over the stack alphabet, which is pushed onto the stack after removing the topmost symbol.

We make the assumption that there are no ϵ-transitions, i.e., $\delta \subseteq Q \times \Sigma \times \Gamma \times Q \times \Gamma^*$; such an automaton is easily obtained from a normalized grammar, if one is not interested, as in our example, in the empty word.

We use the following external atoms:

- $\&car[S](H, T)$ splits S into first symbol H and rest T;
- $\&concat[A, B](C)$ joins A and B to C;
- $\&inc[I](I1)$ increments the integer I to $I1 = I+1$; and
- $\&len[S](L)$ returns the length L of string S.

Then the automaton can be modeled as follows:

$$str(Word, 0) \leftarrow input(Word) \tag{1}$$
$$str(R, I1) \leftarrow str(W, I), \&car[W](C, R), \&inc[I](I1) \tag{2}$$
$$char(C, I) \leftarrow str(W, I), \&car[W](C, R) \tag{3}$$
$$in(start, z, 0) \tag{4}$$
$$\begin{aligned} in(NewState, NewStack, NewPos) \leftarrow \\ in(State, Stack, Pos), char(Pos, Char), \\ \&car[Stack](SChar, SRest), \\ transition(State, Char, SChar, NewState, Push), \\ \&concat[Push, SRest](NewStack), \&inc[Pos](NewPos) \end{aligned} \tag{5}$$
$$accept \leftarrow input(W), \&len[W](L), in(S, z, L), final(S) \tag{6}$$
$$\leftarrow \text{not } accept \tag{7}$$

An atom $in(state, stack, step)$ encodes that when processing symbol $step$, the machine is in state $state$ with stack content $stack$. Rules (1)-(3) split the input string into characters, and the remaining ones model the automaton. The program starts in the initial state $start$ with the initial stack symbol z as stack content (fact (4)). Transition rule (5) splits the current stack content into its topmost symbol $SChar$ and its rest $SRest$ and uses the predicate $transition$ to (nondeterministically) determine the successor state and the string to push onto the stack. The rules (6)-(7) ensure that the input is accepted if eventually a final state is reached such that the input has been completely processed and the stack content is z. Side conditions can now be

modeled, e.g., by additional constraints which restrict the stack content, or by additional body atoms in the transition rule.

The program is not strongly safe as all external atoms occur in cycles and their output is not bounded by ordinary atoms from outside. However, it is domain-expansion safe if we exploit semantical information. String splitting with $\&car$ yields ϵ or a shorter string, i.e., a well-ordering exists. Hence the output terms of $\&car$ are safe by Proposition 4.2 due to Definition 70 (i). Each transition step pushes a finite word onto the stack, and only finitely many steps happen (as no ϵ-transitions occur); hence only finitely many stack contents are possible, i.e., $\&concat$ has a finite output domain. Thus the output terms are safe due to Definition 70 (ii). The domain of $\&inc$ is finite for the same reason, which bounds $NewPos$. Hence, all variables are bounded and all attributes are domain-expansion safe.

Further Applications. We now briefly discuss other applications which exploit the concept of domain-expansion safety. Recursive processing of data structures, such as trees or lists, can easily violate traditional safety criteria. However, in a concrete program the use of the external sources may satisfy syntactic or semantic conditions such that finiteness of the grounding is still guaranteed. For instance, if a list is only subdivided but not recursively extended, then there exists a well-ordering as defined above and the grounding may be finite. Additional application-specific safety criteria can be easily integrated into our framework by customized term bounding functions. We will discuss a concrete application in detail in Chapter 5.

Another application is route planning, which we also discuss in Chapter 5 as a benchmark (the details of the encoding can be found in Appendix A). Importing the complete map a priori into the logic program is too expensive due to the large amount of data. The alternative is to query direct location between nodes in a recursive fashion. But if the set of nodes is not known in advance, then such queries do not satisfy traditional strong safety. However, as the map is finite our notion of domain-expansion safety, the existence of a finite grounding is guaranteed.

4.3 Grounding Algorithm for Liberally Domain-Expansion Safe HEX-Programs

In this section we present a grounding algorithm for liberally domain-expansion safe HEX-programs. It is based on the following idea. Iteratively ground the input program and then check if the grounding contains all relevant ground rules. The check works by evaluating external sources wrt. relevant interpretations and testing if they introduce any new values which were not respected in the grounding. If this is the case, then the set of constants is expanded and the program is grounded again. If the check does not identify additional constants which must be respected in the grounding, then it is guaranteed that the unrespected constants from $\mathcal{C}$ are irrelevant in order to ensure that the grounding has the same answer sets as the original program. For liberally domain-expansion safe programs, this procedure will eventually reach a fixpoint, i.e., all relevant constants are respected in the grounding.

4.3.1 Grounding Algorithm

We start with some concepts and define external atoms which are relevant for domain-expansion safety. Throughout the remainder assume that rules are standardized apart (i.e., have no variables in common). Let R be a set of external atoms and let r be a rule. By $r|_R$ we denote the rule obtained by removing all external atoms which are not in R, i.e., such that

$$H(r|_R) = H(r) \text{ and } B^s(r|_R) = \big(B^s(r) \cap A(r)\big) \cup \big(B^s(r) \cap R\big)$$

for $s \in \{+, -\}$. Similarly, $\Pi|_R = \bigcup_{r\in\Pi} r|_R$, for a program Π.

Definition 71 (Liberal Domain-Expansion Safety Relevance)**.** A set R of external atoms is *relevant for liberal de-safety* of a program Π, if $\Pi|_R$ is liberally de-safe and $var(r) = var(r|_R)$, for all $r \in \Pi$.

Note that for a program, the set of de-safe relevant external atoms is not necessarily unique, leaving room for heuristics. In the following we choose a specific set.

We further need the concepts of input auxiliary and external atom guessing rules. We say that an external atom $\&g[\mathbf{Y}](\mathbf{X})$ *joins* an atom b from the input list $\mathbf{Y}$ (output list $\mathbf{X}$), if some variable from $\mathbf{Y}$ ($\mathbf{X}$) occurs in b, where in case b is an external atom, the occurrence is in the output list of b. Note that this notion is slightly different from the one in previous work, as recapitulated as *basic input auxiliary rule* in Chapter 2. In particular, our new notion makes use of de-safety relevance.

Definition 72 (Input Auxiliary Rule)**.** Let Π be a HEX-program, and let $a = \&g[\mathbf{Y}](\mathbf{X})$ be some external atom with input list $\mathbf{Y}$ occurring in a rule $r \in \Pi$. Then, for each such atom, a rule r^a_{inp} is composed as follows:

- The head is $H\big(r^a_{inp}\big) = \big\{g^{\&g}_{inp}(\mathbf{Y})\big\}$, where $g^{\&g}_{inp}$ is a fresh predicate.
- The body $B\big(r^a_{inp}\big)$ contains each $b \in B^+(r) \setminus \{a\}$ such that a joins b in $\mathbf{Y}$, and b is de-safety relevant if it is an external atom.

The atom $g^{\&g}_{inp}(\mathbf{Y})$ in the head of such a rule is called *input (auxiliary) atom*.

Example 60. Consider the following non-ground HEX-program:

$$\Pi = \{out(Y) \leftarrow \&concat[a, b](X), \&concat[X, c](Y), limit(X), limit(Y)\}$$

Then the input auxiliary rule for $\&concat[a, b](X)$ is $g^{\&concat}_{inp}(a, b) \leftarrow$ (a fact), and that for $\&concat[X, c](Y)$ is $h^{\&concat}_{inp}(X, c) \leftarrow \&concat[a, b](X)$. □

Input auxiliary rules are used to derive all ground input tuples $\mathbf{y}$ with which the external atom needs to be evaluated. Next, we need *non-ground external atom guessing rules*. Note that this concept is different from *ground external atom guessing rules* used in Chapter 3, which always have an empty body. In this chapter we always mean non-ground external atom guessing rules and thus drop the prefix *non-ground*.

Definition 73 (Non-ground External Atom Guessing Rule)**.** Let Π be a HEX-program, and let $a = \&g[\mathbf{Y}](\mathbf{X})$ be some external atom. Then a rule r^a_{guess} is composed as follows:

- The head is $H\big(r^a_{guess}\big) = \big\{ e_{r,\&g[\mathbf{Y}]}(\mathbf{X}), ne_{r,\&g[\mathbf{Y}]}(\mathbf{X}) \big\}$.
- The body $B\big(r^a_{guess}\big)$ contains
  (i) each $b \in B^+(r) \setminus \{a\}$ such that a joins b in $\mathbf{Y}$ or $\mathbf{X}$, and b is de-safety relevant if it is an external atom; and
  (ii) $g^{\&g}_{inp}(\mathbf{Y})$.

It guesses the truth value of external atoms using a choice between the *external replacement atom* $e_{r,\&g[\mathbf{Y}]}(\mathbf{X})$, and fresh atom $ne_{r,\&g[\mathbf{Y}]}(\mathbf{X})$.

Example 61 (ctd.)**.** Consider the HEX-program Π from Example 60. Then the external atom guessing rule for $\&concat[a,b](X)$ is

$$e_{r,\&concat[a,b]}(X) \vee ne_{r,\&concat[a,b]}(X) \leftarrow g^{\&concat}_{inp}(a,b), limit(X)$$

and that for $\&concat[X,c](Y)$ is

$$e_{r,\&concat[X,c]}(Y) \vee ne_{r,\&concat[X,c]}(Y) \leftarrow h^{\&concat}_{inp}(X,c), limit(Y).$$

□

Our approach is based on a grounder for ordinary ASP programs. Compared to the naive grounding $grnd_C(\Pi)$, which substitutes all constants for all variables in all possible ways, we allow the ASP grounder GroundASP to optimize rules by eliminating rules if their body is always false, and ordinary body literals from the grounding if they are always true, as long as this does not change the answer sets.

Definition 74. We call rule r' an *o-strengthening* of r, if $H(r') = H(r)$, $B(r') \subseteq B(r)$ and $B(r) \setminus B(r')$ contains no external atoms and no external atom replacements.

Definition 75. An algorithm GroundASP is a *faithful ASP grounder* for a safe ordinary program Π, if it outputs an equivalent ground program Π' such that

- Π' consists of o-strengthenings of rules in $grnd_{C_\Pi}(\Pi)$;
- if $r \in grnd_{C_\Pi}(\Pi)$ has no o-strengthening in Π', then every answer set of $grnd_{C_\Pi}(\Pi)$ falsifies some ordinary literal in $B(r)$; and
- if $r \in grnd_{C_\Pi}(\Pi)$ has some o-strengthening $r' \in \Pi'$, then every answer set of $grnd_{C_\Pi}(\Pi)$ satisfies all default-literals in $B(r) \setminus B(r')$.

The formalization of the algorithm is shown in Algorithm GroundHEX. Our naming convention is as follows. Program Π is the non-ground input program. Program Π_p is the non-ground ordinary ASP *prototype program*, which is an iteratively updated variant of Π (with replacement atoms in place of external atoms) with additional rules. In each step, the *preliminary ground program* Π_{pg} is produced by grounding Π_p using a standard ASP grounding algorithm. Program Π_{pg} converges against a fixpoint from which the final *ground* HEX*-program* Π_g is extracted.

The algorithm first chooses a set of de-safety relevant external atoms, e.g., all external atoms as a naive and conservative approach, or following a greedy approach as in our implementation, and then introduces input auxiliary rules r^a_{inp} for every external atom $a = \&g[\mathbf{Y}](\mathbf{X})$ in a rule r in Π in Part (a). For all non-relevant external atoms, external atom guessing rules are introduced to ensure that the ground instances of the corresponding external replacement atoms are introduced in the grounding, even if they are not explicitly added. Then, all external atoms $\&g[\mathbf{Y}](\mathbf{X})$ in all rules r in Π_p are replaced by ordinary *replacement atoms* $e_{r,\&g[\mathbf{Y}]}(\mathbf{X})$. This allows the algorithm to use a faithful ASP grounder GroundASP in the main loop at (b). After the grounding step, the algorithm checks whether the grounding is large enough, i.e., if it contains all relevant constants. For this, it traverses all relevant external atoms at (c) and all relevant input assignments and tuples at (d) and at (e); $\mathbf{Y}_m$, $\mathbf{Y}_a$ and $\mathbf{Y}_n$ refer to the sublists of $\mathbf{Y}$ consisting of monotonic, antimonotonic and nonmonotonic input predicates, respectively. Then, constants returned by external sources are added to Π_p at (f); if the constants were already respected, then this will have no effect. Thereafter the main loop starts over again. The algorithm will find a program which respects all relevant constants. It then removes input auxiliary rules and translates replacement atoms to external atoms at (g).

We illustrate our grounding algorithm with the following example.

Example 62. Let Π be the following program:

$$\begin{aligned} f_1\colon\ d(a);\ f_2\colon\ d(b);\ f_3\colon\ d(c);\ r_1\colon\ & s(Y) \leftarrow \&diff[d,n](Y), d(Y) \\ r_2\colon\ & n(Y) \leftarrow \&diff[d,s](Y), d(Y) \\ r_3\colon\ & c(Z) \leftarrow \&count[s](Z) \end{aligned}$$

Here, $\&count[s](i)$ is true for the integer i corresponding to the number of elements in the extension of s. The program first partitions the domain (extension of d) into two sets (extensions of s and n) and then computes the size of s. The external atoms $\&diff[d,n](Y)$ and $\&diff[d,s](Y)$ are not relevant for de-safety. Thus, program Π_p at the beginning of the first iteration is as follows (neglecting input auxiliary rules, which are propositional facts in this example and are not needed for the further processing). Let $e_1(Y)$, $e_2(Y)$ and $e_3(Z)$ be shorthands for $e_{r_1,\&diff[d,n]}(Y)$, $e_{r_2,\&diff[d,s]}(Y)$, and $e_{r_3,\&count[s]}(Z)$, respectively.

$$\begin{aligned} f_1\colon\ d(a);\ f_2\colon\ d(b);\ f_3\colon\ d(c); \quad & r_1\colon\ s(Y) \leftarrow e_1(Y), d(Y) \\ g_1\colon\ e_1(Y) \vee ne_1(Y) \leftarrow d(Y); \quad & r_2\colon\ n(Y) \leftarrow e_2(Y), d(Y) \\ g_2\colon\ e_2(Y) \vee ne_2(Y) \leftarrow d(Y); \quad & r_3\colon\ c(Z) \leftarrow e_3(Z) \end{aligned}$$

The ground program Π_{pg} contains no instances of r_3 because the optimizer recognizes that $e_{r_3,\&count[s]}(Z)$ occurs in no rule head and no ground instance can be true in any answer set. Then the algorithm comes to the checking phase. It does not evaluate the external atoms in r_1 and

Algorithm GroundHEX

Input: A liberally de-safe HEX-program Π

Output: A ground HEX-program Π_g s.t. $\Pi_g \equiv^{pos} \Pi$

(a) Choose a set R of *de-safety relevant* external atoms in Π

$\Pi_p \leftarrow \Pi \cup \{r^a_{inp} \mid a = \&g[\mathbf{Y}](\mathbf{X}) \text{ in } r \in \Pi\}$

$\Pi_p \leftarrow \Pi_p \cup \{r^a_{guess} \mid a = \&g[\mathbf{Y}](\mathbf{X}) \text{ in } r \in \Pi, a \notin R\}$

Replace all external atoms $\&g[\mathbf{Y}](\mathbf{X})$ in all rules r in Π_p by $e_{r,\&g[\mathbf{Y}]}(\mathbf{X})$

(b) **repeat**

`// partial grounding`

$\Pi_{pg} \leftarrow \text{GroundASP}(\Pi_p)$

`// evaluate all de-safety relevant external atoms`

(c) **for** $a = \&g[\boldsymbol{Y}](\boldsymbol{X}) \in R$ *in a rule* $r \in \Pi$ **do**

Let $g^{\&g}_{inp}$ be the unique predicate in the head of r^a_{inp}

$\mathbf{A}_{ma} \leftarrow \{\mathbf{T}p(\mathbf{c}) \mid p(\mathbf{c}) \in A(\Pi_{pg}), p \in \mathbf{Y}_m\} \cup \{\mathbf{F}p(\mathbf{c}) \mid p(\mathbf{c}) \in A(\Pi_{pg}), p \in \mathbf{Y}_a\}$

`// do this wrt. all relevant assignments`

(d) **for** $\mathbf{A}_{nm} \subseteq \{\mathbf{T}p(\boldsymbol{c}), \mathbf{F}p(\boldsymbol{c}) \mid p(\boldsymbol{c}) \in A(\Pi_{pg}), p \in \boldsymbol{Y}_n\}$ *s.t.* $\nexists a : \mathbf{T}a, \mathbf{F}a \in \mathbf{A}_{nm}$ **do**

$\mathbf{A} \leftarrow (\mathbf{A}_{ma} \cup \mathbf{A}_{nm} \cup \{\mathbf{T}a \mid a \leftarrow . \in \Pi_{pg}\}) \setminus \{\mathbf{F}a \mid a \leftarrow . \in \Pi_{pg}\}$

(e) **for** $\boldsymbol{y} \in \{\boldsymbol{c} \mid g^{\&g}_{inp}(\boldsymbol{c}) \in A(\Pi_{pg})\}$ **do**

(f) $O \leftarrow \{\mathbf{x} \mid f_{\&g}(\mathbf{A}, \mathbf{y}, \mathbf{x}) = 1\}$

`// add the respective ground guessing rules`

$\Pi_p \leftarrow \Pi_p \cup \{e_{r,\&g[\mathbf{y}]}(\mathbf{x}) \vee ne_{r,\&g[\mathbf{y}]}(\mathbf{x}) \leftarrow . \mid \mathbf{x} \in O\}$

until Π_{pg} *did not change*

(g) $\Pi_g \leftarrow \Pi_{pg}$

Remove input auxiliary rules and external atom guessing rules from Π_g

Replace all $e_{\&g[\mathbf{y}]}(\mathbf{x})$ in Π_g by $\&g[\mathbf{y}](\mathbf{x})$

return Π_g

r_2, because they are not relevant for de-safety due to the domain predicate $d(Y)$. But it evaluates $\&count[s](Z)$ wrt. all $\mathbf{A} \subseteq \{s(a), s(b), s(c)\}$ because the external atom is nonmonotonic in s. Then the algorithm adds the rules $\{e_3(z) \vee ne_3(z) \leftarrow . \mid \ z \in \{0, 1, 2, 3\}\}$ to Π_p. After the second iteration, the algorithm terminates. □

4.3.2 Soundness and Completeness

One can show that Algorithm GroundHEXNaive is sound and complete. Towards a proof we first consider a computationally slower but conceptually simpler variant of the algorithm, for which we show these properties. Afterwards we prove that the optimizations in Algorithm GroundHEX do not harm soundness and completeness.

Compared to the naive Algorithm GroundHEXNaive, Algorithm GroundHEX contains the following modifications. The first change concerns the ordinary ASP grounder. We allow the

grounder to optimize the grounding as formalized by Definition 75, whereas Algorithm GroundHEXNaive uses the naive grounding $grnd_C(\Pi_p)$.

The second change concerns the external atoms. Intuitively, an external atom may be skipped if it can only return constants, which are guaranteed to appear also elsewhere in the grounding. Thus, Algorithm GroundHEX evaluates only de-safety relevant external atoms, whereas Algorithm GroundHEXNaive evaluates all of them.

The third optimization concerns the enumeration of assignments. Note that Step (c) in Algorithm GroundHEXNaive enumerates all models of Π_{pg}. That is, in order to ground the program, a solver must be called. This is computationally expensive and in fact not necessary. Step (d) in Algorithm GroundHEX simply enumerates assignments, which are directly extracted from the partial grounding, and which are constructed such that it is guaranteed that all relevant ground instances of the external atoms are represented in the grounding.

Algorithm GroundHEXNaive

Input: A liberally de-safe HEX-program Π
Output: A ground HEX-program Π_g s.t. $\mathcal{AS}(\Pi_g) \equiv^{pos} \mathcal{AS}(\Pi)$
(a) $\Pi_p \leftarrow \Pi \cup \{r^a_{inp} \mid a = \&g[\mathbf{Y}](\mathbf{X}) \text{ in } r \in \Pi\}$
Replace all external atoms $\&g[\mathbf{Y}](\mathbf{X})$ in all rules r in Π_p by $e_{r,\&g\mathbf{Y}}(\mathbf{X})$
(b) **repeat**
  `// partial grounding`
  $\Pi_{pg} \leftarrow grnd_C(\Pi_p)$ with constants C in Π_p
  `// check if the grounding is large enough`
(c) **for** *all models* $\mathbf{A}$ *of* Π_{pg} *over* $A(\Pi_{pg})$ **do**
    `// evaluate all external atoms`
(d) **for** $a = \&g[\boldsymbol{Y}](\boldsymbol{X})$ *in a rule* $r \in \Pi$ **do**
      Let $g^{\&g}_{inp}$ be the unique predicate in the head of r^a_{inp}
(e) **for** $\boldsymbol{y} \in \{\boldsymbol{c} \mid \mathbf{T}g^{\&g}_{inp}(\boldsymbol{c}) \in \mathbf{A}\}$ **do**
(f) $O \leftarrow \{\mathbf{x} \mid f_{\&g}(\mathbf{A}, \mathbf{y}, \mathbf{x}) = 1\}$
        `// add the respective ground guessing rules`
        $\Pi_p \leftarrow \Pi_p \cup \{e_{r,\&g[\mathbf{y}]}(\mathbf{x}) \vee ne_{r,\&g[\mathbf{y}]}(\mathbf{x}) \leftarrow . \mid \mathbf{x} \in O\}$
until Π_{pg} *did not change*
(g) $\Pi_g \leftarrow \Pi_{pg}$
Remove input auxiliary rules and external atom guessing rules from Π_g
Replace all $e_{r,\&g[\mathbf{y}]}(\mathbf{x})$ in Π_g by $\&g[\mathbf{y}](\mathbf{x})$
return Π_g

We now illustrate the algorithm with an example.

Example 63. Let

$$\Pi = \{d(x) \vee d(y);\ q(Y) \leftarrow d(X), \&concat[X, a](Y)\}$$

be the input program. In the first iteration we have

$$\Pi_p = \{d(x) \vee d(y);\ q(Y) \leftarrow d(X), e_{r,\&concat[X,a]}(Y); g^{\&g}_{inp}(X) \leftarrow d(X)\},$$

where $g^{\&concat}_{inp}$ is the unique input auxiliary predicate for $\&concat[X,a](Y)$. The grounding step yields

$$\begin{aligned}\Pi_{pg} \quad = \quad & \big\{d(x) \vee d(y)\big\} \cup \\ & \big\{q(c_2) \leftarrow d(c_1), e_{r,\&concat[c_1,a]}(c_2); g^{\&concat}_{inp}(c_1,a) \leftarrow d(c_1) \mid c_1, c_2 \in \{x,y\}\big\}.\end{aligned}$$

Now the algorithm comes to the checking phase at (c) and (d). Note that $g^{\&concat}_{inp}(x,a)$ and $g^{\&concat}_{inp}(y,a)$ appear in all models $\mathbf{A}$ of Π_{pg}. Therefore the algorithm will evaluate $\&concat$ with inputs (x,a) and (y,a) and collect all output tuples $\mathbf{x}$ s.t. $f_{\&g}(\mathbf{A},x,a,\mathbf{x}) = 1$ resp. $f_{\&g}(\mathbf{A},y,a,\mathbf{x}) = 1$ holds. This holds for the unary output tuples (xa) and (ya). Thus, Step (f) adds the rules

$$e_{r,\&g[x,a]}(xa) \vee ne_{r,\&g[x,a]}(xa) \leftarrow \text{ and } e_{r,\&g[y,a]}(xa) \vee ne_{r,\&g[y,a]}(ya) \leftarrow$$

to Π_p and grounding starts over again. In the next iteration,

$$q(xa) \leftarrow d(x), e_{r,\&concat[x,a]}(xa) \text{ and } q(ya) \leftarrow d(y), e_{r,\&concat[y,a]}(ya)$$

will appear in Π_{pg}. As no new atoms $g^{\&concat}_{inp}(\mathbf{y})$ appears in any of the models of the updated Π_{pg}, the loop terminates after the second iteration. □

Soundness and completeness of Algorithm GroundHEXNaive is formalized by the following proposition.

Proposition 4.6. *If* Π *is a liberally de-safe* HEX*-program, then* $\Pi \equiv^{pos}$ *GroundHEXNaive*(Π).

Proof. See Appendix B, page 220.

It can be shown that also the optimized algorithm is sound and complete.

Theorem 6 (Correctness of Algorithm GroundHEX)**.** *If* Π *is a liberally de-safe* HEX*-program, then GroundHEX*$(\Pi) \equiv^{pos} \Pi$.

Proof. See Appendix B, page 225.

4.4 Integration of the Algorithm into the Model-Building Framework

We are now ready to embed our grounding algorithm into the overall evaluation framework and get Algorithm BuildAnswerSetsGeneralized. For this, we first introduce an algorithm which computes the answer sets of a liberally de-safe HEX-program.

We first show that Algorithm EvaluateDomainExpansionSafe returns all answer sets of domain-expansion safe HEX-programs.

Algorithm EvaluateDomainExpansionSafe

Input: A liberally de-safe HEX-program Π, an input interpretation $\mathbf{A}$
Output: All answer sets of $\Pi \cup \{a \leftarrow . \mid \mathbf{T}a \in \mathbf{A}\}$ without $\mathbf{A}$
`// add input facts and ground the program`
$\Pi'_{grnd} \leftarrow \text{GuessAndCheckHexEvaluation}\big(\Pi \cup \{a \leftarrow . \mid \mathbf{T}a \in \mathbf{A}\}\big)$
`// ground program evaluation and output projection`
return $\big\{\mathbf{A}' \setminus \big(\mathbf{A} \cup \{\mathbf{F}a \in \mathbf{A}'\}\big) \mid \mathbf{A}' \in \text{EvalGroundHexProgram}(\Pi'_{grnd})\big\}$

Proposition 4.7. *Given a domain-expansion safe* HEX*-program* Π *and an input assignment* $\mathbf{A}$*, Algorithm EvaluateDomainExpansionSafe returns*

$$\big\{\mathbf{A}' \setminus \big(\mathbf{A} \cup \{\mathbf{F}a \in \mathbf{A}'\}\big) \mid \mathbf{A}' \in \mathcal{AS}\big(\Pi \cup \{a \leftarrow . \mid \mathbf{T}a \in \mathbf{A}\}\big)\big\},$$

i.e., the positive parts of all answer sets of Π *augmented with the positive atoms in* $\mathbf{A}$*.*

Proof. The proposition follows from Theorem 6, which shows that the grounding Π'_{grnd} has the same answer sets as Π (if restricted to their positive parts), and from the soundness and completeness of the evaluation algorithms for ground HEX-programs introduced in Chapter 3. □

We now replace Algorithm EvaluateExtendedPreGroundable in Algorithm BuildAnswerSets by Algorithm EvaluateDomainExpansionSafe which computes the answer sets of the single units. However, the formal incorporation of our algorithms into the framework described by Schüller (2012) and recapitulated in Section 4.1 is nontrivial, because two of the fundamental definitions of the framework are that of an *evaluation unit* and of an *evaluation graph*, which use extended pre-groundable HEX-programs (cf. Definition 25) as units. Our goal is to support the generalized class of liberally domain-expansion safe programs as units. Because of Theorem 4, which proves soundness and completeness of Algorithm BuildAnswerSets, and many intermediate results of Schüller (2012) depend (transitively) on those basic definitions, they do not immediately carry over to a generalized notion of evaluation units. However, a look into the proofs by Schüller (2012) reveals that there is in fact only one proposition (Proposition 13) which directly makes use of the property that evaluation units are extended pre-groundable. We will introduce and prove an equivalent proposition for our generalized class of programs. Then all other results still hold.

We have already shown that it is possible to finitely ground and evaluate domain-expansion safe programs, i.e., the new algorithms work correctly within single evaluation units. However, it remains to show that this is still compatible with the model-building framework introduced in Chapter 2. In particular, we need to show that Theorem 4 still holds if evaluation units are not necessarily extended pre-groundable but liberally domain-expansion safe HEX-programs. To this end, we introduce a generalized notion of evaluation units and evaluation graphs (cf. Definitions 51 and 52).

Definition 76 (Generalized (Evaluation) Unit)**.** A *generalized (evaluation) unit* is a liberally domain-expansion safe HEX-program.

Algorithm BuildAnswerSetsGeneralized

Input: Generalized evaluation graph $\mathcal{E} = (V, E)$ for a HEX-program Π with a unit u_{final} that depends on all other units in V

Output: All answer sets of Π

$M = \emptyset, F = \emptyset, unit = \emptyset, type = \emptyset, int = \emptyset, U = V$

while $U \neq \emptyset$ **do**

  Choose $u \in U$ s.t. $preds_{\mathcal{E}}(u) \cap U = \emptyset$

  Let $\{u_1, \ldots, u_k\} = preds_{\mathcal{E}}(u)$

(a) **if** $k = 0$ **then**

    $m \leftarrow max(M) + 1$

    $M \leftarrow M \cup \{m\}$

    $unit(m) \leftarrow u, type(m) \leftarrow \text{I}, int(m) \leftarrow \emptyset$

(b) **else**

    for $m_1 \in o\text{-}ints(u_1), \ldots, m_k \in o\text{-}ints(u_k)$ **do**

      if $J = m_1 \bowtie \cdots \bowtie m_k$ *is defined* **then**

        $m \leftarrow max(M) + 1$

        $M \leftarrow M \cup \{m\}$

        $F \leftarrow F \cup \big\{(m, m_i) \mid 1 \leq i \leq k\big\}$

        $unit(m) \leftarrow u, type(m) \leftarrow \text{I}, int(m) \leftarrow J$

(c) **if** $u = u_{final}$ **then**

    return $i\text{-}ints(u_{final})$

(d) **for** $m' \in i\text{-}ints(u)$ **do**

    $O \leftarrow \text{EvaluateDomainExpansionSafe}\big(u, int(m')\big)$

    for $o \in O$ **do**

      $m \leftarrow max(M) + 1$

      $M \leftarrow M \cup \{m\}$

      $F \leftarrow F \cup \big\{(m, m') \mid 1 \leq i \leq k\big\}$

      $unit(m) \leftarrow u, type(m) \leftarrow \text{O}, int(m) \leftarrow o$

  $U \leftarrow U \setminus \{u\}$

Definition 77 (Generalized Evaluation Graph)**.** A *generalized evaluation graph* $\mathcal{E} = \langle V, E \rangle$ of a program Π is a directed acyclic graph; vertices V are generalized evaluation units and $\mathcal{E}$ has the following properties:

(a) $\bigcup_{u \in V} u = \Pi$, i.e., every rule $r \in \Pi$ is contained in at least one unit;

(b) for every non-constraint $r \in \Pi$, it holds that $\big|\{u \in V \mid r \in u\}\big| = 1$, i.e., r is contained in exactly one unit;

(c) for each nonmonotonic dependency $r \to_n s$ between rules $r, s \in \Pi$ and for all $u \in V$ with $r \in u$ and $v \in V$ with $s \in v$ s.t. $u \neq v$, there exists an edge $(u, v) \in E$, i.e., nonmonotonic dependencies between rules have corresponding edges everywhere in $\mathcal{E}$; and

(d) for each monotonic dependency $r \to_m s$ between rules $r, s \in \Pi$, there exists one $u \in V$ with $r \in u$ such that E contains all edges (u, v) with $v \in V$, $s \in v$ and $v \neq u$, i.e., there is (at least) one unit in $\mathcal{E}$ where all monotonic dependencies from r to other rules have corresponding outgoing edges in $\mathcal{E}$.

Example 64. Graph $\mathcal{E}$ from Example 45 is an evaluation graph and also a generalized evaluation graph of program Π. Another generalized evaluation graph, which is not an evaluation graph, is $\mathcal{E}' = \big\langle \{u_1 = \Pi, u_{final}\}, \{(u_{final}, u_1)\} \big\rangle$. □

We show now that for a generalized evaluation graph $\mathcal{E} = (V, E)$, BuildAnswerSetsGeneralized still returns $\mathcal{AS}(\Pi)$.

Theorem 7 (Soundness and Completeness of Algorithm BuildAnswerSetsGeneralized)**.** *Algorithm BuildAnswerSetsGeneralized applied to a generalized evaluation graph* $\mathcal{E} = (V, E)$ *of a* HEX*-program* Π *returns* $\mathcal{AS}(\Pi)$.

Proof. The proposition corresponds to Theorem 4, which is Theorem 15 by Schüller (2012), but with generalized evaluation units in place of evaluation units, i.e., units may be domain-expansion safe programs which are not extended pre-groundable.

The proofs by Schüller (2012) on which Theorem 15 depends in fact make use of pre-groundability only in a single part. This is in Proposition 13, which states that for an extended pre-groundable HEX-program Π and an input interpretation $\mathbf{A}$, Algorithm EvaluateExtended-PreGroundable returns $\big\{\mathbf{A}' \setminus (\mathbf{A} \cup \{\mathbf{F}a \in \mathbf{A}'\}) \mid \mathbf{A}' \in \mathcal{AS}(\Pi \cup \{a \leftarrow . \mid \mathbf{T}a \in \mathbf{A}\})\big\}$, i.e., the positive parts of all answer sets of Π augmented with $\mathbf{A}$.

However, we have shown in Proposition 4.7 that Algorithm EvaluateDomainExpansionSafe behaves exactly like this for domain-expansion safe HEX-programs. Because the remaining parts of the proofs by Schüller (2012) do not make use of the property of extended pre-groundability, Theorem 15 goes through also for domain-expansion safe programs if Algorithm EvaluateExtendedPreGroundable is replaced by Algorithm EvaluateDomainExpansionSafe. □

4.5 Greedy Evaluation Heuristics

The motivation for the evaluation framework introduced by Eiter et al. (2011a) and described in more detail by Schüller (2012) was performance enhancement. However, not every strongly safe program is extended pre-groundable; thus program decomposition is in some cases *indispensable* for program evaluation. This is in contrast to the grounding algorithm introduced above, which can directly ground any liberally de-safe, and thus strongly safe, program.

Example 65. Program Π from Example 62 cannot be grounded by the traditional HEX algorithms as it is not extended pre-groundable. Instead, it needs to be partitioned into two units $u_1 = \{f_1, f_2, f_3, r_1, r_2\}$ and $u_2 = \{r_3\}$ with $u_1 \to_n u_2$. Now u_1 and u_2 are extended pre-groundable HEX-programs. Then the answer sets of u_1 must be computed before u_2 can be grounded. Our algorithm can ground the whole program immediately. □

Therefore, in contrast to the previous algorithms one can keep the whole program as a single unit, but also still apply decomposition with liberally de-safe programs as units. While program decomposition led to performance increase for the traditional solving algorithms, it is counterproductive for new learning-based algorithms because learned knowledge cannot be effectively reused. In guess-and-check ASP programs, existing heuristics for the generation of the evaluation graph frequently even split the guessing from the checking part, which is derogatory to the learning. Thus, from this perspective is advantageous to have few units. However, for the grounding algorithm a worst case is that a unit contains an external atom that is relevant for de-safety and receives nonmonotonic input from the same unit. In this case it needs to consider exponentially many assignments.

Example 66. Reconsider program Π from Example 62. The algorithm evaluates $\&count[s](Z)$ wrt. all $\mathbf{A} \subseteq \{s(a), s(b), s(c)\}$ because it is nonmonotonic and de-safety relevant. Now assume that the program contains the additional constraint

$$c_1\colon \quad \leftarrow s(X), s(Y), s(Z), X \neq Y, X \neq Z, Y \neq Z,$$

i.e., no more than two elements can be in set s. Then the algorithm would still check all $\mathbf{A} \subseteq \{s(a), s(b), s(c)\}$, but it is clear that the subset with three elements, which introduces the constant 3, is irrelevant because this interpretation will never occur in an answer set. If the program is split into units $u_1 = \{f, r_1, r_2, c_1\}$ and $u_2 = \{r_3\}$ with $u_2 \to_n u_1$, then $\{s(a), s(b), s(c)\}$ does not occur as an answer set of u_1. Thus, u_2 never receives this interpretation as input and never is evaluated wrt. this interpretation. □

Algorithm GroundHEX evaluates the external sources wrt. all interpretations such that the set of observed constants is maximized. While monotonic and antimonotonic input atoms are not problematic (the algorithm can simply set all to true resp. false), nonmonotonic parameters require an exponential number of evaluations. Thus, in such cases program decomposition is still useful as it restricts grounding to those interpretations which are actually relevant in some answer set. Program decomposition can be seen as a hybrid between traditional and lazy grounding (cf. e.g. Palù et al. (2009)), as program parts are instantiated which are larger than single rules but smaller than the whole program.

We thus introduce a heuristics in Algorithm GreedyGEG for generating a good generalized evaluation graph, which iteratively merges units. Condition (d) maintains acyclicity, while the condition at (e) deals with two opposing goals: (1) minimizing the number of units, and (2) splitting the program whenever a de-relevant nonmonotonic external atom would receive input from the same unit. It greedily gives preference to (1).

Algorithm GreedyGEG

Input: A liberally de-safe HEX-program Π
Output: A generalized evaluation graph $\mathcal{E} = \langle V, E \rangle$ for Π
(a) $G \leftarrow \langle \Pi, \rightarrow_m \cup \rightarrow_n \rangle$
Let V be the set of (subset-maximal) strongly connected components of G
Update E
(b) **while** V *was modified* **do**
(c) **for** $u_1, u_2 \in V$ *such that* $u_1 \neq u_2$ **do**
(d) **if** *there is no indirect path from* u_1 *to* u_2 *(via some* $u' \neq u_1, u_2$*) or vice versa* **then**
(e) **if** *no de-relevant &*$g[\boldsymbol{y}](\boldsymbol{x})$ *in some* u_2 *has a nonmonotonic predicate input from* u_1 **then**
$V \leftarrow \big(V \setminus \{u_1, u_2\}\big) \cup \{u_1 \cup u_2\}$
Update E
return $\mathcal{E} = \langle V, E \rangle$

We illustrate the heuristics with an example.

Example 67. Reconsider program Π from Examples 62 and 66. Then Algorithm GreedyGEG creates a generalized evaluation graph with the two units $u_1 = \{f_1, f_2, f_3, r_1, r_2, c_1\}$ and $u_2 = \{r_3\}$ with $u_2 \rightarrow_n u_1$, which is as desired. □

It is not difficult to show that the heuristics yields a sound result.

Proposition 4.8. *For a liberally de-safe program* Π*, Algorithm GreedyGEG returns a suitable generalized evaluation graph of* Π*.*

Proof. The initial set of nodes defined at (a) is the set of all subset-maximal strongly connected components of the rules of Π wrt. $\rightarrow_m \cup \rightarrow_n$. This ensures that the graph is acyclic, that every rule (including constraints) is contained in exactly one unit, and that unit dependencies are updated according to the rule dependencies. Thus the initial decomposition forms a generalized evaluation graph.

Loop (b) then iteratively merges two different units, where Condition (d) ensures that the graph remains acyclic. As the algorithm also updates E according to the rule dependencies, all conditions of a generalized evaluation graph remain satisfied. □

4.6 Related Work and Summary

In this section, we discuss some other notions of safety from the literature and discuss their relationship to liberal domain-expansion safety. We will establish that our concept is strictly more general than many other notions of safety. In particular, we formally compare our notion to strong safety, VI-restricted programs and logic programs with functions symbols. Afterwards we summarize the chapter and give an outlook on future work.

4.6.1 Related Work

Our notion of liberal domain-expansion safety using b_{synsem} compares to the traditionally used strong domain-expansion safety and to other formalizations.

Strong Safety. We have defined strong safety in Definitions 22 and 23. One can show that (liberal) domain-expansion safety is strictly less restrictive.

Theorem 8. *Every strongly domain-expansion safe program* Π *is domain-expansion safe.*

Proof. Suppose Π is strongly safe. We show that for any attribute α of Π, we have $a \in S_n(\Pi)$ for some $n \geq 0$, i.e., a is domain-expansion safe.

Let a be an attribute of Π and let j be the number of malign cycles wrt. $\emptyset$ in $G_A(\Pi)$ from which a is reachable. We prove by induction that if a is reachable from $j \geq 0$ malign cycles wrt. $\emptyset$ in $G_A(\Pi)$, then a is domain-expansion safe.

If $j = 0$ we make a case distinction. Case 1: if a is of form $p{\restriction}i$, then there is no information flow from a malign cycle wrt. $\emptyset$ to $p{\restriction}i$. Therefore, for every rule r with $p(t_1, \ldots, t_\ell) \in H(r)$ we have that $t_i \in B_{n+1}(r, \Pi, b_{synsem})$ for all $n \geq 0$ due to Condition (i) in Definition 70. But then $p{\restriction}i$ is domain-expansion safe.

Case 2: if a is of form $\&g[\mathbf{Y}]_r{\restriction}_{\mathrm{I}}i$, then for every variable $Y_i \in \mathbf{Y}$ with $type(\&g, i) = \mathbf{const}$ we have $Y_i \in B_{n+1}(r, \Pi, b_{synsem})$ due to Condition (i) in Definition 70, and for every predicate $p_i \in \mathbf{Y}$ with $type(\&g, i) = \mathbf{pred}$ we have that $p_i{\restriction}j$ is domain-expansion safe for every $1 \leq j \leq ar(p_i)$ by Case 1; note that $p_i{\restriction}j$ is not reachable from any malign cycle wrt. $\emptyset$ because this would by transitivity of reachability mean that also $\&g[\mathbf{Y}]_r{\restriction}_{\mathrm{I}}i$ is reachable from such a cycle, which contradicts our assumption. But then also $\&g[\mathbf{Y}]_r{\restriction}_{\mathrm{I}}i$ is domain-expansion safe by Definition 63.

Case 3: if a is of form $\&g[\mathbf{Y}]_r{\restriction}_{\mathrm{O}}i$, then no $\&g[\mathbf{Y}]_r{\restriction}_{\mathrm{I}}j$ for $1 \leq j \leq ar_{\mathrm{I}}(\&g)$ is reachable from a malign cycle wrt. $\emptyset$, because then also $\&g[\mathbf{Y}]_r{\restriction}_{\mathrm{O}}i$ would be reachable from such a cycle. But then by Definition 63, $\&g[\mathbf{Y}]_r{\restriction}_{\mathrm{O}}i$ is domain-expansion safe. Hence, attributes of any kind, which are not reachable from malign cycles wrt. $\emptyset$, are domain-expansion safe.

Induction step $j \mapsto j + 1$: If a is reachable from $j + 1$ malign cycles wrt. $\emptyset$, then there is an attribute α' in such a cycle C from which a is reachable. The malign cycle C wrt. $\emptyset$ contains an attribute of kind $\&g[\mathbf{Y}]_r{\restriction}_{\mathrm{O}}i$, corresponding to an external atom $\&g[\mathbf{Y}](\mathbf{X})$ in rule r. Since $\&g[\mathbf{Y}]_r{\restriction}_{\mathrm{O}}i$ is cyclic in $G_A(\Pi)$, $\&g[\mathbf{Y}](\mathbf{X})$ is cyclic in $ADG(\Pi)$. Then by strong safety of Π, each variable in Y occurs in a body atom $p(t_1, \ldots, t_\ell) \in B^+(r)$ which is not part of C, i.e., it is captured by $p{\restriction}k$ for some $1 \leq k \leq ar(p)$. But since $p(t_1, \ldots, t_\ell)$ is not part of the cycle C in $ADG(\Pi)$, also $p{\restriction}k$ is not part of it. Therefore $p{\restriction}k$ is reachable from (at least) one malign

cycle wrt. $\emptyset$ less than a, i.e., it is reachable from at most j malign cycles. Thus $p{\restriction}k$ is domain-expansion safe by induction hypothesis. But then by Condition (ii) in Definition 64, also a is domain-expansion safe. □

The converse does not hold, as there are domain-expansion safe programs that are not strongly safe, cf. Example 49.

VI-Restricted Programs. The notion of *VI-restrictedness* for *VI programs* was introduced by Calimeri et al. (2007) and amounts to the class of HEX-programs in which all input parameters to external atoms are of type $\mathbf{const}$. More formally:

Definition 78 (VI-Programs)**.** A *VI-program* is a HEX-program Π such that for every external atom $\&g[\mathbf{X}](\mathbf{Y})$ in Π we have $type(\&g, i) = \mathbf{const}$ for all $1 \leq i \leq ar_{\text{I}}(\&g)$.

The notion of attribute dependency graph by Calimeri et al. (2007) is related to our notion of $ADG(\Pi)$, which is more fine-grained for attributes of external predicates. While we use a separate node $\&g[\mathbf{Y}]_r{\restriction}_T i$ for each external predicate $\&g$ with input list $\mathbf{Y}$ in a rule r and $T \in \{\text{I}, \text{O}\}$ for all $1 \leq i \leq ar_T(\&g)$, Calimeri et al. (2007) use just one attribute $\&g{\restriction}i$ for each $i \in \{1, \ldots, ar_{\text{I}}(\&g){+}ar_{\text{O}}(\&g)\}$ independent of $\mathbf{Y}$. Thus, neither multiple occurrences of $\&g$ with different input lists in a rule, nor of the same attribute in multiple rules are distinguished; this collapses distinct nodes in our attribute dependency graph into one. We call the graph $G_{\bar{A}}(\Pi)$, which possibly contains (spurious) cycles not visible in $G_A(\Pi)$.

Example 68. Consider the program

$$\Pi = \{r_1\colon\ t(X) \leftarrow s(Y), \&e[Y](X);\ r_2\colon\ r(X) \leftarrow t(Y), \&e[Y](X)\}.$$

The attributes are $s{\restriction}1$, $t{\restriction}1$, $r{\restriction}1$, $\&e[Y]_{r_1}{\restriction}_{\text{I}}1$, $\&e[Y]_{r_1}{\restriction}_{\text{O}}1$, $\&e[Y]_{r_2}{\restriction}_{\text{I}}1$ and $\&e[Y]_{r_2}{\restriction}_{\text{O}}1$.
We get the following edges from the first rule:

$$(s{\restriction}1, \&e[Y]_{r_1}{\restriction}_{\text{I}}1),(\&e[Y]_{r_1}{\restriction}_{\text{I}}1, \&e[Y]_{r_1}{\restriction}_{\text{O}}1) \text{ and } (\&e[Y]_{r_1}{\restriction}_{\text{O}}1, t{\restriction}1)$$

We get the following edges from the second rule:

$$(t{\restriction}1, \&e[Y]_{r_2}{\restriction}_{\text{I}}1),(\&e[Y]_{r_2}{\restriction}_{\text{I}}1, \&e[Y]_{r_2}{\restriction}_{\text{O}}1) \text{ and } (\&e[Y]_{r_2}{\restriction}_{\text{O}}1, r{\restriction}1)$$

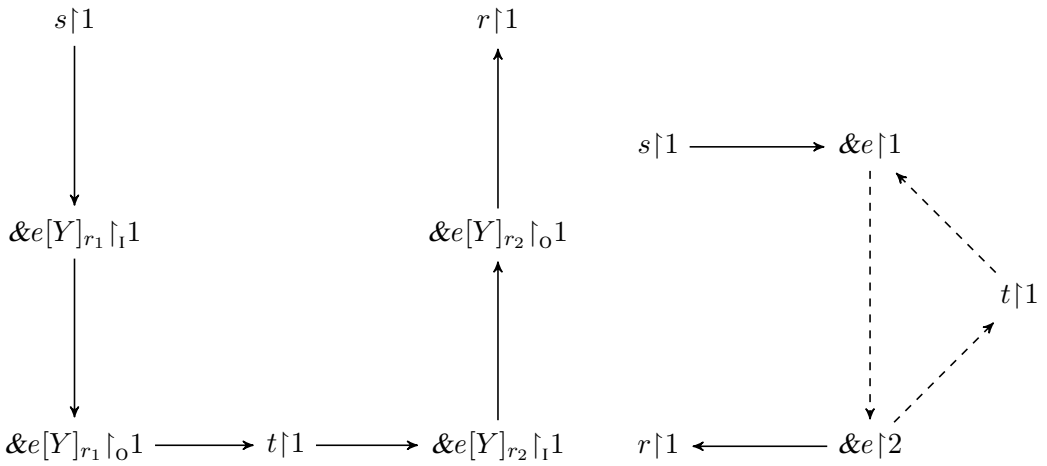
In contrast, Calimeri et al. (2007) have the attributes $s{\restriction}1$, $t{\restriction}1$, $r{\restriction}1$, $\&e{\restriction}1$ and $\&e{\restriction}2$ with the following edges:

$$(s{\restriction}1, \&e{\restriction}1),(\&e{\restriction}1, \&e{\restriction}2),(\&e{\restriction}2, t{\restriction}1),(t{\restriction}1, \&e{\restriction}1) \text{ and } (\&e{\restriction}2, r{\restriction}1)$$

The graphs $G_A(\Pi)$ and $G_{\bar{A}}(\Pi)$ are visualized in Figure 4.5. □

Towards a definition of the class of *VI-restricted programs* we use the following notions:

- A rule r *poisons* an attribute $p{\restriction}i$, if $p(t_1, \ldots, t_\ell) \in H(r)$ and t_i is a variable.



(a) $G_A(\Pi)$ without Cycles (b) $G_{\bar{A}}(\Pi)$ with a Cycle (dashed)

Figure 4.5: Visualization of the Program Π from Example 68

- A rule r is *dangerous*, if it poisons an attribute $p\restriction i$ which is in a cycle in $G_{\bar{A}}(\Pi)$; $p\restriction i$ is called *dangerous attribute in* r.

The sets of dangerous and of savior attributes are defined in a mutually recursive fashion as the least sets satisfying the following conditions:

- If r is dangerous, then a dangerous attribute $p\restriction i$ capturing X is *blocked in* r, if for every $\&g[\mathbf{Y}](\mathbf{X})$ with $X \in \mathbf{X}$, it holds that for every variable $Y \in \mathbf{Y}$ there is a body atom $q(t_1, \ldots, t_\ell) \in B^+(r)$ such that $X = t_i$ for some $1 \leq i \leq ar(q)$ and attribute $q\restriction i$ is *savior*.

- An attribute $p\restriction i$ is *savior*, if for every rule $r \in \Pi$ with $p(t_1, \ldots, t_\ell) \in H(r)$:
  - t_i is a constant; or
  - there is some ordinary atom $q(s_1, \ldots, s_{ar(q)}) \in B^+(r)$ such that $t_i = s_j$ for some $1 \leq j \leq ar(q)$ and $q\restriction j$ is savior; or
  - $p\restriction i$ is blocked in r.

We now introduce a class of VI-programs as follows.

Definition 79 (VI-restricted Programs)**.** A rule $r \in \Pi$ is *VI-restricted*, if all its dangerous attributes are blocked; a program Π is *VI-restricted*, if all its dangerous rules are VI-restricted.

Using b_{synsem}, we can show:

Theorem 9. *Every VI-restricted program* Π *is domain-expansion safe.*

Proof. We first reformulate the definitions of *blocking* and *savior attributes* in an inductive way, which is possible because criteria are monotonic.

Blocking:

- $blocked_0(r) = \emptyset$ for all $r \in \Pi$
- $blocked_{n+1}(r) = \big\{p{\restriction}i \mid p{\restriction}i$ is dangerous in r and $p{\restriction}i$ captures X in r and
  for every $\&g[\mathbf{Y}](\mathbf{X})$ with $X \in \mathbf{X}$,
  for every variable $Y \in \mathbf{Y}$ there is a body atom $q(t_1, \ldots, t_\ell)$
  s.t. $X = t_i$ for some $1 \leq i \leq ar(q)$ and $q{\restriction}i \in savior_n\big\}$,
  for all $n \geq 0$
- $blocked_\infty(r) = \bigcup_{n \geq 0} blocked_n(r)$

Savior attributes:

- $savior_0 = \emptyset$
- $savior_{n+1} = \big\{p{\restriction}i \mid$ for all $r \in \Pi$ with $p(t_1, \ldots, t_\ell) \in H(r)$, either
  t_i is a constant; or
  t_i is captured by some $q{\restriction}j \in savior_n$ in $B^+(r)$; or
  $p{\restriction}i \in blocked_n(r)\big\}$,
  for all $n \geq 0$
- $savior_\infty = \bigcup_{n \geq 0} savior_n$

We show now by induction on n for all $n \geq 0$:

- If $p{\restriction}i \in blocked_n(r)$ and $p{\restriction}i$ captures variable X in r, then $X \in B_n(r, \Pi, S, b_{synsem})$.
- If $p{\restriction}i \in savior_n$ for some $n \geq 0$, then $p{\restriction}i \in S_n(\Pi)$.

For $n = 0$ this is trivial.

For the induction step $n \mapsto n + 1$, suppose $p{\restriction}i \in blocked_{n+1}(r)$. Then $p{\restriction}i$ is dangerous and captures some X in r. For every $\&g[\mathbf{Y}](\mathbf{X})$ with $X \in \mathbf{X}$ and for every variable $Y \in \mathbf{Y}$ there is a body atom $q(t_1, \ldots, t_\ell)$ such that $X = t_j$ for some $1 \leq j \leq ar(q)$ and $q{\restriction}j \in savior_n$ Then, by the induction hypothesis, $q{\restriction}j$ is domain-expansion safe. But then by Condition (ii) in Definition 64 all input variables $Y \in \mathbf{Y}$ are declared bounded in the first step, i.e., $Y \in B_{n+1,1}(r, \Pi, b_{synsem})$. Then by Condition (iii) in Definition 64 also all output variables $X \in \mathbf{X}$ are declared bounded in the second step, i.e., $X \in B_{n+1,2}(r, \Pi, b_{synsem})$. Thus we have $X \in B_{n+1}(r, \Pi, S_n(\Pi), b_{synsem})$.

Now suppose $p{\restriction}i$ in $savior_{n+1}$. Then we have for every rule $r \in \Pi$ with $p(t_1, \ldots, t_\ell) \in H(r)$ that

(i) t_i is a constant; or

(ii) t_i is captured by some $q{\restriction}j \in savior_n$ in $B^+(r)$; or

(iii) $p{\restriction}i \in \mathit{blocked}_n(r)$.

In Case (i), $t_i \in B_{n+1}(r, \Pi, S_n(\Pi), b_{synsem})$ by Condition (i) in Definition 64. In Case (ii), $q{\restriction}j$ is domain-expansion safe by the induction hypothesis and thus t_i is declared bounded by Condition (ii) in Definition 64. In Case (iii), $t_i \in B_{n+1}(r, \Pi, S, b_{synsem})$ as shown above.

This shows that all dangerous (but blocked) attributes are domain-expansion safe. It remains to show that also all non-dangerous attributes are domain-expansion safe. Let a be such an attribute. If it occurs in a cycle in $G_A(\Pi)$, then it occurs also in a cycle in $G_{\bar{A}}(\Pi)$ because in this graph nodes from $G_A(\Pi)$ may be merged, i.e., the graph is less fine-grained. If it is of type $p{\restriction}i$, then it is dangerous and we already know that it is domain-expansion safe. Otherwise it is an external input attribute of form $\&g[\mathbf{X}]_r{\restriction}_\mathrm{I}i$ or output attribute of form $\&g[\mathbf{X}]_r{\restriction}_\mathrm{O}i$. If it is an input attribute, then we know that its cyclic input depends (possibly transitively) on domain-expansion safe ordinary attributes. As the output attributes of external atoms become domain-expansion safe as soon as the input becomes domain-expansion safe by Definition 63, domain-expansion safety will be propagated by Condition (iii) in Definition 64 along the cycle, beginning at the ordinary predicates, i.e., the input parameter will be declared domain-expansion safe after finitely many steps (since the cycle is of finite length). This shows that all attributes in cycles in $G_A(\Pi)$ are domain-expansion safe.

As all attributes in cycles are domain-expansion safe, the remaining attributes (attributes which depend on a cycle but are not in a cycle) will also be declared domain-expansion safe after finitely many steps by Definition 63. □

The converse does not hold, as there are domain-expansion safe VI-programs (e.g. due to semantic criteria) that are not VI-restricted.

Example 69. Consider the program $\Pi = \{p(Y) \leftarrow p(X), \&le[X](Y)\}$ where $f_{\&le}(\mathbf{A}, x, y) = 1$ iff$_{\text{def}}$ $0 \leq y \leq x$, for all assignments $\mathbf{A}$. Then Π is not VI-restricted because the attribute $p{\restriction}1$ appears in a cycle in $G_{\bar{A}}(\Pi)$ and is thus dangerous and not blocked because X does not occur in a savior body atom. However, the program is domain-expansion safe by Condition (i) in Definition 70 using the well-ordering $\leq$. □

This shows that our notion of domain-expansion safety is strictly more liberal than VI-restrictedness.

Logic Programs with Function Symbols. Another related notion is that of *ω-restricted logic programs* by Syrjänen (2001), which allow function symbols under a level mapping to control the introduction of new terms with function symbols to ensure decidability.

In this paragraph we assume that a program is a set of rules of form

$$a \leftarrow b_1, \ldots, b_m, \mathrm{not}\, b_{m+1}, \ldots, \mathrm{not}\, b_n,$$

with $k + n > 0$ where each a_i for $1 \leq i \leq k$ is an atom $p(t_1, \ldots, t_\ell)$ with function terms t_j, $1 \leq j \leq \ell$, and each b_i for $1 \leq i \leq n$ is a classical atom.

The notion of ω-restricted logic programs hinges on the concept of predicate dependencies.

Definition 80. For a program Π, the predicate dependencies are defined as follows.

- Sets $P_1^+(\Pi)$ resp. $P_1^-(\Pi)$ are the least sets such that for all $p_1, p_2 \in \mathcal{P}$ it holds that $(p_1, p_2) \in P_1^+(\Pi)$ resp. $(p_1, p_2) \in P_1^-(\Pi)$ whenever p_1 occurs in an atom in $H(r)$ and p_2 occurs in an atom in $B^+(r)$ resp. $B^-(r)$ for some $r \in \Pi$.
- Set $P^+(\Pi)$ is the transitive closure of $P_1^+(\Pi) \cup P_1^-(\Pi)$.
- For all $p_1, p_n \in \mathcal{P}$ it holds that $(p_1, p_n) \in P^-(\Pi)$ if there is a sequence $\langle p_1, p_2, \ldots, p_n \rangle$ with $(p_i, p_{i+1}) \in P_1^+(\Pi) \cup P_1^-(\Pi)$ for all $1 \leq j < n$ and $(p_j, p_{j+1}) \in P_1^-(\Pi)$ for some $1 \leq j < n$.

Intuitively, a predicate p_1 depends positively on a predicate p_2, if there is a derivation path from an atom over p_2 to an atom over p_1 in the program. It depends negatively on p_2 if at least one derivation step in such a path uses default-negation.

We then recall the definition of ω-restricted logic programs as follows.

Definition 81. An ω-stratification of a program Π is a function $s\colon\ \mathcal{P} \to \mathbb{N} \cup \{\omega\}$ such that

- for all $p_1, p_2 \in \mathcal{P}$, if $(p_1, p_2) \in P^+(\Pi)$ then $s(p_1) \geq s(p_2)$; and
- for all $p_1, p_2 \in \mathcal{P}$, if $(p_1, p_2) \in P^-(\Pi)$ then $s(p_1) > s(p_2)$ or $s(p_1) = \omega$.

By convention, $\omega > n$ for all $n \in \mathbb{N}$.

For a rule $r \in \Pi$ with $p(\mathbf{t}) \in H(r)$ and an ω-stratification, let

$$\Omega(r, s) = s(p)$$

and

$$\Omega(v, r, s) = min\left(\left\{s(q) \mid q(\mathbf{t}') \in B^+(r) \text{ and } v \in var(q(\mathbf{t}'))\right\} \cup \{\omega\}\right).$$

A program Π is *ω-restricted* if it holds for all $r \in \Pi$ that

$$\text{for all } v \in var(r) \text{ we have } \Omega(v, r, s) < \Omega(r, s).$$

Intuitively, if a predicate p_1 depends positively on p_2, then its stratum must be at least as high as the stratum of p_2. If p_1 depends negatively on p_2, then the stratum of p_1 must be higher as those of p_2 or both must be on the ω-stratum.

It was observed that such programs Π can be rewritten to VI-programs $F(\Pi)$ using special external predicates that compose/decompose terms from/into function symbols and a list of arguments, such that $F(\Pi)$ is VI-restricted [Calimeri et al., 2007].

We introduce for each $k \in \mathbb{N}$ two external predicates $\&compse_k$ and $\&decompse_k$ with $ar_{\text{I}}(\&compose_k) = 1 + k$ and $ar_{\text{O}}(\&compose_k) = 1$, and $ar_{\text{I}}(\&decompose_k) = 1$ and $ar_{\text{O}}(\&decompose_k) = 1 + k$. We define

$$f_{\&compose_k}(\mathbf{A}, f, X_1, \ldots, X_k, T) = f_{\&decompose_k}(\mathbf{A}, T, f, X_1, \ldots, X_k) = v$$

with $v = 1$ if $T = f(X_1, \ldots, X_k)$ and $v = 0$ otherwise.

Then composition and decomposition of function terms can be simulated using these external predicates. Intuitively, function terms are replaced by new variables and appropriate external

atoms with predicate $\&compose_k$ or $\&decompose_k$ are added to the rule body to compute their values (cf. Example 70).

As every VI-restricted program, viewed as a HEX-program, is by Proposition 9 also domain-expansion safe, we obtain:

Theorem 10. *For every logic program with function symbols* Π*, if* Π *is* ω*-restricted, then* $F(\Pi)$ *is domain-expansion safe and there is a 1-to-1 mapping between the answer sets of* Π *and* $F(\Pi)$*.*

Proof. By Theorem 6 of Calimeri et al. (2007), $F(\Pi)$ is VI-restricted, and thus by Theorem 9 also domain-expansion safe using $b_{synsem}(\Pi, r, S, B)$. The correspondence of the answer sets of Π and $F(\Pi)$ follows from Proposition 3 of Calimeri et al. (2007). □

As for the converse there exist programs Π with function terms which are not ω-restricted but such that $F(\Pi)$ is VI-restricted, VI-restrictedness is strictly more liberal than ω-restrictedness. It turns out that domain-expansion safety is even more liberal because there exist programs Π with function terms which are not ω-restricted such that $F(\Pi)$ is not VI-restricted but domain-expansion safe.

Example 70. Consider the program $\Pi = \{p(f(f(f(a)))).\ p(X) \leftarrow p(f(X))\}$. We get the translation $F(\Pi) = \{p(f(f(f(a)))).\ p(X) \leftarrow p(T), \&decompose_1[T](f, X)\}$. This program is not VI-restricted (and thus Π is not ω-restricted) because $p{\restriction}1$ occurs in a cycle in $G_A(\Pi)$ and cannot be declared domain-expansion safe by syntactic criteria. However, the program is domain-expansion safe by Condition (i) in Definition 70 using the well-ordering $\leq_{\mathcal{C}}^{strlen}$ s.t. $x \leq_{\mathcal{C}}^{eq} y$ if the length of the string x is shorter or equal to the one of y. Clearly, we have $X \leq_{\mathcal{C}}^{strlen} T$ for all output terms X of $\&decompose_1$ with input T. Thus, the cycle in $G_A(\Pi)$ turns out to be *benign*, which makes the program domain-expansion safe by Condition (i) in Definition 70. □

The reason why the program in Example 70 is not VI-restricted but domain-expansion safe is that it cannot be detected by syntactic criteria alone that the cycle produces only strictly smaller terms in each iteration. This requires semantic insights, which are captured by our notion of semantic term bounding function in Definition 70.

More expressive variants of ω-restricted programs are λ-restricted [Gebser et al., 2007b] and argument-restricted programs [Lierler and Lifschitz, 2009]. There are argument-restricted programs Π s.t. $F(\Pi)$ is *not* domain-expansion safe wrt. b_{synsem}. The reason is that specific properties of the external atoms for term (de)composition are exploited, while our approach uses general external sources. However, these classes of programs can be captured within our framework as well if tailored TBFs are used. This is not surprising as TBFs have full access to the program, thus the criteria of λ-restricted and argument-restricted programs can be checked by the TBF and all terms can be declared bounded if they hold. This shows the flexibility of our modular approach. The extension of argument-restricted programs by Greco et al. (2013), which is called *bounded programs*, also uses parameterization of safety criteria but focuses on programs with function symbols rather than general external sources; this notion might also be captured in our approach by using dedicated term bounding functions.

Similarly, by means of dedicated external atoms for (de)composing terms and a specialized TBF, so-called *FD programs* [Calimeri et al., 2008a] map into our framework. *Finitary programs* [Bonatti, 2001; Bonatti, 2002] and *FG programs* [Calimeri et al., 2008a], however, differ more fundamentally from our approach and cannot be captured as domain-expansion safe wrt. appropriate TBFs, as they are not effectively recognizable (and the former are in general not even finitely restrictable, i.e., there is no finite grounding which has the same answer sets as the original program).

Term Rewriting Systems. A term rewriting system is a set of rules for rewriting terms to other terms, cf. Klop (1992). Termination is usually shown by proving that the right-hand side of every rule is strictly smaller than its left-hand side [Zantema, 1994; Zantema, 2001]. Our notion of benign cycles is similar, but different from term rewriting systems the values do not need to *strictly* decrease. While terms that stay equal may prevent termination in term rewriting systems, they do not harm in our case because they cannot expand the grounding infinitely.

Other Notions of Safety. Related to semantic properties in our safety concept are the works by Sagiv and Vardi (1989), Ramakrishnan et al. (1987) and Krishnamurthy et al. (1996). They exploit finiteness of attributes (cf. Condition (ii) in Definition 70) in sets of Horn clauses and derive finiteness of further attributes using *finiteness dependencies*. This is related to Condition (iii) in Definition 70 and Condition (iii) in Definition 64.

Also related is the work of Heymans et al. (2004), who exploit syntactic restrictions to guarantee tree-shapedness of the models of the program. But unlike our approach, this does not guarantee finiteness of the model but only finite representability.

Less related to our approach are the works of Lee et al. (2008), Cabalar et al. (2009), and Bartholomew and Lee (2010), who extend safety, resp. argument restrictedness, to arbitrary first-order formulas without/with function symbols under the stable model semantics, rather than generalizing the concepts.

4.6.2 Summary and Future Work

We have presented a framework for obtaining classes of HEX-programs that allow for finite groundings sufficient for evaluation over an infinite domain (which arises by value invention in calls of external sources). It is based on *term bounding functions* (*TBFs*) and enables modular exchange and enhancement of such functions, and an easy combination of hitherto separate syntactic and semantic criteria into a single notion of *liberal domain expansion safety*. Our work pushes the classes of HEX-programs with evaluation via finite grounding considerably, leading to strictly larger classes than available via well-known criteria for answer set programs over infinite domains. We provided two concrete TBFs that capture syntactic criteria similar to but more fine-grained than the ones by Calimeri et al. (2007), and semantic criteria related to those of Sagiv and Vardi (1989), Ramakrishnan et al. (1987) but targeting model generation (not query answering).

We have then presented an algorithm for grounding arbitrary liberally domain-expansion safe HEX-programs. The algorithm is based on iterative grounding and checking whether the

grounding is large enough. The algorithm incorporates several optimizations which try to avoid the (expensive) evaluation of external atoms. A worst-case scenario for the grounding algorithm is a program, that contains cyclic dependencies of nonmonotonic external atoms. However, this worst-case can be effectively avoided in many programs using a newly developed decomposition heuristics (see below).

Next, we integrated the grounding algorithm into the existing evaluation framework for HEX-programs, which is extended for this purpose. In particular, we defined the notion of *generalized evaluation graphs*, which allows for using arbitrary liberally domain-expansion safe programs as units. In contrast to the traditional notion of evaluation graphs, splitting of programs is not necessary anymore, but still useful an some cases. Thus we developed a new evaluation heuristics which tries to achieve two contrary goals: splitting the program as rarely as possible (because this is harmful to the learning-based algorithms), but as often as necessary in order to avoid the worst-case for the grounding algorithm.

Issues for ongoing and future work are the identification of further TBFs and suitable well-orderings of domains in practice. On the algorithmic side, further refinement and optimizations are an interesting topic. The grounding algorithm may be extended in the future such that the worst-case can be avoided in more cases. Also other optimizations to the algorithm are possible, e.g., by reusing previous results of the grounding step instead of iterative regrounding of the whole program. Moreover, the new evaluation heuristics for the (extended) evaluation framework may be refined. Currently, the heuristics tries to avoid fusion of evaluation units whenever this would introduce cyclic dependencies of nonmonotonic external atoms, which is a worst-case for the grounding algorithm. However, sometimes this worst-case is not practically relevant because the concerned external atom has only few output tuples. Detecting such cases would allow for fewer evaluation units in some cases, which is an advantage for the solving algorithms.

The setup of a library of TBFs, which exploit specific properties of concrete external sources, is also a possible starting point for future work on the system side. Currently, the available TBFs exploit rather generic properties of external atoms. However, it is expected that domain-specific knowledge can be used to further relax safety criteria or speedup the grounding algorithm.

Chapter 5

Implementation and Evaluation

In this chapter we discuss the practical aspects of our work. We start with a description of the system implementation in Section 5.1, including its architecture, command-line options which are relevant in the context of this thesis, and the realization of specific features.

Then we present benchmark results for our new system implementation and compare them to the traditional algorithms for HEX-evaluation, which will show a significant speedup in many use cases. We first discuss evaluation of the learning-based algorithms for ground HEX-programs that we developed in Chapter 3 in Section 5.2. Although the encodings of our benchmark problems may involve variables and value invention, the hardness of the programs stems clearly from ground HEX-program solving rather than from grounding. The grounding algorithm from Chapter 4 is evaluated in Section 5.3, using a different benchmark suite for which grounding is computationally more sophisticated.

5.1 Implementation

In this section we give some details on the implementation of the techniques developed in this thesis. Our prototype system is called DLVHEX and is written in C++. It is available from http://www.kr.tuwien.ac.at/research/systems/dlvhex as open-source software. The sourcecode is hosted by https://github.com under https://github.com/hexhex. The system was initially released as version 1.0.0 in 2006. After major parts of the system were rewritten for architectural and efficiency reasons, and the model-building framework from Section 4.1 was introduced, version 2.0.0 appeared in March 2012. The current version (released in December 2013) is 2.3.0 and integrates all solving and grounding techniques from this thesis.

We first describe the general architecture, the major components, and their interplay. Then we give an overview about the command-line options of the system in general, and the new features compared to the previous version by Schüller (2012) specifically. We further show how

external source providers can define user-defined learning functions. Moreover, we show some language extensions developed during the work on this thesis.

5.1.1 System Architecture

The DLVHEX system architecture is shown in Figure 5.1. The arcs model both control and data flow within the system. The evaluation of a HEX-program works as follows.

First, the input program is read from the file system or from standard input and passed to the *evaluation framework* described in Section 4.1 ①. The evaluation framework creates then a *generalized evaluation graph* depending on the chosen evaluation heuristics. This results in a number of interconnected *generalized evaluation units*. While the interplay of the units is managed by the evaluation framework, the individual units are handeled by *model generators* of different kinds.

As described in Chapter 3, general program components use a guess-and-check algorithm, while monotonic program components may use a more efficient fixpoint iteration. This is realized as different model generators. Each instance of a model generator takes care of a single evaluation unit, receives *input interpretations* from the framework (which are either output by predecessor units or come from the input facts for leaf units), and sends output interpretations back to the framework ②, which manages the integration of these interpretations to final answer sets.

Internally, the model generators make use of a *grounder* and a *solver* for ordinary ASP programs. The architecture of our system is flexible and supports multiple concrete backends which can be plugged in. Currently it supports DLV, GRINGO 3.0.4 and CLASP 2.1.3, and an internal grounder and a solver which were built from scratch during the work on this thesis (mainly for testing purposes); they use basically the same core algorithms as GRINGO and CLASP, but without any kind of optimizations. The reasoner backends GRINGO and CLASP are statically linked to our system, thus no interprocess communication is necessary. The model generator within the DLVHEX core sends a non-ground program to the HEX-grounder, as described in Chapter 4, and receives a ground program ③. The HEX-grounder in turn uses an ordinary ASP grounder as submodule ④ and accesses external sources to handle value invention ⑤. The ground-program is then sent to the solver and answer sets of the ground program (i.e. candidate compatible sets) are returned ⑥. Note that the grounder and the solver are separated and communicate only through the model generator, which is in contrast to previous implementations of DLVHEX where the external grounder and solver were used as a single unit (i.e., the non-ground program was sent and the answer sets were retrieved). Separating the two units became necessary because the DLVHEX core needs access to the ground-program. Otherwise important structural information, such as cyclicity as used in Section 3.2, would be hidden.

The solver backend makes callbacks to the *post propagator* in the DLVHEX core once a model has been found or after unit and unfounded set propagation has been finished (actually, with DLV backend there are no callbacks during model building but only after a candidate compatible set has been found). During the callback, a complete or partial model is sent from the solver backend to the post propagator, and learned nogoods are sent back to the external solver ⑦. In case of CLASP as backend, we exploit its SMT interface, which was previously used for the special case of constraint answer set solving [Gebser et al., 2009]. The post propa-

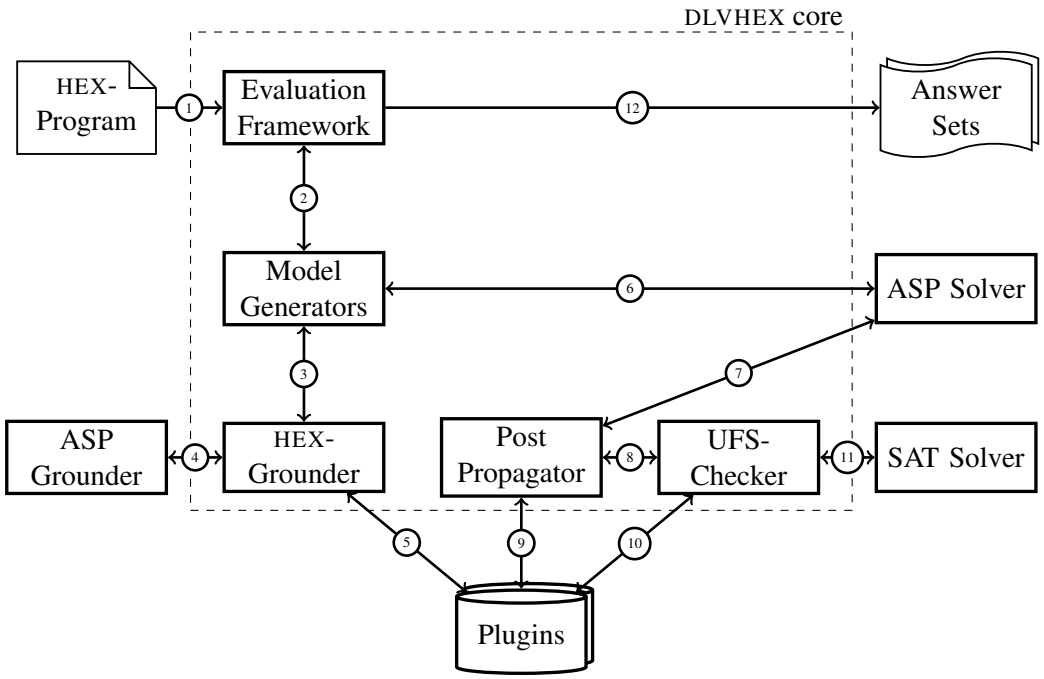


Figure 5.1: Architecture of DLVHEX

gator has then two key tasks: *compatibility checking with learning* and *unfounded set detection*. Compatibility checking, as formalized in Part (d) in Algorithm GuessAndCheckHexEvaluation and in Part (c) in Algorithm Hex-CDNL, checks whether the guesses of the external atom replacements by the ordinary ASP solver coincide with the actual semantics of the external source. This check also requires calls to the *plugins*, which implement the external sources. The input list is sent to the external source and the truth values and possibly user-defined learnt nogoods are returned to the post propagator ⑨. Moreover, the post propagator also sends the (complete or partial) model to the *unfounded set checker (UFS checker)* to find unfounded sets which are not detected by the ordinary ASP solver (unfounded sets caused by external sources). For this, the UFS checker employs a SAT solver ⑪, which can either be CLASP or the internal solver. The UFS checker possibly returns nogoods learned from unfounded sets to the post propagator ⑧. UFS detection also needs to call the external sources for guess verification, as shown in Algorithm FLPCheck ⑩. The post propagator sends all learned nogoods (either directly from external sources or from unfounded sets) back to the ASP solver. This makes sure that eventually only valid answer sets arrive at the model generator ⑥.

Finally, after the evaluation framework has built the final answer sets from the output interpretations of the individual evaluation units, they are output to the user ⑫.

5.1.2 Command-Line Options

DLVHEX supports various command-line options which control the algorithms. An exhaustive overview is available as online help which is printed if DLVHEX is run without arguments, i.e., `$ dlvhex2`.

The following list gives an overview about the most relevant options in the scope of this thesis and describes the possible parameter values.

- `--heuristics=[old,easy,monolithic,greedy]`
  Chooses the heuristics for the construction of the evaluation graph.
  `old` and `easy` are described by Eiter et al. (2011a)
  `monolithic` creates a single evaluation unit for the whole program
  `greedy` is the new default heuristics as described in Section 4.5

- `--solver=[dlv,genuineii,genuineic,genuinegi,genuinegc]`
  Selects the grounder and solver backend.
  `dlv` uses DLV for grounding and solving
  `genuineii` uses the internal grounder and solver
  `genuineic` uses the internal grounder and CLASP
  `genuinegi` uses GRINGO and the internal solver
  `genuinegc` uses GRINGO and CLASP

- `--eaevalheuristics=[never,inputcomplete,always]`
  Controls the heuristics for external atom evaluation in Part (c) of Algorithm Hex-CDNL Hex-CDNL.
  `never` does not evaluate external atoms during model building
  `inputcomplete` evaluates external atoms whenever their input is fully known
  `always` evaluates whenever the solver backend makes a callback to the post propagator

- `--ufscheckheuristics=[post,periodic,max]`
  Controls how often the UFS checker is invoked.
  `post` invokes it only for complete interpretations
  `periodic` invokes it in regular intervals
  `max` invokes it whenever the solver backend makes a callback to the post propagator

- `--flpcheck=[explicit,ufs,ufsm,aufs,aufsm]`
  Selects the algorithm for the FLP check.
  `explicit` uses the reduct as introduced at the beginning of Section 3.2
  `ufsm` uses the UFS algorithm with encoding Γ but without program decomposition
  `ufs` uses the UFS algorithm with encoding Γ and program decomposition
  `aufsm` uses the UFS algorithm with encoding Ω but without program decomposition
  `aufs` uses the UFS algorithm with encoding Ω and program decomposition

- `--ufslearn=[none,ufs,reduct]`
  Selects a strategy for learning from unfounded sets.
  `none` uses no learning

`ufs` learns as formalized by $L_1(U, \Pi, \mathbf{A})$ in Section 3.2.3
`reduct` learns as formalized by $L_2(U, \Pi, \mathbf{A})$ in Section 3.2.3

- `--extlearn=[none,iobehavior,monotonicity,functionality,linearity,neg,user]`

  Chooses one or more strategies for learning from external sources as described in Section 3.1.2.
  `none` uses no learning
  `iobehavior` learns in an uninformed fashion as described in Section 3.1.2
  `monotonicity` learns as described in Section 3.1.2 by exploiting (anti-)monotonicity
  `functionality` learns as described in Section 3.1.2 by exploiting functionality
  `linearity` learns as described in Section 3.1.2 by exploiting linearity
  `neg` learns as described in Section 3.1.2 by exploiting negative information
  `user` exploits user-defined learning functions (if provided by the plugin developer)
  If `--extlearn` is passed without any specific values, then all learning functions are activated (for those external sources which have the respective properties).

- `--noflpcriterion`
  Do not apply the decision criterion developed in Section 3.2.5.

- `--liberalsafety`
  Use liberal domain-expansion safety from Section 4.2 instead of strong domain-expansion safety.

A typical example for a complete call to DLVHEX is as follows:

```
$ dlvhex2 --heuristics=monolithic --solver=genuinegc --extlearn\
          --flpcheck=aufs --liberalsafety setpartitioning.hex
```

The syntax of the input language is very similar to the one used in this thesis, but with `:-` instead of $\leftarrow$ and letter `v` for $\vee$. Moreover, rules must be terminated with a dot.

Example 71. The program Π from Example 2 is encoded in file `setpartitioning.hex` follows:

```
sel(X) :- domain(X), &diff[domain, nsel](X).
nsel(X) :- domain(X), &diff[domain, sel](X).
domain(a).
```

□

It uses `monolithic` evaluation heuristics, GRINGO and CLASP as solver backends, turns all options for learning from external sources on, uses the UFS-based FLP check with encoding Ω, and uses liberal de-safety.

5.1.3 Heuristics for External Atom Evaluation and Unfounded Set Checking

Our implementation supports customized heuristics for steps (c) and (d) in Algorithm Hex-CDNL. A customized heuristics for external atom evaluation gets as input a partial assignment and an external atom in the program and decides whether this external atom shall be evaluated. In the positive case, the system evaluates the learning function associated with the external atom and its input list wrt. to the partial assignment (which may not even fully define the input to the external atom), and adds the respective nogoods to the set of dynamic nogoods. Note that this allows for learning nogoods which imply the truth values of yet unassigned input or output atoms of the external atom; this technique is well-known as *theory propagation* [Nieuwenhuis and Oliveras, 2005]. However, exploiting the power of this technique requires user-defined learning functions (see below) which need to be tightly coordinated with the external atom heuristics such that useful nogoods can be learned even wrt. partial interpretations. This step is strongly application dependent and beyond the scope of this thesis. However, our system provides a user-friendly programming interface for adding such learning functions and heuristics. In our benchmarks we used our default heuristics which evaluates external atoms whenever their input is completely known.

Customized heuristics for unfounded set checking also take as input a partial assignment and decide whether an unfounded set check shall be done. In the positive case, the unfounded set check is performed only over the subset-maximal subprogram over which the interpretation is already complete. As described in Section 3.2.4, this ensures that detected unfounded sets remain unfounded wrt. any completion of the assignment. We provide three concrete heuristics:

(1) Start unfounded set checking only wrt. complete interpretations.

(2) Start unfounded set checks periodically.

(3) Start unfounded set checks whenever no other propagation method yields additional assignments (i.e., before guessing).

However, our experiments have shown that (1) is superior to the other two methods in all our benchmarks. This is the case because unfounded set checking with external sources is a very expensive check, while the benefit of detecting unfounded sets earlier is marginal as learning effectively avoids the regeneration of unfounded sets anyway. Thus, we stick with heuristics (1) in our benchmarks.

5.1.4 User-Defined Learning Functions

User-defined learning functions as described in Section 3.1.2 can be implemented in two ways. The learned nogoods may be stated either directly as sets of signed literals, or encoded as (possibly non-ground) ASP rules. Stating them directly requires the user for writing some lines of C++ code to assemble the learned nogoods during evaluation of the external source. This may be more efficient than using ASP but less convenient. The traditional DLVHEX API for writing plugins requires the plugin developer to provide an implementation of the method:

```
void retrieve(const Query& query, Answer& answer);
```

It needs to transform a query object, containing the input list of the external atom and the extensions of the relevant predicates, into an answer object, containing all output tuples for which the external atom shall be true. This was described by Schüller (2012). While this method exploits specific properties of external sources, as described in Section 3.1.2, the user may specify custom learning methods by overriding the following method instead:

```
void retrieve(const Query& query, Answer& answer, NogoodContainerPtr ngcont);
```

Here, access to a *nogood container* is provided (roughly corresponding to the ASP solver), which allows for adding custom nogoods by appropriate function calls. The construction of the learned nogoods out of single literals is up to the user, but is supported by a library of helper functions, which may be used to construct parts of the nogoods automatically. For instance, there is a helper method which automatically constructs the set of all input atoms in the query object.

One particular helper function allows for writing learning rules in a fragment of ASP itself. The idea can be described as follows. Each rule specifies the preconditions for learning in terms of signed literals in the current assignment in its body. The head atom specifies the output tuples generated by the external source, i.e., they encode which atoms will be in the output (resp. not in the output) under these preconditions. We used the predicates in_i for $1 \leq i \leq n$ (for n-ary input) in the learning rule body and out resp. $nout$ in its head. For instance, for external predicate with input list $\&g[p_1, \ldots, p_n]$ the atom $in_2(c)$ in the body of a learning rule is true, if c is in the extension of the second input parameter, which is p_2 in this case. The atom $out(c)$ in the head of a learning rule states that c is in the output of the external source whenever the body of the learning rule is satisfied. Note that we have to use anonymized predicates in_i, $1 \leq i \leq n$ instead of the predicate names used in a certain external atom, because the learning rule must work also for different external atoms (but using the same external predicate).

Example 72. The behavior of external predicate with input list $\&diff[p, q]$ can compactly be described by the rule $out(X) \leftarrow in_1(X), \text{not } in_2(X)$. □

The application of a learning rule during evaluation of an external atom under assignment $\mathbf{A}$ works then as follows.

1. First, all predicates out resp. $nout$ are replaced by the auxiliary name $e_{\&g[\mathbf{y}]}$ for the respective external predicate with input list $\&g[\mathbf{y}]$.

2. Then, all signed literals $\mathbf{T}a \in \mathbf{A}$ from the current assignment $\mathbf{A}$ which are input atoms to the external atom (i.e. $a = p_i(c)$ for some c and $p_i \in \mathbf{y}$) are transformed into facts $a \leftarrow$.

3. The facts and the learning rule form the *learning program*, which is grounded in the third step. It is crucial to do the grounding *without optimization*. State-of-the-art grounders are highly optimized and evaluate the deterministic part of a program already during grounding as far as possible. This means in particular that atoms which are known to be true in all answer sets, are removed from rule bodies. If a program is stratified, which is usually the case for learning programs, it is completely evaluated to a set of facts. However, this is undesired here: because we want to use the ground rule to construct a learned nogood, we

have to ensure that it is correct also for future assignments, which may be different from the one wrt. which the external atom is currently evaluated.

4. Finally, the algorithm learns for each ground rule r of type $H(r) \leftarrow B^+(r), \text{not } B^-(r)$ the nogood $\{\mathbf{F}h \mid h \in H(r)\} \cup \{\mathbf{T}p \mid p \in B^+(r)\} \cup \{\mathbf{F}n \mid n \in B^-(r)\}$.

Example 73 (ctd.)**.** Continuing Example 72, the rule after renaming all predicates according to the external atom is $e_{\&diff[p,q]}(X) \leftarrow p(X), \text{not } q(X)$. Let $\mathbf{A} = \{\mathbf{T}p(a), \mathbf{T}p(b), \mathbf{F}p(c), \mathbf{T}q(a), \mathbf{F}q(b), \mathbf{T}q(c)\}$, then the grounded learning program is $e_{\&diff[p,q]}(b) \leftarrow p(b), \text{not } q(b)$ and the nogood which is learned by the system is $\{\mathbf{F}e_{\&diff[p,q]}(b), \mathbf{T}p(b), \mathbf{F}q(b)\}$. □

Observe that optimization during grounding would lead to an incorrect nogood. The ground rule would be $e_{\&diff[p,q]}(b) \leftarrow$ in this case, which leads to the nogood $\{\mathbf{F}e_{\&diff[p,q]}(b)\}$ that encodes that b is *always* in the output of the external source. However, this is only the case for the current input interpretation, the precondition $p(b), \text{not } q(b)$ was lost due to grounding optimization.

5.1.5 Language Extension for Property Specification

The development of the algorithms presented in Chapters 3 and 4 required lots of experiments with external sources of different kinds. Recall that our algorithms exploit syntactic and semantic properties of external sources, such as monotonicity. Such properties are in practical use reasonably specified by the provider of the respective external atoms. For this purpose, our C++ API offers *property lists* which can be assigned to external atoms. The properties of external sources are usually fixed once the plugin has been developed.

However, during the work on this thesis it was of great interest to experiment with different property lists for the same external atom. For instance, it was important to see what happens if an external atom is monotonic, but not declared as such, i.e., the property is concealed from the system. Then the system might be forced to use a more general and slower algorithm than if the property was known. Conversely, it was sometimes also interesting to see how the system behaves if a property is wrongly asserted, e.g., some nonmonotonic source is declared to be monotonic. On the one hand, this was useful for making the system more robust against user errors and programming errors in plugins, on the other hand the observations also led to a better intuitive understanding of the algorithms.

It would have been cumbersome to change the C++ code of external sources for every experiment with a different property list. Thus, the following extension of the HEX-language was developed. External atoms $\&g[\mathbf{y}](\mathbf{x})$ can now be post-fixed by *inline property lists* directly in the HEX-program, which are of kind $\langle P_1, \ldots, P_n \rangle$ where the P_i for $1 \leq i \leq n$ are *property assertions*, such as $monotonic\ y_k$ to define that $\&g$ is monotonic in predicate input parameter $y_k \in \mathbf{y}$, or $finitedomain\ 2$ to define that the second output element x_2 has a finite domain. For an exhaustive description of the supported property assertions, we refer to the system documentation.

5.2 Evaluation of the Learning-based Algorithms

We evaluated the implementation on a Linux server with two 12-core AMD 6176 SE CPUs with 128GB RAM running an *HTCondor* load distribution system[1] which ensures robust runtimes (i.e., multiple runs of the same instance have negligible deviations) and using DLVHEX version 2.3.0. The grounder and solver backends for all benchmarks are GRINGO 3.0.4 and CLASP 2.1.3. For each instance, we limited the CPU usage to two cores and 8GB RAM. The timeout for all instances was 300 seconds. The instances of all benchmarks discussed in this section are available as compressed tar archives from http://www.kr.tuwien.ac.at/staff/redl/aspext. The required plugins are available from the repository (https://github.com/hexhex).

For evaluating the solving algorithms for ground HEX-programs, there is a large number of combinations of the techniques developed in this thesis. We may either activate or deactivate external behavior learning (EBL) and use either the explicit or the UFS-based minimality check. In the latter case, we can further use unfounded set learning (UFL), the decision criterion for skipping the unfounded set check can be exploited or ignored, and program decomposition might be used. Moreover, we can choose between the encodings Γ and Ω.

However, we will restrict our discussion to some interesting configurations. In general, we will activate the developed features stepwise such that in our tables the efficiency increases from left to right. We will start with the traditional algorithm based on an explicit minimality check without any learning techniques described in this thesis (i.e., only conflict-driven learning inside CLASP is used). In the next step we will add external behavior learning (EBL), while UFL is not possible with the explicit check. Then we switch from the explicit minimality check to the UFS-based one without learning and without exploiting the decision criterion and program decomposition. Nevertheless, this naive kind of the UFS-based minimality checking is usually already more efficient than the explicit minimality check with EBL. In the next step, we add the decision criterion and program decomposition. In the following, *monolithic (mol.)* means that both the decision criterion and the program decomposition are off, and *modular (mod.)* that they are on. Then we add EBL and UFL to the UFS-based minimality check, which leads usually to a significant speedup. Finally, we switch the encoding from Γ to Ω; in our experiments we always enable modular decomposition and the decision criterion if encoding Ω is used. Since the two encodings have different variables and clauses, their respective search spaces are of a different structure, which may prefer or penalize the one or the other encoding for a given instance. However, the systematic difference between the encodings is that Ω needs to be constructed only once, which is an advantage and often leads to a smaller overall runtime. We might skip some of the steps for specific benchmarks and argue why they are uninteresting in the respective cases. The numbers in parentheses indicate the number of instances and the number of timeouts in the respective categories. In all benchmarks of this section we used the `monolithic` decomposition heuristics, i.e., we do not split the program.

We can see a clear improvement both for synthetic and for application instances, due to the UFS check and EBL. Since some benchmarks are motivated by real applications, they will be discussed in more detail in Chapter 6. Here we only give a brief description as far as this is necessary to understand the benchmark results. A closer analysis shows that the UFS check in

[1] http://research.cs.wisc.edu/htcondor

domain	All Answer Sets						First Answer Set					
	explicit	+EBL	UFS Γ mol.	UFS Γ mod.	+EBL	UFS Ω +EBL	explicit	+EBL	UFS Γ mol.	UFS Γ mod.	+EBL	UFS Ω +EBL
1 (1)	0.05 (0)	0.05 (0)	0.04 (0)	0.04 (0)	0.04 (0)	0.04 (0)	0.04 (0)	0.04 (0)	0.04 (0)	0.04 (0)	0.04 (0)	0.04 (0)
2 (1)	0.28 (0)	0.20 (0)	0.04 (0)	0.04 (0)	0.04 (0)	0.04 (0)	0.09 (0)	0.10 (0)	0.04 (0)	0.04 (0)	0.04 (0)	0.04 (0)
3 (1)	4.65 (0)	2.82 (0)	0.06 (0)	0.05 (0)	0.05 (0)	0.05 (0)	0.70 (0)	0.70 (0)	0.04 (0)	0.04 (0)	0.04 (0)	0.04 (0)
4 (1)	69.66 (0)	36.64 (0)	0.14 (0)	0.14 (0)	0.06 (0)	0.06 (0)	6.34 (0)	6.35 (0)	0.04 (0)	0.04 (0)	0.05 (0)	0.05 (0)
5 (1)	300.00 (1)	300.00 (1)	0.33 (0)	0.32 (0)	0.09 (0)	0.07 (0)	54.02 (0)	53.80 (0)	0.05 (0)	0.05 (0)	0.05 (0)	0.05 (0)
6 (1)	300.00 (1)	300.00 (1)	0.77 (0)	0.81 (0)	0.12 (0)	0.10 (0)	300.00 (1)	300.00 (1)	0.04 (0)	0.05 (0)	0.06 (0)	0.06 (0)
7 (1)	300.00 (1)	300.00 (1)	1.73 (0)	1.78 (0)	0.20 (0)	0.13 (0)	300.00 (1)	300.00 (1)	0.06 (0)	0.06 (0)	0.06 (0)	0.07 (0)
8 (1)	300.00 (1)	300.00 (1)	4.35 (0)	4.17 (0)	0.31 (0)	0.16 (0)	300.00 (1)	300.00 (1)	0.07 (0)	0.06 (0)	0.07 (0)	0.07 (0)
9 (1)	300.00 (1)	300.00 (1)	10.42 (0)	10.21 (0)	0.47 (0)	0.23 (0)	300.00 (1)	300.00 (1)	0.08 (0)	0.07 (0)	0.08 (0)	0.09 (0)
10 (1)	300.00 (1)	300.00 (1)	26.31 (0)	25.13 (0)	0.53 (0)	0.29 (0)	300.00 (1)	300.00 (1)	0.09 (0)	0.09 (0)	0.11 (0)	0.12 (0)
15 (1)	300.00 (1)	300.00 (1)	300.00 (1)	300.00 (1)	2.83 (0)	0.79 (0)	300.00 (1)	300.00 (1)	0.19 (0)	0.15 (0)	0.27 (0)	0.26 (0)
20 (1)	300.00 (1)	300.00 (1)	300.00 (1)	300.00 (1)	12.98 (0)	1.95 (0)	300.00 (1)	300.00 (1)	0.38 (0)	0.29 (0)	0.57 (0)	0.57 (0)
25 (1)	300.00 (1)	300.00 (1)	300.00 (1)	300.00 (1)	45.18 (0)	4.11 (0)	300.00 (1)	300.00 (1)	0.70 (0)	0.47 (0)	1.09 (0)	1.08 (0)

Table 5.1: Set Partitioning – Benchmark Results

some cases not only decreases the runtime but also the numbers of enumerated candidates (UFS candidates resp. model candidates of the FLP-reduct) and of external atom evaluations.

5.2.1 Detailed Benchmark Description

Set Partitioning. This benchmark extends the program from Example 32 by the additional constraint $\leftarrow sel(X), sel(Y), sel(Z), X \neq Y, X \neq Z, Y \neq Z$ and varies the size of $domain$. The results are shown in Table 5.1. We can observe a big advantage of the UFS check over the explicit check, both for computing all answer sets and for finding the first one. A closer investigation shows that the improvement is mainly due to the optimizations described in Section 3.2.3 which make the UFS check investigate significantly fewer candidates than the explicit FLP check. Furthermore the UFS check requires fewer external computations.

Both the explicit and the UFS-based minimality check benefit from EBL if we compute all answer sets, but the results show that the UFS-based check benefits more. In contrast, UFL (not shown in the table) does not lead to a further speedup because no unfounded sets will be detected in this program. Also the decision criterion and program decomposition (not shown in the table) do not help because there is a cycle which involves the whole program.

If we compute only one answer set, then EBL turns out to be counterproductive. This is because learning is involved with additional overhead, while the algorithm cannot profit much from the learned knowledge if it aborts after the first answer set, hence the costs exceed the benefit.

Using the encoding Ω instead of Γ increases the efficiency in this case, because there is not only a large number of answer sets but also a large number of answer set candidates. Thus, a reusable encoding is very beneficial, even if we compute only one answer set.

Multi-Context Systems (MCSs). Multi-context systems [Brewka and Eiter, 2007] are a formalism for interlinking knowledge based systems. So-called *bridge rules* are used to intercon-

nect the single knowledge bases by deriving the truth of some atom in one knowledge base depending on atoms in different knowledge bases. More details are discussed in Chapter 6.

An important reasoning task for MCSs is finding *inconsistency explanations (IEs)*, as defined by Eiter et al. (2010), in terms of sets of sets of bridge rules which cause this inconsistency. This benchmark computes the IEs, which correspond 1-1 to answer sets of an encoding rich in cycles through external atoms (which evaluate local knowledge base semantics). We used random instances of different topologies created with an available benchmark generator.

For most instances, we observed that the number of candidates for smaller models of the FLP-reduct equals the one of unfounded set candidates. This is intuitive as each unfounded set corresponds to a smaller model; the optimization techniques do not prune the search space in this case. However, as we stop the enumeration as soon as a smaller model resp. an unfounded set is found, depending on the specific program and solver heuristics, the explicit and the UFS check may consider different numbers of interpretations. This explains why the UFS check is sometimes slightly slower than the explicit check. However, it always has a smaller delay between different UFS candidates, which sometimes makes it faster even if it visits more candidates.

Note that MCS topologies are bound to certain system sizes, and the difficulty of the instances varies among topologies; thus larger instances may have shorter runtimes.

The results for consistent MCSs are shown in Table 5.2 and the results for inconsistent MCSs in Table 5.3. Consistent MCSs have no answer sets, thus we do not distinguish between computing all and the first answer set. The effects of external behavior learning and of unfounded set learning are clearly evident in the MCS benchmarks, both for computing all and for computing the first answer set. The UFS check profits more from EBL than the explicit check, further adding to its advantage. By activating UFL (not possible in the explicit check) we gain another significant speedup.

Intuitively, consistent and inconsistent MCSs are dual, as for each candidate the explicit resp. UFS check fails, i.e., stops early, vs. for some (or many) candidates the check succeeds (stops late). However, the mixed results do not permit us to draw solid conclusions on the computational relationship of the evaluation methods.

We now discuss the effects of the syntactic decision criterion and program decomposition. We used the HEX-encoding by Eiter et al. (2012d), which contains *saturation* over external atoms, where nearly all cycles in the HEX-program contain at least one external atom[2]. Therefore, the decision criterion can reduce the set of atoms, for which the UFS check needs to be performed, only by the atoms that are defined in the EDB. This does not yield significant efficiency improvements. However, the benchmark results for MCS instances confirm that the syntactic check is very cheap and does not hurt performance, even if the instance does not admit

[2]The *saturation technique* (cf. e.g. Eiter et al. (2009)) is a programming technique in disjunctive ASP that exploits the subset-minimality property of answer sets for checking that a given property holds for *all* guesses in a given search space. The idea is that whenever the property holds for all assignments, the program has a unique *saturation model*, containing a unique atom a_{sat} and all 'bad' assignments, i.e., assignments which do not fulfill the property. Thus, the saturation model is a proper superset of any bad assignment. Then by minimality of answer sets, the saturation model is an answer set if and only if there is no bad assignment. Other rules in the program may refer to a_{sat} to check if the property holds. A typical example is the check if a graph is *not* 3-colorable, i.e., all possible colorings violate the conditions. A similar technique can be realized with external atoms.

#ctx	explicit	+EBL	UFS Γ mol.	UFS Γ mod.	+EBL	+UFL	UFS Ω +EBL+UFL
3 (6)	4.78 (0)	3.97 (0)	2.96 (0)	2.97 (0)	1.65 (0)	0.08 (0)	0.08 (0)
4 (10)	51.90 (1)	45.91 (1)	48.71 (1)	48.59 (1)	23.48 (0)	0.10 (0)	0.11 (0)
5 (8)	149.53 (3)	137.95 (3)	150.80 (3)	150.64 (3)	94.45 (1)	0.10 (0)	0.12 (0)
6 (6)	159.41 (3)	154.69 (3)	157.62 (3)	157.72 (3)	151.89 (3)	0.12 (0)	0.15 (0)
7 (12)	231.23 (9)	227.45 (9)	234.74 (9)	234.63 (9)	216.75 (8)	0.17 (0)	0.20 (0)
8 (5)	244.39 (4)	204.92 (3)	246.42 (4)	246.34 (4)	190.60 (3)	0.17 (0)	0.21 (0)
9 (8)	300.00 (8)	278.44 (7)	300.00 (8)	300.00 (8)	264.65 (6)	0.22 (0)	0.24 (0)
10 (11)	300.00 (11)	268.78 (9)	300.00 (11)	300.00 (11)	247.16 (8)	0.25 (0)	0.31 (0)

Table 5.2: Consistent MCSs – Benchmark Results

#ctx	All Answer Sets						
	explicit	+EBL	UFS Γ mol.	UFS Γ mod.	+EBL	+UFL	UFS Ω +EBL+UFL
3 (9)	3.29 (0)	2.70 (0)	2.44 (0)	2.34 (0)	1.09 (0)	0.14 (0)	0.14 (0)
4 (14)	41.57 (1)	17.94 (0)	37.04 (1)	37.03 (1)	6.05 (0)	2.71 (0)	0.61 (0)
5 (11)	154.55 (5)	148.11 (5)	154.17 (5)	153.94 (5)	108.87 (2)	3.65 (0)	1.28 (0)
6 (18)	130.90 (7)	102.57 (6)	128.26 (7)	128.12 (7)	87.75 (4)	10.61 (0)	1.55 (0)
7 (13)	166.14 (5)	118.04 (5)	157.67 (5)	157.06 (5)	107.50 (4)	84.08 (2)	29.47 (0)
8 (6)	261.96 (5)	143.75 (2)	262.95 (5)	263.00 (5)	118.36 (2)	55.86 (1)	51.13 (1)
9 (14)	286.74 (13)	206.10 (9)	287.10 (12)	287.32 (12)	189.48 (8)	124.34 (5)	130.56 (6)
10 (12)	300.00 (12)	300.00 (12)	300.00 (12)	300.00 (12)	290.18 (11)	290.69 (11)	277.05 (11)

#ctx	First Answer Set						
	explicit	+EBL	UFS Γ mol.	UFS Γ mod.	+UFL	+EBL	UFS Ω +EBL+UFL
3 (9)	0.09 (0)	0.09 (0)	0.08 (0)	0.08 (0)	0.08 (0)	0.08 (0)	0.09 (0)
4 (14)	0.13 (0)	0.14 (0)	0.11 (0)	0.12 (0)	0.12 (0)	0.11 (0)	0.13 (0)
5 (11)	0.16 (0)	0.17 (0)	0.14 (0)	0.14 (0)	0.14 (0)	0.14 (0)	0.16 (0)
6 (18)	0.18 (0)	0.19 (0)	0.16 (0)	0.16 (0)	0.15 (0)	0.15 (0)	0.18 (0)
7 (13)	0.19 (0)	0.17 (0)	0.17 (0)	0.17 (0)	0.15 (0)	0.15 (0)	0.17 (0)
8 (6)	0.23 (0)	0.20 (0)	0.21 (0)	0.20 (0)	0.17 (0)	0.17 (0)	0.19 (0)
9 (14)	0.32 (0)	0.27 (0)	0.28 (0)	0.28 (0)	0.22 (0)	0.23 (0)	0.28 (0)
10 (12)	0.44 (0)	0.33 (0)	0.39 (0)	0.39 (0)	0.29 (0)	0.29 (0)	0.34 (0)

Table 5.3: Inconsistent MCSs – Benchmark Results

to seriously reduce the size of the UFS check. There is also no difference in the number of instance timeouts.

If we use encoding Ω instead of Γ, we can observe another significant speedup if we compute all models of inconsistent MCSs. This is because there usually exist many answer sets (often many thousands), and therefore a reusable encoding is very beneficial. In contrast, if we compute only the first answer set or the MCS is consistent (and has therefore no answer set), then the check becomes slightly slower with encoding Ω than with Γ. This is because of the higher complexity of the encoding, while the reusability does not help if we abort after the first first answer set.

In summary, we can observe that the encoding Ω leads to a significant performance gain over encoding Γ, while the decision criterion and decomposition do not help. In our next benchmark we will observe converse effects.

Abstract Argumentation. In this benchmark we compute ideal sets (cf. [Dung et al., 2007]) for randomly generated instances of abstract argumentation frameworks (AFs) [Dung, 1995] of different sizes. Roughly, this corresponds to the detection of sets of nodes in a directed graph, which fulfill certain conditions. The problem of checking whether a given set of arguments is an ideal set of an AF is co-NP-complete [Dunne, 2009]. In this benchmark we used a HEX encoding that mirrors this complexity: it guesses such a set and checks its ideality using the Saturation technique involving an external atom. The HEX-encoding has been worked out by Peter Schüller, to whom the author is grateful for his work, and is described in Section A.2.

Table 5.4 shows average runtimes for different numbers of arguments, each accumulated over 30 benchmark instances. We compare the following configurations both for computing all and for computing the first answer set. In the first column we do an explicit minimality check without learning techniques. Our experiments have shown that learning (EBL) leads to slightly higher runtime results (second column). This can be explained by the structure of the encoding, which does not allow for effectively reusing learned nogoods but learning causes additional overhead.

In the third column, we perform an UFS-based minimality check without application of the decision criterion and decomposition using our encoding Γ. We can observe that this already significantly improves the results compared to the explicit minimality check, which proves the effectiveness of our new approach. As with MCS, the numbers of reduct model candidates and UFS candidates are in most cases equal, but the UFS check again enumerates its candidates faster; this explains the observed speedup.

Next we enable the decision criterion and program decomposition and can observe a further speedup. This is because cycles in our argumentation instances usually involve only small parts of the overall program, thus the UFS search can be significantly simplified by excluding large parts of the programs. We further have observed that program decomposition without application of the decision criterion is counterproductive (not shown in the table), because a single UFS search over the whole program is replaced by many UFS searches over program components (without the decision criterion, no such check is excluded). This involves more overhead.

In the fifth column we enable EBL and UFL, which leads to a small speedup in some cases. However, as already mentioned above, no effective reuse of learned nogoods is possible.

Finally, we switch the encoding from Γ to Ω, which leads to a small speedup in some cases, but is also counterproductive in other cases. This is explained by the different search spaces traversed when the encoding is switched, which may have positive or negative effects on efficiency. On the other hand, because of the small number of answer sets of the instances in this benchmark, only few unfounded set checks are performed anyway. Thus, the lower initialization overhead of the encoding Ω does not influence the runtime dramatically and there is no systematic advantage of Ω.

Conformant Planning. This benchmark was implemented and carried out by Peter Schüller; the encoding is described in Section A.2. A planning problem consists of a set of actions, their preconditions and effects in the world. The most important reasoning task is to find a sequence of actions which reaches a certain goal state from a given initial state. In conformant planning, this needs to be done in a nondeterministic domain and with an only partially specified initial

#args	All Answer Sets					
	explicit	+EBL	UFS Γ mol.	UFS Γ mod.	+EBL+UFL	UFS Ω +EBL+UFL
1 (30)	0.06 (0)	0.06 (0)	0.05 (0)	0.05 (0)	0.05 (0)	0.05 (0)
2 (30)	0.08 (0)	0.07 (0)	0.06 (0)	0.06 (0)	0.06 (0)	0.07 (0)
3 (30)	0.11 (0)	0.10 (0)	0.08 (0)	0.08 (0)	0.08 (0)	0.09 (0)
4 (30)	0.19 (0)	0.19 (0)	0.14 (0)	0.12 (0)	0.12 (0)	0.13 (0)
5 (30)	0.32 (0)	0.32 (0)	0.26 (0)	0.18 (0)	0.18 (0)	0.19 (0)
6 (30)	0.71 (0)	0.72 (0)	0.55 (0)	0.33 (0)	0.33 (0)	0.36 (0)
7 (30)	1.58 (0)	1.66 (0)	1.16 (0)	0.52 (0)	0.51 (0)	0.56 (0)
8 (30)	4.75 (0)	5.04 (0)	3.06 (0)	1.09 (0)	1.08 (0)	1.15 (0)
9 (30)	14.02 (0)	14.97 (0)	8.65 (0)	1.86 (0)	1.84 (0)	1.95 (0)
10 (30)	41.10 (0)	44.38 (0)	24.53 (0)	4.73 (0)	4.58 (0)	4.79 (0)
11 (30)	129.35 (1)	139.80 (2)	51.39 (0)	9.34 (0)	9.34 (0)	9.48 (0)
12 (30)	250.16 (12)	258.82 (17)	119.44 (0)	12.49 (0)	12.38 (0)	12.39 (0)
13 (30)	294.91 (27)	296.67 (27)	274.65 (19)	24.26 (0)	24.33 (0)	24.44 (0)
14 (30)	290.01 (29)	290.01 (29)	290.00 (29)	51.38 (3)	51.65 (3)	51.98 (3)
15 (30)	290.01 (29)	290.01 (29)	290.00 (29)	79.93 (3)	78.00 (3)	78.19 (3)
16 (30)	300.00 (30)	300.00 (30)	300.00 (30)	80.10 (4)	77.91 (4)	77.95 (4)
17 (30)	300.00 (30)	300.00 (30)	300.00 (30)	81.90 (5)	77.04 (5)	76.85 (5)
18 (30)	300.00 (30)	300.00 (30)	300.00 (30)	127.43 (8)	126.57 (8)	125.91 (8)
19 (30)	300.00 (30)	300.00 (30)	280.39 (28)	173.16 (13)	148.13 (10)	147.62 (10)
20 (30)	300.00 (30)	300.00 (30)	278.20 (27)	167.72 (12)	167.02 (12)	166.07 (12)

#args	First Answer Set					
	explicit	+EBL	UFS Γ mol.	UFS Γ mod.	+EBL+UFL	UFS Ω +EBL+UFL
1 (30)	0.05 (0)	0.05 (0)	0.05 (0)	0.05 (0)	0.05 (0)	0.05 (0)
2 (30)	0.07 (0)	0.07 (0)	0.06 (0)	0.06 (0)	0.06 (0)	0.06 (0)
3 (30)	0.09 (0)	0.09 (0)	0.08 (0)	0.08 (0)	0.07 (0)	0.08 (0)
4 (30)	0.14 (0)	0.14 (0)	0.12 (0)	0.10 (0)	0.10 (0)	0.12 (0)
5 (30)	0.22 (0)	0.22 (0)	0.21 (0)	0.15 (0)	0.15 (0)	0.17 (0)
6 (30)	0.46 (0)	0.47 (0)	0.42 (0)	0.27 (0)	0.27 (0)	0.29 (0)
7 (30)	0.76 (0)	0.79 (0)	0.68 (0)	0.37 (0)	0.37 (0)	0.40 (0)
8 (30)	2.34 (0)	2.44 (0)	1.98 (0)	0.89 (0)	0.90 (0)	0.94 (0)
9 (30)	7.35 (0)	7.82 (0)	5.76 (0)	1.36 (0)	1.28 (0)	1.34 (0)
10 (30)	19.47 (0)	21.05 (0)	15.37 (0)	3.54 (0)	3.53 (0)	3.68 (0)
11 (30)	63.39 (1)	67.39 (1)	26.30 (0)	4.61 (0)	4.66 (0)	4.69 (0)
12 (30)	119.65 (4)	126.18 (4)	60.88 (0)	6.11 (0)	6.11 (0)	6.13 (0)
13 (30)	197.04 (14)	201.27 (15)	149.25 (3)	16.34 (0)	16.49 (0)	16.50 (0)
14 (30)	227.27 (22)	227.72 (22)	218.00 (17)	41.28 (2)	41.68 (2)	41.76 (2)
15 (30)	260.02 (26)	260.02 (26)	260.01 (26)	40.92 (2)	41.38 (2)	41.62 (2)
16 (30)	230.04 (23)	230.04 (23)	230.02 (23)	40.63 (3)	40.69 (3)	40.84 (3)
17 (30)	250.03 (25)	250.03 (25)	250.01 (25)	35.24 (2)	35.60 (2)	35.57 (2)
18 (30)	270.02 (27)	270.02 (27)	270.01 (27)	74.89 (5)	75.47 (5)	75.10 (5)
19 (30)	230.06 (23)	230.06 (23)	211.12 (21)	66.58 (4)	67.03 (4)	67.04 (4)
20 (30)	220.07 (22)	220.07 (22)	200.29 (20)	81.81 (5)	82.33 (5)	82.45 (5)

Table 5.4: Argumentation – Benchmark Results

state such that the computed plan reaches the goal for all possible nondeterministic changes in the world and for all possible initial states.

In this benchmark we assume that two robots with a limited sensor range patrol an area. An object in the area is at an unknown location (partially specified initial state). We are looking for a sequence of movements of the two robots such that the object is detected in all cases. Deciding whether a robot detects the object if is is at a certain location is done by an external atom.

The results are displayed in Table 5.5. For each size of the area we present the average runtimes over 10 instances with randomized robot and object locations. The instances have $x \times 4$ grids with a required plan length of $x \in \{3, \ldots, 9\}$, for finding a solution.

The explicit FLP check performs worst, which is as expected, followed by the monolithic UFS check using encoding Γ, followed by modular UFS using encoding Γ. The best results are achieved using the UFS check with Ω encoding (but without external behavior learning and without unfounded set learning) performs best.

Interestingly and in contrast to all other benchmarks, although EBL and UFL decrease the number of unfounded set checks required and the number of external atoms evaluated, they both have negative effects on the runtime. EBL and UFL decrease the performance significantly for the modular UFS Γ check and slightly for the Ω check. As for the explicit check the runtimes do not change with EBL, we omit these results. Detailed analysis identified two reasons for this observation. First, the external atoms depend on a large part of the interpretation (locations of the robots and the object), thus EBL cannot eliminate significant portions of the search space. Second, the external atom is efficiently computable, i.e., it takes only a negligible amount of time. Thus beneficial effects of EBL do not become evident or become hidden by the computational overhead introduced by learning. Also UFL is not effective because the instances of this benchmark contain only few unfounded sets (less than half of the answer set candidates are eliminated) therefore UFL does not improve performance significantly but introduces additional overhead.

With UFS encoding Ω, the UFS check encoding is constructed only once, thus the overhead of EBL and UFL observed with encoding Γ does no longer have such a big impact but is still visible.

For small area sizes one can observe that for the encoding Ω, the 3×4 instance actually seem to be harder to solve than the larger 4×4 and 5×4 instances. This is because all these instances require plan length of only 1, while the larger instances are more constrained. Thus the robots have less freedom to move around while still detecting the object. Consequently, for 5×4 maps the solver finds solutions faster than for 4×4 areas.

We conclude that in some scenarios, using EBL and UFL can reduce efficiency.

Default Reasoning over Description Logics Benchmarks. We consider now a scenario using the DL-plugin [Eiter et al., 2008] for DLVHEX, which integrates description logic (DL) knowledge bases and nonmonotonic logic programs. The DL-plugin allows to access an ontology using the description logic reasoner `RacerPro` 2.0 (http://www.racer-systems.com). For our first experiment, consider the program (shown left) and the terminological part of a DL knowledge

Map Size	Plan Length	All Answer Sets						
		explicit	UFS Γ mol.	UFS Γ mod.	+EBL	+UFL	UFS Ω -EBL-UFL	UFS Ω +EBL+UFL
3×4 (10)	1	7.10 (0)	0.12 (0)	0.11 (0)	0.11 (0)	0.12 (0)	0.12 (0)	0.14 (0)
4×4 (10)	1	10.66 (0)	0.16 (0)	0.15 (0)	0.15 (0)	0.15 (0)	0.15 (0)	0.18 (0)
5×4 (10)	1	10.69 (0)	0.15 (0)	0.15 (0)	0.14 (0)	0.14 (0)	0.13 (0)	0.15 (0)
6×4 (10)	2	206.45 (2)	1.98 (0)	1.38 (0)	1.67 (0)	1.69 (0)	1.09 (0)	1.35 (0)
7×4 (10)	2	258.82 (5)	2.85 (0)	1.79 (0)	2.44 (0)	2.43 (0)	1.50 (0)	1.84 (0)
8×4 (10)	3	300.00 (10)	36.80 (0)	16.41 (0)	40.94 (0)	40.99 (0)	10.42 (0)	13.88 (0)
9×4 (10)	3	300.00 (10)	43.20 (0)	19.53 (0)	78.11 (0)	77.10 (0)	13.91 (0)	19.62 (0)
10×4 (10)	4	300.00 (10)	300.00 (10)	274.53 (5)	300.00 (10)	300.00 (10)	203.70 (2)	252.31 (5)
11×4 (10)	4	300.00 (10)	299.76 (9)	239.61 (5)	300.00 (10)	300.00 (10)	174.86 (2)	209.41 (3)
12×4 (10)	5	300.00 (10)	300.00 (10)	300.00 (10)	300.00 (10)	300.00 (10)	300.00 (10)	300.00 (10)
13×4 (10)	5	300.00 (10)	300.00 (10)	300.00 (10)	300.00 (10)	300.00 (10)	300.00 (10)	300.00 (10)
14×4 (10)	6	300.00 (10)	300.00 (10)	300.00 (10)	300.00 (10)	300.00 (10)	300.00 (10)	300.00 (10)
15×4 (10)	6	300.00 (10)	300.00 (10)	300.00 (10)	300.00 (10)	300.00 (10)	300.00 (10)	300.00 (10)
16×4 (10)	7	300.00 (10)	300.00 (10)	300.00 (10)	300.00 (10)	300.00 (10)	300.00 (10)	300.00 (10)

Map Size	Plan Length	First Answer Set						
		explicit	UFS Γ mol.	UFS Γ mod.	+EBL	+UFL	UFS Ω -EBL-UFL	UFS Ω +EBL+UFL
3×4 (10)	1	0.89 (0)	0.05 (0)	0.05 (0)	0.05 (0)	0.05 (0)	0.06 (0)	0.06 (0)
4×4 (10)	1	1.36 (0)	0.06 (0)	0.05 (0)	0.05 (0)	0.06 (0)	0.06 (0)	0.06 (0)
5×4 (10)	1	2.23 (0)	0.06 (0)	0.07 (0)	0.06 (0)	0.06 (0)	0.07 (0)	0.07 (0)
6×4 (10)	2	7.21 (0)	0.22 (0)	0.15 (0)	0.14 (0)	0.14 (0)	0.12 (0)	0.13 (0)
7×4 (10)	2	17.39 (0)	0.34 (0)	0.22 (0)	0.21 (0)	0.20 (0)	0.17 (0)	0.18 (0)
8×4 (10)	3	139.26 (1)	6.07 (0)	2.73 (0)	2.73 (0)	2.69 (0)	1.45 (0)	1.78 (0)
9×4 (10)	3	150.50 (3)	3.24 (0)	1.47 (0)	1.69 (0)	1.70 (0)	0.89 (0)	1.16 (0)
10×4 (10)	4	255.89 (7)	92.19 (2)	47.58 (0)	82.84 (2)	82.52 (2)	24.23 (0)	31.36 (0)
11×4 (10)	4	300.00 (10)	97.11 (2)	39.99 (0)	84.08 (1)	83.85 (1)	19.53 (0)	25.85 (0)
12×4 (10)	5	287.76 (9)	198.75 (5)	143.52 (4)	184.81 (5)	184.78 (5)	131.46 (4)	136.64 (4)
13×4 (10)	5	300.00 (10)	287.07 (9)	211.97 (5)	277.79 (9)	277.71 (9)	165.64 (4)	185.84 (4)
14×4 (10)	6	300.00 (10)	300.00 (10)	244.33 (7)	300.00 (10)	300.00 (10)	213.89 (5)	232.85 (6)
15×4 (10)	6	300.00 (10)	300.00 (10)	300.00 (10)	300.00 (10)	300.00 (10)	285.36 (9)	296.10 (9)
16×4 (10)	7	300.00 (10)	300.00 (10)	300.00 (10)	300.00 (10)	300.00 (10)	300.00 (10)	300.00 (10)

Table 5.5: Conformant Planning – Benchmark Results

base on the right:

$$\begin{aligned} \mathit{birds}(X) &\leftarrow \mathit{DL}[\mathit{Bird}](X) & \mathit{Flier} &\sqsubseteq \neg \mathit{NonFlier} \\ \mathit{flies}(X) &\leftarrow \mathit{birds}(X), \text{not } \mathit{neg\_flies}(X) & \mathit{Penguin} &\sqsubseteq \mathit{Bird} \\ \mathit{neg\_flies}(X) &\leftarrow \mathit{birds}(X), \mathit{DL}[\mathit{Flier} \uplus \mathit{flies}; \neg \mathit{Flier}](X) & \mathit{Penguin} &\sqsubseteq \mathit{NonFlier} \end{aligned}$$

This encoding realizes the classic Tweety bird example using DL-atoms (which is an alternative syntax for external atoms in this example and allows to express queries over description logics in a more accessible way). The ontology states that Flier is disjoint with $\mathit{NonFlier}$, and that penguins are birds and do not fly; the rules express that birds fly by default, i.e., unless the contrary is derived. The program amounts to the Ω-*transformation* of default logic over ontologies to dl-programs [Dao-Tran et al., 2009b] (not to be confused with the Ω encoding of the unfounded set search from Section 3.2), where the last rule ensures consistency of the guess with the DL ontology. If the assertional part of the DL knowledge base contains $\mathit{Penguin}(\mathit{tweety})$,

then $\mathit{flies}(\mathit{tweety})$ is inconsistent with the given DL-program ($\mathit{neg\_flies}(\mathit{tweety})$ is derived by monotonicity of DL atoms and $\mathit{flies}(\mathit{tweety})$ loses its support). Note that defaults cannot be encoded in standard (monotonic) description logics. Instead, cyclic interaction of DL-rules and the DL knowledge base is necessary.

As all individuals appear in the extension of the predicate flies, all of them are considered simultaneously. This requires a guess on the ability to fly for each individual and a subsequent check, leading to a combinatorial explosion. Intuitively, however, the property can be determined for each individual independently. Hence, a query may be split into independent subqueries, which is achieved by our learning function for *linear sources*, cf. Example 23. The learned nogoods are smaller and more candidate models are eliminated. Table 5.6 shows the runtime for different numbers of individuals. The runs with EBL exhibit a significant speedup, as they exclude many model candidates; here, most of the time is spent calling the description logic reasoner and not for the evaluation of the logic program. The runs with unfounded set checking instead of the explicit FLP check do not show a further speedup because there is only one candidate answer set for each instance, which is not enough to benefit from UFS checking. Also UFL and the switch to encoding Ω do not lead to further performance improvements because of the same reason.

The findings carry over to large ontologies (DL knowledge bases) used in real-world applications. We did similar experiments with a scaled version of the wine ontology (http://kaon2.semanticweb.org/download/test_ontologies.zip). The instances differ in the size of the ABox (ranging from 247 individuals in wine_0 to 79287 in wine_10) and in several other parameters (e.g., on the number of concept inclusions and concept equivalences; Motik and Sattler (2006) describe the particular instances wine_i for all $1 \leq i \leq 10$). We implemented a number of default rules using an analogous encoding as above: e.g., wines not derivable to be dry are not dry, wines which are not sweet are assumed to be dry, wines are white by default unless they are known to be red. Here, we discuss the results of the latter scenario. The experiments classified the wines in the 34 main concepts of the ontology (the immediate subconcepts of the concept Wine, e.g., $\mathit{DessertWine}$ and $\mathit{ItalianWine}$), which have varying numbers of known concept memberships (e.g., ranging from 0 to 43, and 8 on average, in wine_0) and percentiles of red wines among them (from 0% to 100%, and 47% on average). That is, we have one benchmark instance for each main category C, which computes for each individual wine w in C (i.e., the Abox contains the assertion $C(w)$). whether it is red or white using default-negation and a recursive DL-atom. Each such benchmark instance has exactly one answer set that encodes the classification of all wines in the respective category. The results are summarized in Table 5.7. Again, EBL leads to a significant improvement for most concepts and ontology sizes. E.g., there is a gain for 16 out of the 34 concepts of the wine_0 runs, as EBL can exploit linearity. Furthermore, we observed that 6 additional instances can be solved within the 300 seconds time limit. If a concept could be classified both with and without EBL, we could observe a gain of up to 33.02 (on average 6.93). As expected, larger categories profit more from EBL as we can reuse learned nogoods in these instances. As before, UFL does not have a significant impact because there is only one minimality check for each instance.

Besides Ω, Dao-Tran et al. (2009b) describe other transformations of default rules over description logics. Experiments with these transformations revealed that the structure of the result-

#cnt	explicit	+EBL	UFS Γ mol.	UFS Γ mod.	+EBL	+UFL	UFS Ω +EBL+UFL
1 (1)	0.47 (0)	0.50 (0)	0.48 (0)	0.49 (0)	0.49 (0)	0.48 (0)	0.48 (0)
2 (1)	0.57 (0)	0.49 (0)	0.57 (0)	0.55 (0)	0.48 (0)	0.48 (0)	0.51 (0)
3 (1)	0.70 (0)	0.55 (0)	0.75 (0)	0.74 (0)	0.53 (0)	0.54 (0)	0.50 (0)
4 (1)	1.17 (0)	0.48 (0)	1.17 (0)	1.17 (0)	0.48 (0)	0.47 (0)	0.58 (0)
5 (1)	2.57 (0)	0.61 (0)	2.68 (0)	2.65 (0)	0.63 (0)	0.65 (0)	0.60 (0)
6 (1)	4.81 (0)	0.64 (0)	4.59 (0)	4.84 (0)	0.65 (0)	0.65 (0)	0.63 (0)
7 (1)	9.26 (0)	0.69 (0)	9.32 (0)	9.40 (0)	0.66 (0)	0.71 (0)	0.70 (0)
8 (1)	17.68 (0)	0.71 (0)	18.28 (0)	19.30 (0)	0.70 (0)	0.74 (0)	0.70 (0)
9 (1)	39.01 (0)	0.76 (0)	38.59 (0)	39.48 (0)	0.79 (0)	0.75 (0)	0.77 (0)
10 (1)	75.80 (0)	0.86 (0)	72.34 (0)	72.72 (0)	0.86 (0)	0.84 (0)	0.87 (0)
11 (1)	168.96 (0)	0.88 (0)	169.03 (0)	163.63 (0)	0.85 (0)	0.88 (0)	0.91 (0)
12 (1)	300.00 (1)	1.28 (0)	300.00 (1)	300.00 (1)	1.30 (0)	1.28 (0)	1.31 (0)
13 (1)	300.00 (1)	1.38 (0)	300.00 (1)	300.00 (1)	1.30 (0)	1.37 (0)	1.46 (0)
14 (1)	300.00 (1)	1.74 (0)	300.00 (1)	300.00 (1)	1.68 (0)	1.67 (0)	1.67 (0)
15 (1)	300.00 (1)	1.79 (0)	300.00 (1)	300.00 (1)	1.77 (0)	1.79 (0)	1.77 (0)
16 (1)	300.00 (1)	2.94 (0)	300.00 (1)	300.00 (1)	2.95 (0)	2.94 (0)	2.94 (0)
17 (1)	300.00 (1)	3.15 (0)	300.00 (1)	300.00 (1)	3.17 (0)	3.27 (0)	3.16 (0)
18 (1)	300.00 (1)	6.08 (0)	300.00 (1)	300.00 (1)	6.08 (0)	6.15 (0)	6.13 (0)
19 (1)	300.00 (1)	6.67 (0)	300.00 (1)	300.00 (1)	6.48 (0)	6.63 (0)	6.50 (0)
20 (1)	300.00 (1)	14.08 (0)	300.00 (1)	300.00 (1)	14.23 (0)	14.15 (0)	14.11 (0)

Table 5.6: Bird-Penguin – Benchmark Results

ing HEX-programs prohibits an effective reuse of learned nogoods. Hence, the overall picture does not show a significant gain with EBL for these encodings. We could however still observe a small improvement for some runs.

In this scenario, the decision criterion eliminates all unfounded set checks, because all cyclic dependencies over external atoms involve negation and are therefore no cycles according to Definition 45. However, as there is only one compatible set per instance, there would be only one unfounded set check anyway, hence the speedup due to the decision criterion is not significant and not visible in the results. But the effect of the decision criterion can be increased by slightly modifying the scenario such that there are multiple compatible sets. This can be done, for instance, by nondeterministic default classifications, e.g., if a wine is not Italian, then it is either French or Spanish by default. Our experiments have shown that with a small number of compatible sets, the performance enhancement due to the decision criterion is marginal, but increases with larger numbers of compatible sets.

5.2.2 Unfounded Set Checking wrt. Partial Assignments

An unfounded set check wrt. partial assignments, that is sound with respect to any extension to a complete assignment, is possible if the ASP solver has finished unit propagation over a maximal subset of the program such that the interpretation is already complete on it, and all guessed values of external atom replacements are correct. We thus used this criterion, which is easy to test, for a greedy heuristics to issue UFS checks in our prototype system.

However, in contrast to our initial expectation, we found that for all our benchmarks the UFS check wrt. partial assignments was not productive. A closer look reveals that this is essentially because nogood learning from unfounded sets (UFL) effectively avoids the reconstruction of the

ontology	explicit	+EBL	UFS Γ mol.	UFS Γ mod.	+EBL	+UFL	UFS Ω +EBL+UFL
wine_00 (34)	89.30 (9)	33.11 (3)	89.29 (9)	88.75 (9)	33.19 (3)	33.00 (3)	33.15 (3)
wine_01 (34)	188.79 (18)	105.22 (10)	189.51 (18)	188.18 (18)	104.80 (9)	104.59 (10)	105.47 (10)
wine_02 (34)	217.87 (22)	142.67 (14)	217.32 (22)	217.13 (22)	142.79 (14)	142.75 (14)	142.65 (14)
wine_03 (34)	266.10 (30)	183.98 (18)	266.15 (30)	266.14 (30)	183.69 (18)	184.91 (18)	184.73 (18)
wine_04 (34)	266.52 (30)	202.22 (19)	266.47 (30)	266.48 (30)	201.19 (18)	201.46 (19)	201.32 (19)
wine_05 (34)	266.67 (30)	220.87 (21)	266.83 (30)	266.83 (30)	221.21 (21)	221.07 (21)	220.33 (21)
wine_06 (34)	268.03 (30)	258.18 (26)	268.08 (30)	267.98 (30)	257.76 (26)	257.99 (26)	257.96 (26)
wine_07 (34)	271.86 (30)	269.57 (30)	271.97 (30)	272.14 (30)	269.43 (30)	269.45 (30)	269.20 (30)
wine_08 (34)	278.06 (30)	272.57 (30)	278.16 (30)	277.81 (30)	272.37 (30)	272.57 (30)	272.72 (30)
wine_09 (34)	295.35 (31)	282.05 (30)	295.32 (30)	295.35 (31)	282.27 (30)	282.19 (30)	281.87 (30)
wine_10 (34)	300.00 (34)	299.45 (32)	300.00 (34)	300.00 (34)	299.55 (32)	299.63 (32)	299.58 (32)

Table 5.7: Wine Ontology – Benchmark Results

same unfounded set anyway. Therefore, we believe that UFS checking wrt. a partial interpretation rarely identifies an unfounded set earlier than UFS checking wrt. complete assignments. As UFS checking for HEX-programs involves the evaluation of external sources and compatibility testing, this easily leads to costs that are higher than the potential savings. A more detailed analysis requires further studies; since the results do not seem to be promising, we leave this for possible future work.

Tables 5.8, 5.9, 5.10, 5.11 and 5.12 show the benchmark results if UFS checking wrt. partial assignments is enabled when computing all or the first answer set only. We do not show results for the description-logic benchmarks (bird-penguin and wine ontology), because in these benchmarks the decision criterion eliminates unfounded set checks anyway, as described above.

The first column shows the runtime with UFS checking wrt. complete interpretations only, using encoding Ω, EBL and UFL (equivalent to the last column in the tables in Section 5.2.1). The second column shows the results with UFS checking wrt. partial assignments, using a heuristics which performs the UFS check periodically (*periodic*). The third column shows the runtimes if the UFS check is always performed, if no other propagation technique can derive further truth values (*max*).

It can be observed that UFS checking wrt. partial assignments does not lead to a further speadup in any case. Quite to the contrary, some instances have significantly higher runtimes with more frequent unfounded set checks. This is best visible in the set partitioning benchmark (Table 5.8), when computing all explanations for inconsistent MCSs with 5, 6 or 7 contexts (Table 5.10), and when computing all answer sets in the conformant planning benchmark (Table 5.12). In the set partitioning benchmark the effects are especially significant, which is as expected because every compatible set is unfounded-free. Thus, additional UFS checks are always counterproductive. In the consistent multi-context systems, reasoning is fast anyway, thus the frequency of UFS checking has no significant impact (Table 5.9). In the argumentation benchmark we can also observe a slight slowdown by more frequent UFS checking, although it is less dramatic than in the other benchmarks because the other propagation methods are applicable more frequently and thus fewer UFS checks are performed even with setting *max* (Table 5.11).

#ctx	All Answer Sets			First Answer Set		
	Ω +EBL+UFL	Ω partial (periodic) +EBL+UFL	Ω partial (max) +EBL+UFL	Ω +EBL+UFL	Ω partial (periodic) +EBL+UFL	Ω partial (max) +EBL+UFL
1 (1)	0.04 (0)	0.04 (0)	0.04 (0)	0.04 (0)	0.04 (0)	0.05 (0)
2 (1)	0.04 (0)	0.05 (0)	0.05 (0)	0.04 (0)	0.04 (0)	0.05 (0)
3 (1)	0.05 (0)	0.05 (0)	0.07 (0)	0.05 (0)	0.04 (0)	0.05 (0)
4 (1)	0.06 (0)	0.07 (0)	0.08 (0)	0.05 (0)	0.05 (0)	0.06 (0)
5 (1)	0.07 (0)	0.09 (0)	0.11 (0)	0.05 (0)	0.06 (0)	0.07 (0)
6 (1)	0.10 (0)	0.13 (0)	0.15 (0)	0.06 (0)	0.07 (0)	0.09 (0)
7 (1)	0.13 (0)	0.15 (0)	0.19 (0)	0.07 (0)	0.08 (0)	0.11 (0)
8 (1)	0.18 (0)	0.20 (0)	0.26 (0)	0.08 (0)	0.10 (0)	0.14 (0)
9 (1)	0.24 (0)	0.26 (0)	0.35 (0)	0.09 (0)	0.12 (0)	0.17 (0)
10 (1)	0.29 (0)	0.33 (0)	0.47 (0)	0.11 (0)	0.14 (0)	0.21 (0)
15 (1)	0.80 (0)	0.96 (0)	1.61 (0)	0.24 (0)	0.38 (0)	0.73 (0)
20 (1)	1.96 (0)	2.46 (0)	4.92 (0)	0.51 (0)	0.97 (0)	2.30 (0)
25 (1)	4.15 (0)	5.52 (0)	11.25 (0)	0.97 (0)	1.98 (0)	4.50 (0)

Table 5.8: Set Partitioning – Benchmark Results with UFS Checking wrt. Partial Assignments

#ctx	Ω +EBL+UFL	Ω partial (periodic) +EBL+UFL	Ω partial (max) +EBL+UFL
3 (6)	0.08 (0)	0.09 (0)	0.10 (0)
4 (10)	0.11 (0)	0.11 (0)	0.12 (0)
5 (8)	0.12 (0)	0.12 (0)	0.13 (0)
6 (6)	0.15 (0)	0.15 (0)	0.16 (0)
7 (12)	0.20 (0)	0.20 (0)	0.21 (0)
8 (5)	0.21 (0)	0.21 (0)	0.22 (0)
9 (8)	0.24 (0)	0.24 (0)	0.27 (0)
10 (11)	0.31 (0)	0.31 (0)	0.32 (0)

Table 5.9: Consistent MCSs – Benchmark Results with UFS Checking wrt. Partial Assignments

#ctx	All Answer Sets			First Answer Set		
	Ω +EBL+UFL	Ω partial (periodic) +EBL+UFL	Ω partial (max) +EBL+UFL	Ω +EBL+UFL	Ω partial (periodic) +EBL+UFL	Ω partial (max) +EBL+UFL
3 (9)	0.14 (0)	0.13 (0)	0.16 (0)	0.09 (0)	0.09 (0)	0.10 (0)
4 (14)	0.61 (0)	0.64 (0)	0.88 (0)	0.13 (0)	0.13 (0)	0.14 (0)
5 (11)	1.28 (0)	1.36 (0)	1.81 (0)	0.16 (0)	0.16 (0)	0.17 (0)
6 (18)	1.55 (0)	1.67 (0)	2.49 (0)	0.18 (0)	0.18 (0)	0.18 (0)
7 (13)	29.47 (0)	31.54 (0)	44.90 (1)	0.17 (0)	0.17 (0)	0.18 (0)
8 (6)	51.13 (1)	51.22 (1)	51.66 (1)	0.19 (0)	0.20 (0)	0.21 (0)
9 (14)	130.56 (6)	130.99 (6)	133.84 (6)	0.28 (0)	0.27 (0)	0.28 (0)
10 (12)	277.05 (11)	277.20 (11)	278.21 (11)	0.34 (0)	0.35 (0)	0.36 (0)

Table 5.10: Inconsistent MCSs – Benchmark Results with UFS Checking wrt. Partial Assignments

#args	All Answer Sets			First Answer Set		
	Ω +EBL+UFL	Ω partial (periodic) +EBL+UFL	Ω partial (max) +EBL+UFL	Ω +EBL+UFL	Ω partial (periodic) +EBL+UFL	Ω partial (max) +EBL+UFL
1 (30)	0.05 (0)	0.05 (0)	0.05 (0)	0.05 (0)	0.05 (0)	0.05 (0)
2 (30)	0.07 (0)	0.06 (0)	0.07 (0)	0.06 (0)	0.07 (0)	0.07 (0)
3 (30)	0.09 (0)	0.09 (0)	0.10 (0)	0.08 (0)	0.08 (0)	0.09 (0)
4 (30)	0.13 (0)	0.14 (0)	0.16 (0)	0.12 (0)	0.12 (0)	0.14 (0)
5 (30)	0.19 (0)	0.20 (0)	0.22 (0)	0.17 (0)	0.16 (0)	0.18 (0)
6 (30)	0.36 (0)	0.36 (0)	0.39 (0)	0.29 (0)	0.29 (0)	0.31 (0)
7 (30)	0.56 (0)	0.56 (0)	0.59 (0)	0.40 (0)	0.40 (0)	0.42 (0)
8 (30)	1.15 (0)	1.15 (0)	1.19 (0)	0.94 (0)	0.94 (0)	0.96 (0)
9 (30)	1.95 (0)	1.94 (0)	2.01 (0)	1.34 (0)	1.35 (0)	1.39 (0)
10 (30)	4.79 (0)	4.80 (0)	4.96 (0)	3.68 (0)	3.67 (0)	3.75 (0)
11 (30)	9.48 (0)	9.49 (0)	9.71 (0)	4.69 (0)	4.71 (0)	4.74 (0)
12 (30)	12.39 (0)	12.42 (0)	12.79 (0)	6.13 (0)	6.11 (0)	6.23 (0)
13 (30)	24.44 (0)	24.45 (0)	25.32 (0)	16.50 (0)	16.46 (0)	16.80 (0)
14 (30)	51.98 (3)	52.03 (3)	52.57 (3)	41.76 (2)	41.80 (3)	41.98 (3)
15 (30)	78.19 (3)	78.14 (3)	79.81 (3)	41.62 (2)	41.53 (2)	42.02 (2)
16 (30)	77.95 (4)	77.99 (4)	79.52 (4)	40.84 (3)	40.79 (3)	41.04 (3)
17 (30)	76.85 (5)	76.86 (5)	77.82 (5)	35.57 (2)	35.53 (2)	35.58 (2)
18 (30)	125.91 (8)	126.17 (8)	128.83 (8)	75.10 (5)	75.32 (5)	75.37 (5)
19 (30)	147.62 (10)	147.51 (10)	149.62 (10)	67.04 (4)	66.88 (4)	67.59 (4)
20 (30)	166.07 (12)	165.96 (12)	168.53 (12)	82.45 (5)	82.27 (5)	82.90 (5)

Table 5.11: Argumentation – Benchmark Results with UFS Checking wrt. Partial Assignments

Map Size	Plan Length	All Answer Sets			First Answer Set		
		Ω +EBL+UFL	Ω partial (periodic) +EBL+UFL	Ω partial (max) +EBL+UFL	Ω +EBL+UFL	Ω partial (periodic) +EBL+UFL	Ω partial (max) +EBL+UFL
3×4 (10)	1	0.14 (0)	0.14 (0)	0.16 (0)	0.06 (0)	0.06 (0)	0.08 (0)
4×4 (10)	1	0.18 (0)	0.17 (0)	0.20 (0)	0.06 (0)	0.06 (0)	0.08 (0)
5×4 (10)	1	0.15 (0)	0.15 (0)	0.18 (0)	0.07 (0)	0.07 (0)	0.09 (0)
6×4 (10)	2	1.35 (0)	1.35 (0)	1.48 (0)	0.13 (0)	0.13 (0)	0.15 (0)
7×4 (10)	2	1.84 (0)	1.83 (0)	2.03 (0)	0.18 (0)	0.18 (0)	0.21 (0)
8×4 (10)	3	13.88 (0)	14.23 (0)	17.27 (0)	1.78 (0)	1.86 (0)	2.36 (0)
9×4 (10)	3	19.62 (0)	19.96 (0)	23.75 (0)	1.16 (0)	1.18 (0)	1.42 (0)
10×4 (10)	4	252.31 (5)	257.18 (5)	289.20 (7)	31.36 (0)	33.40 (0)	49.36 (0)
11×4 (10)	4	209.41 (3)	214.72 (3)	244.84 (5)	25.85 (0)	27.15 (0)	37.64 (0)
12×4 (10)	5	300.00 (10)	300.00 (10)	300.00 (10)	136.64 (4)	137.22 (4)	142.74 (4)
13×4 (10)	5	300.00 (10)	300.00 (10)	300.00 (10)	185.84 (4)	188.73 (4)	209.44 (4)
14×4 (10)	6	300.00 (10)	300.00 (10)	300.00 (10)	232.85 (6)	235.06 (7)	243.73 (7)
15×4 (10)	6	300.00 (10)	300.00 (10)	300.00 (10)	296.10 (9)	297.42 (9)	300.00 (10)
16×4 (10)	7	300.00 (10)	300.00 (10)	300.00 (10)	300.00 (10)	300.00 (10)	300.00 (10)

Table 5.12: Conformant Planning – Benchmark Results with UFS Checking wrt. Partial Assignments

5.2.3 Summary

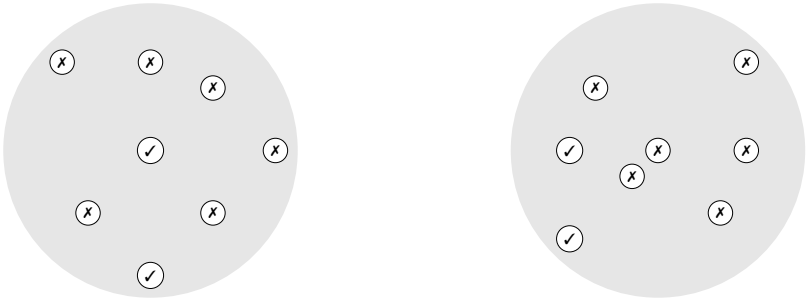
Our experiments have shown that the developed techniques lead to significant performance improvements in most cases, with few exceptions for specific benchmarks. The effects of external behavior learning (EBL) are clearly evident both for the explicit minimality check and for the unfounded set-based check, but are even more prominent with the latter. Independently of whether EBL is used or not, unfounded set checking pushes efficiency of HEX-program evaluation compared to explicit minimality checking. Moreover, it allows for learning of additional nogoods, which is also advantageous in most of our benchmarks. Regarding the two problem encodings, the benchmarks show that the UFS check is usually faster with the Ω encoding than with the Γ encoding, but the former one involves more initialization overhead, which might be counterproductive for small programs.

A detailed analysis has revealed the reasons why the unfounded set-based minimality check is faster than the explicit check. As illustrated in Figure 5.2a, the overall search spaces for smaller models of the reduct and for unfounded sets contain often the same number of candidates, where some of them prove to be indeed smaller models or unfounded sets, respectively, (✓), while others are spurious (✗). This is not surprising, as each unfounded set corresponds to a smaller model and vice versa, cf. Faber et al. (2011). However, part of the search space for unfounded sets is potentially cut off by our optimizations, as shown in Figure 5.2b. Moreover, as the two search spaces are formed by two entirely different search problems, it might be the case that a *true* smaller model resp. unfounded set is found earlier in one of the searches than in the other, see Figure 5.2c. This effect sometimes also favors the explicit minimality check. However, we have also discovered that the search space of the unfounded set-based minimality check can be traversed faster than those of the explicit check and is in this sense 'more compact'. This is explained by the formalism in use: the explicit search space corresponds to an ASP instance whereas the unfounded set search is realized as a SAT instance. As SAT is (in practice) easier than ASP (due to initialization overhead, polynomial unfounded set check in ASP solvers, etc), the search space can be often explored faster in the latter case, even in cases where more candidates need to be investigated before the search can be aborted. In particular, if an answer set has been found, then there is no smaller model of the reduct resp. unfounded set, and the whole search space needs to be traversed in both implementations of the minimality check, see Figure 5.2d; in this case both checks often need to investigate the same number of candidates (if the optimizations do not prune the search space), but the unfounded set search needs less time because of the faster enumeration.

The decision criterion may lead to an additional speedup and does not introduce notable overhead, thus it can always be activated. Finally, program decomposition often leads to an additional performance gain, but should only be used in combination with the decision criterion because otherwise a single UFS check is replaced by multiple UFS checks, which involves more overhead.

Explicit Check
Search for Smaller Models of the Reduct

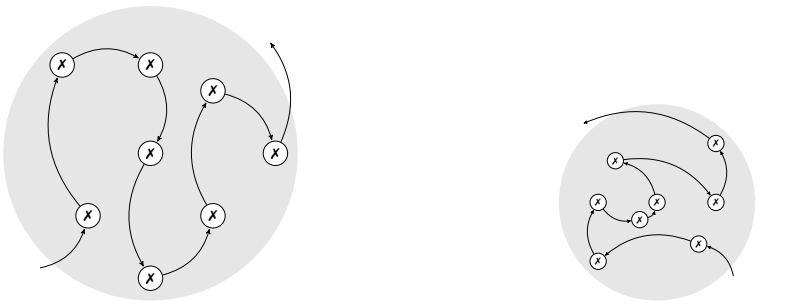
UFS-based Check
Search for Unfounded Sets



(a) Different Search Spaces, but same Number of Candidates

(b) Reduced Search Space due to Optimization

(c) Different Search Problems: One of them may Enumerate Fewer Candidates



(d) Faster Candidate Enumeration in UFS Search

Figure 5.2: Illustration of the Observations

#	With Domain Predicates			Without Domain Predicates		
	wall clock	ground	solve	wall clock	ground	solve
15 (10)	0.59 (0)	0.28 (0)	0.08 (0)	0.49 (0)	0.23 (0)	0.06 (0)
25 (10)	5.78 (0)	4.67 (0)	0.33 (0)	2.94 (0)	1.90 (0)	0.35 (0)
35 (10)	36.99 (0)	33.99 (0)	1.00 (0)	14.02 (0)	11.30 (0)	0.95 (0)
45 (10)	161.91 (0)	155.40 (0)	2.18 (0)	53.09 (0)	47.19 (0)	2.22 (0)
55 (10)	300.00 (10)	300.00 (10)	n/a	171.46 (0)	158.58 (0)	5.74 (0)
65 (10)	300.00 (10)	300.00 (10)	n/a	300.00 (10)	300.00 (10)	n/a

Table 5.13: Reachability – Benchmark Results

5.3 Evaluation of the Grounding Algorithm

For the evaluation of the grounding algorithm, we present for all benchmarks the total wall clock runtime (wt), the grounding time (gt) and the solving time (st). We possibly have $wt \neq gt + st$ because wt includes also computations other than grounding and solving (e.g., passing models through the evaluation graph). As for the benchmarks in Section 5.2, the numbers in parentheses indicate the number of instances and the number of timeouts in the respective categories, and the instances of all benchmarks are available as compressed tar archives from http://www.kr.tuwien.ac.at/staff/redl/aspext. The required plugins are available from the repository (https://github.com/hexhex). For determining de-safety relevant external atoms, our implementation follows a greedy strategy and tries to identify as many external atoms as irrelevant as possible.

Since this section evaluates the grounding algorithm, we set the learning options to the optimal values, i.e., minimality checking is UFS-based using encoding Ω, and EBL and UFL are both enabled. Except for *Argumentation with Subsequent Processing*, which compares two decomposition heuristics, the decomposition heuristics in this section is the greedy heuristics developed in Section 4.5. Note that the set partitioning and the bird-penguin benchmark have already been used for evaluating the learning-based algorithms. However, the results from this section and from the previous section are not directly comparable because different decomposition heuristics are used.

5.3.1 Detailed Benchmark Description

Reachability. We consider reachability, where the edge relation is provided as an external atom $\&out[X](Y)$ delivering all nodes Y that are directly reached from a node X (see Section A.3). The traditional implementation imports all nodes into the program and then uses domain predicates. An alternative is to query outgoing edges of nodes on-the-fly, which needs no domain predicates. This benchmark is motivated by route planning applications, where importing a complete map might be infeasible due to the amount of data.

The results are shown in Table 5.13. We used random graphs with a node count from 5 to 70, an edge probability of 0.25 and the problem encoding from Section A.3. For each node count, we average over 10 instances. Here we can observe that the encoding without domain predicates is more efficient in all cases because only a small part of the map is active in the logic program, which does not only lead to a smaller grounding, but also to a smaller search space during solving.

#	With Domain Predicates			Without Domain Predicates		
	wall clock	ground	solve	wall clock	ground	solve
10 (1)	0.49 (0)	0.01 (0)	0.39 (0)	0.52 (0)	0.02 (0)	0.41 (0)
20 (1)	3.90 (0)	0.05 (0)	3.62 (0)	4.67 (0)	0.10 (0)	4.23 (0)
30 (1)	16.12 (0)	0.18 (0)	15.32 (0)	19.59 (0)	0.36 (0)	18.32 (0)
40 (1)	48.47 (0)	0.48 (0)	46.71 (0)	51.55 (0)	0.90 (0)	48.74 (0)
50 (1)	115.56 (0)	1.00 (0)	112.14 (0)	119.40 (0)	1.79 (0)	114.11 (0)
60 (1)	254.66 (0)	1.84 (0)	248.88 (0)	257.78 (0)	3.35 (0)	248.51 (0)

Table 5.14: Set Partitioning for Liberal Safety – Benchmark Results

Set Partitioning. In this benchmark we consider a program similar to Example 32 with a variable in place of the constant, which implements for each domain element X a choice from $sel(X)$ and $nsel(X)$ by an external atom, i.e., forms a partitioning of the domain into two subsets.

The domain predicate $domain$ is not necessary under de-safety because *&diff* does not introduce new constants. The effect of removing it is presented in Table 5.14. Since *&diff* is monotonic in the first parameter and antimonotonic in the second, the measured overhead is small in the grounding step. Although the ground programs of the strongly safe and the liberally safe variants of the program are identical, the solving step is slower in the latter case; we explain this with caching effects. Grounding liberally de-safe programs needs more memory than grounding strongly safe programs, which might have negative effects on the later solving step because parts of the code have been removed from the memory cache. However, the total slowdown is moderate.

Recursive Processing of Data Structures. This benchmark shows how data structures can be recursively processed. As an example we implement the merge sort algorithm using external atoms for *splitting a list in half* and *merging two sorted lists*, where lists are encoded as constants consisting of elements and delimiters (see Section A.4 for more information about the encoding). However, this is only a showcase and performance cannot be compared to native merge sort implementations.

In order to implement the application with strong safety, one must manually add a domain predicate with the set of all instances of the data structures at hand as extension, e.g., the set of all permutations of the input list. This number is factorial in the input size and thus already unmanageable for very small instances. The problems are both due to grounding and solving. Similar problems arise with other recursive data structures when strong safety is required (e.g., trees, for the pushdown automaton described by Eiter et al. (2013c), where the domain is the set of all strings up to a certain length). However, only a small part of the domain will ever be relevant during computation, hence the grounding algorithm for liberally de-safe programs performs quite well, as shown in Table 5.15.

Bird-Penguin with Nonmonotonic External Atom. We consider now the Bird-Penguin scenario using the DL-plugin as described above. In order to show the behavior of the algorithm in presence of nonmonotonic external atom, we assume that the DL-atom in the above program is

#	With Domain Predicates			Without Domain Predicates		
	wall clock	ground	solve	wall clock	ground	solve
5 (10)	0.22 (0)	0.04 (0)	0.10 (0)	0.10 (0)	0.01 (0)	0.04 (0)
6 (10)	1.11 (0)	0.33 (0)	0.54 (0)	0.10 (0)	0.01 (0)	0.04 (0)
7 (10)	9.84 (0)	4.02 (0)	4.42 (0)	0.11 (0)	0.01 (0)	0.05 (0)
8 (10)	115.69 (0)	61.97 (0)	42.30 (0)	0.12 (0)	0.01 (0)	0.05 (0)
9 (10)	300.00 (10)	300.00 (10)	n/a	0.14 (0)	0.01 (0)	0.07 (0)
10 (10)	300.00 (10)	300.00 (10)	n/a	0.15 (0)	0.08 (0)	0.01 (0)
15 (10)	300.00 (10)	300.00 (10)	n/a	0.23 (0)	0.14 (0)	0.01 (0)
20 (10)	300.00 (10)	300.00 (10)	n/a	0.47 (0)	0.35 (0)	0.02 (0)
25 (10)	300.00 (10)	300.00 (10)	n/a	1.90 (0)	1.58 (0)	0.06 (0)
30 (10)	300.00 (10)	300.00 (10)	n/a	4.11 (0)	3.50 (0)	0.12 (0)
35 (10)	300.00 (10)	300.00 (10)	n/a	20.98 (0)	18.45 (0)	0.51 (0)
40 (10)	300.00 (10)	300.00 (10)	n/a	61.94 (0)	54.62 (0)	1.46 (0)
45 (10)	300.00 (10)	300.00 (10)	n/a	144.22 (2)	133.99 (2)	2.26 (0)
50 (10)	300.00 (10)	300.00 (10)	n/a	300.00 (10)	300.00 (0)	n/a

Table 5.15: Merge Sort – Benchmark Results

#	With Domain Predicates			Without Domain Predicates		
	wall clock	ground	solve	wall clock	ground	solve
5 (1)	0.06 (0)	<0.005 (0)	0.01 (0)	0.08 (0)	0.02 (0)	0.01 (0)
10 (1)	0.14 (0)	<0.005 (0)	0.08 (0)	1.32 (0)	1.12 (0)	0.10 (0)
11 (1)	0.27 (0)	<0.005 (0)	0.19 (0)	2.85 (0)	2.43 (0)	0.27 (0)
12 (1)	0.32 (0)	<0.005 (0)	0.23 (0)	6.05 (0)	5.53 (0)	0.26 (0)
13 (1)	0.69 (0)	0.01 (0)	0.60 (0)	12.70 (0)	11.76 (0)	0.61 (0)
14 (1)	0.66 (0)	<0.005 (0)	0.57 (0)	28.17 (0)	26.70 (0)	0.73 (0)
15 (1)	1.66 (0)	0.01 (0)	1.49 (0)	59.73 (0)	57.14 (0)	1.46 (0)
16 (1)	1.69 (0)	0.01 (0)	1.53 (0)	139.47 (0)	131.87 (0)	1.92 (0)
17 (1)	3.83 (0)	0.01 (0)	3.57 (0)	300.00 (1)	300.00 (1)	n/a
18 (1)	4.34 (0)	0.01 (0)	4.08 (0)	300.00 (1)	300.00 (1)	n/a
19 (1)	10.07 (0)	0.01 (0)	9.56 (0)	300.00 (1)	300.00 (1)	n/a
20 (1)	11.36 (0)	0.01 (0)	10.87 (0)	300.00 (1)	300.00 (1)	n/a
24 (1)	95.60 (0)	0.01 (0)	93.35 (0)	300.00 (1)	300.00 (1)	n/a
25 (1)	300.00 (1)	0.01 (0)	300.00 (1)	300.00 (1)	300.00 (1)	n/a

Table 5.16: Bird-Penguin with Nonmonotonic External Atom and Liberal Safety – Benchmark Results

replaced by a more general external atom which answers the DL-query as usual, but which also returns a dedicated constant $cons$ in addition whenever the updated ontology is consistent; note that this makes the external atom nonmonotonic. Similar consistency checks have been used in some of the encodings by Dao-Tran et al. (2009b).

The results are shown in Table 5.16. The external atom in the updated third rule is nonmonotonic and appears in a cycle, which is the worst case and cannot be avoided by the greedy heuristics in this case. The results show a slowdown for the encoding without domain predicates. It is mainly caused by the grounding, but also solving becomes slightly slower without domain predicates.

Argumentation with Subsequent Processing. This benchmark demonstrates the advantage of our new `greedy` decomposition heuristics, which is compared to the evaluation without

#	monolithic			greedy		
	wall clock	ground	solve	wall clock	ground	solve
4 (30)	0.57 (0)	0.11 (0)	0.38 (0)	0.25 (0)	0.01 (0)	0.18 (0)
5 (30)	2.12 (0)	0.67 (0)	1.26 (0)	0.44 (0)	0.01 (0)	0.37 (0)
6 (30)	18.93 (0)	7.45 (0)	10.86 (0)	0.88 (0)	0.01 (0)	0.80 (0)
7 (30)	237.09 (9)	170.12 (9)	65.12 (0)	1.65 (0)	0.01 (0)	1.57 (0)
8 (30)	300.00 (30)	300.00 (30)	n/a	3.13 (0)	0.01 (0)	3.05 (0)
9 (30)	300.00 (30)	300.00 (30)	n/a	7.41 (0)	0.02 (0)	7.31 (0)
10 (30)	300.00 (30)	300.00 (30)	n/a	15.92 (0)	0.02 (0)	15.81 (0)
11 (30)	300.00 (30)	300.00 (30)	n/a	31.19 (0)	0.02 (0)	31.05 (0)
12 (30)	300.00 (30)	300.00 (30)	n/a	63.16 (0)	0.02 (0)	62.95 (0)
13 (30)	300.00 (30)	300.00 (30)	n/a	172.75 (1)	0.03 (0)	172.38 (1)
14 (30)	300.00 (30)	300.00 (30)	n/a	256.60 (18)	0.01 (0)	256.44 (18)
15 (30)	300.00 (30)	300.00 (30)	n/a	290.01 (29)	<0.005 (0)	290.00 (29)

Table 5.17: Argumentation with Subsequent Processing – Benchmark Results

splitting (`monolithic`). We compute ideal sets for randomized instances of abstract argumentation frameworks [Dung, 1995] of different sizes. Additionally, we perform a processing of the arguments in each extension, e.g., by using an external atom for generating LaTeX code for the visualization of the framework and its extensions (see Section A.5). Without program decomposition, this is the worst case for the grounding algorithm because the code generating atom is nonmonotonic and receives input from the same component. But then the grounding algorithm calls it for exponentially many extensions, although only few of them are actually extensions of the framework.

We used random instances with an argument count from 1 to 15, and an edge probability from $\{0.30, 0.45, 0.60\}$; we used 10 instances for each combination. We can observe that grounding the whole program in a single pass causes large programs wrt. grounding time and size. Since the grounding is larger, also the solving step takes much more time than with our new decomposition heuristics, which avoids the worst case, cf. Table 5.17.

Route Planning. We have implemented two route planning scenarios using the public transport system of Vienna. The data is available under creative commons license (cc-by)[3] and contains a map of 158 subway, tram, city bus and rapid transit train lines with a total number of 1701 stations. Since the data does not contain information about the distances between stations, we uniformly assumed costs of 1, 2 and 3 for each stop traveled by subway/rapid transit train, tram or bus, respectively. We further assumed costs of 10 for each necessary change, representing walking and waiting time. However, with more detailed data, our encoding would also allow for using different values for each line or station. Access to the data is provided via an external atom $\&path[s, d](a, b, c, l)$, which returns for a start location s and a destination d the shortest direct connection (computed using Dijkstra's algorithm), represented as set edges (a, b) between stations a and b with costs c using line l.

In order to model changes between lines, our graph has for each station and each line which arrives at this station a separate node, with a label consisting of the actual name of the station and the respective line. In order to encode a change, the external atom returns a tuple

[3] See data.wien.gv.at.

$(a, a', 10, change)$, where a and a' are two nodes representing the same station but for different lines, and $change$ is just a dedicated 'line' representing walks between platforms. In order to relieve the user from writing line-specific names of stations in the input to the program, we further have for each station a generic node which is connected to all line-specific nodes for this station.

For instance, a journey from $Wien\ Mitte$ to $Taubstummengasse$ is possible using subway line $U4$ from $Wien\ Mitte$ to $Karlsplatz$ (with intermediate stop at *Stadtpark*), changing to subway line *U1*, and going from $Karlsplatz$ to $Taubstummengasse$ (which is just one stop). This will be represented as follows:

$$\big\{(Wien\ Mitte, Wien\ Mitte\ (U4), 10, change),$$
$$(Wien\ Mitte\ (U4), Stadtpark\ (U4), 1, U4),$$
$$(Stadtpark\ (U4), Karlsplatz\ (U4), 1, U4),$$
$$(Karlsplatz\ (U4), Karlsplatz\ (U1), 10, change),$$
$$(Karlsplatz\ (U1), Taubstummengasse\ (U1), 1, U1),$$
$$(Taubstummengasse\ (U1), Taubstummengasse, 10, change)\big\}$$

Note that there will never be cycles in the direct path between two stations because the costs are minimized, thus the set representation is sufficient and there is no need to formally store the order of the edges. Further note that the tuples $(Wien\ Mitte, Wien\ Mitte\ (U4), 10, change)$ and $(Taubstummengasse\ (U1), Taubstummengasse, 10, change)$ are merely the connections between the generic stations and the line-specific nodes, and are actually no real changes. This allows the user to use the constants $Wien\ Mitte$ and $Taubstummengasse$ in the input without predetermining which line to take at these stations. However, as these spurious changes at the start and at the destination node are necessary in any route, this does not affect the minimization of the costs.

Single Route Planning. We now come to the first scenario and consider route planning of a single person who wants to visit a number of locations. Additionally, we have the side constraint that the person wants to go for lunch in a restaurant when the tour is longer than the given limit of costs 300. Because the external source allows only for computing direct connections between two locations, it cannot solve the task completely and there needs to be interaction between the HEX-program and the external source. We considered instances with $1 \leq n \leq 15$ locations to visit.

The sequence in which the locations are visited is guessed non-deterministically in the logic program. While the direct connections between two locations are of minimum length by definition of the external atom, the length of the overall tour is only optimal wrt. to the chosen sequence of locations, but other sequences might lead to a shorter overall tour. However, we have the constraint that for visiting n locations, there should be at most $\lceil n \times 1.5 \rceil$ changes. Due to this constraint not all instances have a solution. It would be easy to extend the scenario to predetermine the sequence of locations by additional constraints, e.g., by global weak constraints in order to minimize the costs.

For each instance size n, we generated 50 instances by randomly drawing n locations to visit *plus* n possible locations for having lunch (the data does not provide information about such locations, but usually there are restaurants or snack bars in the near area of stations). We show

for each instance size the averages of the wall clock runtimes, the grounding times, the solving times, the percentage of instances for which a solution was found within the time limit (column *s.%*)[4], the average path length (costs) of the instances with solutions (column *length*), the average number of necessary changes for instances with solutions, not counting changes between generic and line-specific station nodes (column *changes*), and the percentage of instances with solutions which require a restaurant visit due to length of the tour (column *r.%*). The results are shown in Tables 5.18a, 5.18b and 5.18c using the full map, the map restricted to tram and subway, and the map restricted to subway only, respectively.

The hardness of the benchmark stems from the side constraint concerning lunch. Without this constraint, the tour could be computed deterministically by successive calls of the external source, once the sequence of locations was guessed. However, due to the side constraint, not only the overall tour does depend on the individual locations, but also the individual locations depend on the overall tour (they need to contain a restaurant if the tour is too long and should not contain one otherwise). This leads to a cycle over the external atom $\&path$. With the notion of strong safety, this requires the output variables of this external atom to be bounded by domain predicates, thus the whole map needs to be imported a priori.

Pair-Route Planning. In our second scenario we consider two persons. Each of them wants to visit a number $1 \leq n \leq 15$ of locations (with at most $\lceil n \times 1.5 \rceil$ changes). Additionally, the two persons want to meet, thus the two tours need to intersect at some point. Possible meeting locations are drawn randomly. We further have the side constraint that the meeting location shall be a restaurant, if at least one of the tours is longer than the limit of costs 300. We created for each instance size $1 \leq n \leq 15$ a number of 50 instances, where the n locations for each person, n possible (non-restaurant) meeting locations and n restaurants are drawn randomly. The results are shown in Tables 5.19a, 5.19b and 5.19c using the full map, the map restricted to tram and subway, and the map restricted to subway only, respectively. Columns *length* and *changes* show the sums of the lengths of the tours and of the necessary changes for both persons.

Observations. In both scenarios we can observe that importing the whole map a priori is merely impossible. Already the grounder fails with a timeout, but due to the large number of (unnecessary) external atoms in the ground program, also solving would not be reasonably possible with the given data. Only liberal safety allows for solving the task in the given time limit by importing only the relevant part of the map during grounding. The external atom implements a cache both for the graph representation of the map and for the results of Dijkstra's algorithm. The first call of the external source needs on our benchmark system approximately 5 seconds to load the map (in case of the full map, but not for single route planning with $n = 1$ because there will be no call of the external source). Moreover, Dijkstra's algorithm computes for a given start node the shortest paths to *all* nodes; thus after the external source has been called for a certain start node, successive calls for the same start node are significantly faster. In particular, the cache is already filled with all relevant data during grounding, thus solving will spend only very little time in external sources and the solving time comes mainly from the HEX evaluation algorithms.

For pair-route planning, note that even instances with $n = 1$ have a path longer than 0 because the location for the meeting is not included in the instance size n.

[4]The number of instances for which no solution was found include both timeout instances and instances which have no solution.

#	With Domain Predicates							Without Domain Predicates							
	wall clock	ground	solve	s.%	length	changes	r.%	wall clock	ground	solve	s.%	length	changes	r.%	
1 (50)	300.00 (50)	300.00 (50)	0.00 (0)	0.00	n/a	n/a	n/a	2.40 (0)	1.71 (0)	0.54 (0)	100.00	0.00	0.00	0.00	
2 (50)	300.00 (50)	300.00 (50)	0.00 (0)	0.00	n/a	n/a	n/a	7.82 (0)	5.00 (0)	2.42 (0)	90.00	82.64	2.24	0.00	
3 (50)	300.00 (50)	300.00 (50)	0.00 (0)	0.00	n/a	n/a	n/a	16.44 (0)	9.46 (0)	5.81 (0)	76.00	152.21	3.92	0.00	
4 (50)	300.00 (50)	300.00 (50)	0.00 (0)	0.00	n/a	n/a	n/a	36.60 (0)	16.69 (0)	16.90 (0)	52.00	213.00	5.31	3.85	
5 (50)	300.00 (50)	300.00 (50)	0.00 (0)	0.00	n/a	n/a	n/a	102.71 (0)	26.63 (0)	69.26 (0)	52.00	281.27	7.58	11.54	
6 (50)	300.00 (50)	300.00 (50)	0.00 (0)	0.00	n/a	n/a	n/a	284.69 (38)	236.43 (38)	45.56 (0)	16.00	368.12	9.00	100.00	
7 (50)	300.00 (50)	300.00 (50)	0.00 (0)	0.00	n/a	n/a	n/a	300.00 (50)	300.00 (50)	0.00 (0)	0.00	n/a	n/a	n/a	

(a) Full Map

#	With Domain Predicates							Without Domain Predicates							
	wall clock	ground	solve	s.%	length	changes	r.%	wall clock	ground	solve	s.%	length	changes	r.%	
1 (50)	300.00 (50)	300.00 (50)	0.00 (0)	0.00	n/a	n/a	n/a	1.13 (0)	0.83 (0)	0.20 (0)	100.00	0.00	0.00	0.00	
2 (50)	300.00 (50)	300.00 (50)	0.00 (0)	0.00	n/a	n/a	n/a	3.50 (0)	2.34 (0)	0.85 (0)	100.00	68.12	1.80	0.00	
3 (50)	300.00 (50)	300.00 (50)	0.00 (0)	0.00	n/a	n/a	n/a	8.91 (0)	4.88 (0)	3.01 (0)	96.00	125.19	3.38	0.00	
4 (50)	300.00 (50)	300.00 (50)	0.00 (0)	0.00	n/a	n/a	n/a	34.05 (2)	20.11 (2)	11.41 (0)	92.00	192.80	4.65	0.00	
5 (50)	300.00 (50)	300.00 (50)	0.00 (0)	0.00	n/a	n/a	n/a	62.98 (0)	13.57 (0)	43.58 (0)	90.00	284.89	7.53	46.67	
6 (50)	300.00 (50)	300.00 (50)	0.00 (0)	0.00	n/a	n/a	n/a	192.58 (11)	81.96 (11)	101.66 (0)	74.00	361.86	8.95	100.00	
7 (50)	300.00 (50)	300.00 (50)	0.00 (0)	0.00	n/a	n/a	n/a	287.99 (38)	234.55 (38)	49.27 (0)	24.00	410.17	10.50	100.00	
8 (50)	300.00 (50)	300.00 (50)	0.00 (0)	0.00	n/a	n/a	n/a	299.75 (48)	289.34 (48)	9.48 (0)	4.00	418.00	12.00	100.00	
9 (50)	300.00 (50)	300.00 (50)	0.00 (0)	0.00	n/a	n/a	n/a	300.00 (50)	300.00 (50)	0.00 (0)	0.00	n/a	n/a	n/a	

(b) Subway and Tram

#	With Domain Predicates							Without Domain Predicates							
	wall clock	ground	solve	s.%	length	changes	r.%	wall clock	ground	solve	s.%	length	changes	r.%	
1 (50)	295.42 (49)	294.73 (49)	0.29 (0)	2.00	0.00	0.00	0.00	0.89 (0)	0.65 (0)	0.15 (0)	100.00	0.00	0.00	0.00	
2 (50)	300.00 (50)	300.00 (50)	0.00 (0)	0.00	n/a	n/a	n/a	2.89 (0)	1.92 (0)	0.66 (0)	100.00	64.04	1.22	0.00	
3 (50)	300.00 (50)	300.00 (50)	0.00 (0)	0.00	n/a	n/a	n/a	8.08 (0)	4.10 (0)	2.91 (0)	100.00	127.16	2.62	0.00	
4 (50)	300.00 (50)	300.00 (50)	0.00 (0)	0.00	n/a	n/a	n/a	23.87 (0)	7.31 (0)	13.72 (0)	100.00	206.78	4.16	12.00	
5 (50)	300.00 (50)	300.00 (50)	0.00 (0)	0.00	n/a	n/a	n/a	57.30 (0)	11.69 (0)	39.37 (0)	100.00	317.08	7.20	90.00	
6 (50)	300.00 (50)	300.00 (50)	0.00 (0)	0.00	n/a	n/a	n/a	107.05 (0)	17.32 (0)	78.33 (0)	100.00	364.32	8.46	100.00	
7 (50)	300.00 (50)	300.00 (50)	0.00 (0)	0.00	n/a	n/a	n/a	225.16 (9)	73.97 (9)	135.58 (0)	82.00	405.27	9.61	100.00	
8 (50)	300.00 (50)	300.00 (50)	0.00 (0)	0.00	n/a	n/a	n/a	299.90 (48)	289.15 (48)	9.83 (0)	4.00	353.00	9.00	100.00	
9 (50)	300.00 (50)	300.00 (50)	0.00 (0)	0.00	n/a	n/a	n/a	300.00 (50)	300.00 (50)	0.00 (0)	0.00	n/a	n/a	n/a	

(c) Subway Only

Table 5.18: Single Route Planning – Benchmark Results

As expected, a restriction of the map to trams and subway or to subway only usually leads to smaller runtimes. Also the number of changes decreases because multiple tram and especially subway lines have usually more common stations than bus lines. With increasing number of locations to visit, the number of restaurant visits usually increases as well. However, this is not a strict rule and the tables show some exceptions since the locations were drawn randomly and their distance to each other is an important factor.

5.3.2 Summary

Our new grounding algorithm allows for grounding liberally de-safe programs. Instances that can be grounded by the traditional algorithm as well usually require domain predicates to be manually added (which is often cumbersome and infeasible in practice, as for recursive data

#	With Domain Predicates							Without Domain Predicates							
	wall clock	ground	solve	s.%	length	changes	r.%	wall clock	ground	solve	s.%	length	changes	r.%	
1 (50)	300.00 (50)	300.00 (50)	0.00 (0)	0.00	n/a	n/a	n/a	11.60 (0)	9.32 (0)	1.31 (0)	82.00	150.98	4.54	0.00	
2 (50)	300.00 (50)	300.00 (50)	0.00 (0)	0.00	n/a	n/a	n/a	34.20 (0)	26.00 (0)	6.54 (0)	90.00	300.69	8.53	0.00	
3 (50)	300.00 (50)	300.00 (50)	0.00 (0)	0.00	n/a	n/a	n/a	89.21 (0)	53.31 (0)	29.95 (0)	44.00	449.77	11.14	9.09	
4 (50)	300.00 (50)	300.00 (50)	0.00 (0)	0.00	n/a	n/a	n/a	204.73 (4)	107.60 (4)	83.28 (0)	76.00	592.87	15.47	84.21	
5 (50)	300.00 (50)	300.00 (50)	0.00 (0)	0.00	n/a	n/a	n/a	297.29 (44)	278.85 (44)	15.03 (0)	12.00	710.00	17.50	100.00	
6 (50)	300.00 (50)	300.00 (50)	0.00 (0)	0.00	n/a	n/a	n/a	300.00 (50)	300.00 (50)	0.00 (0)	0.00	n/a	n/a	n/a	

(a) Full Map

#	With Domain Predicates							Without Domain Predicates							
	wall clock	ground	solve	s.%	length	changes	r.%	wall clock	ground	solve	s.%	length	changes	r.%	
1 (50)	300.00 (50)	300.00 (50)	0.00 (0)	0.00	n/a	n/a	n/a	8.17 (0)	7.32 (0)	0.44 (0)	98.00	133.76	3.67	0.00	
2 (50)	300.00 (50)	300.00 (50)	0.00 (0)	0.00	n/a	n/a	n/a	23.35 (0)	19.09 (0)	2.78 (0)	100.00	269.54	7.62	0.00	
3 (50)	300.00 (50)	300.00 (50)	0.00 (0)	0.00	n/a	n/a	n/a	59.04 (0)	36.47 (0)	17.64 (0)	98.00	390.37	10.08	0.00	
4 (50)	300.00 (50)	300.00 (50)	0.00 (0)	0.00	n/a	n/a	n/a	147.43 (3)	75.46 (3)	59.60 (0)	94.00	582.55	14.51	89.36	
5 (50)	300.00 (50)	300.00 (50)	0.00 (0)	0.00	n/a	n/a	n/a	255.94 (17)	161.93 (17)	76.12 (0)	66.00	636.55	16.61	100.00	
6 (50)	300.00 (50)	300.00 (50)	0.00 (0)	0.00	n/a	n/a	n/a	300.00 (50)	300.00 (50)	0.00 (0)	0.00	n/a	n/a	n/a	

(b) Subway and Tram

#	With Domain Predicates							Without Domain Predicates							
	wall clock	ground	solve	s.%	length	changes	r.%	wall clock	ground	solve	s.%	length	changes	r.%	
1 (50)	300.00 (50)	300.00 (50)	0.00 (0)	0.00	n/a	n/a	n/a	7.72 (0)	6.97 (0)	0.30 (0)	98.00	124.51	2.45	0.00	
2 (50)	300.00 (50)	300.00 (50)	0.00 (0)	0.00	n/a	n/a	n/a	21.35 (0)	17.69 (0)	2.19 (0)	100.00	251.66	4.94	0.00	
3 (50)	300.00 (50)	300.00 (50)	0.00 (0)	0.00	n/a	n/a	n/a	60.00 (0)	33.79 (0)	21.05 (0)	100.00	375.08	7.46	4.00	
4 (50)	300.00 (50)	300.00 (50)	0.00 (0)	0.00	n/a	n/a	n/a	167.44 (5)	80.97 (5)	74.14 (0)	90.00	565.33	10.98	82.22	
5 (50)	300.00 (50)	300.00 (50)	0.00 (0)	0.00	n/a	n/a	n/a	267.17 (20)	169.57 (20)	81.00 (0)	60.00	627.70	12.30	100.00	
6 (50)	300.00 (50)	300.00 (50)	0.00 (0)	0.00	n/a	n/a	n/a	299.05 (48)	292.38 (48)	4.88 (0)	4.00	640.00	13.00	100.00	
7 (50)	300.00 (50)	300.00 (50)	0.00 (0)	0.00	n/a	n/a	n/a	300.00 (50)	300.00 (50)	0.00 (0)	0.00	n/a	n/a	n/a	

(c) Subway Only

Table 5.19: Pair-Route Planning – Benchmark Results

structures). We can observe that our algorithm does not only relieve the user from writing domain predicates, but in many cases also has a significantly better performance. Nonmonotonic external atoms might be problematic for our new grounding algorithm. However, the worst case can mostly be avoided by our new decomposition heuristics. Some benchmarks show that with the traditional notion of safety, grounding is practically impossible.

5.4 Summary and Future Work

We now give a summary of this chapter and hint some starting points for future work.

5.4.1 Related Work

Our DLVHEX system is mainly based on GRINGO and CLASP from the Potassco suite, which are statically linked to DLVHEX in order to avoid overhead due to interprocess communication. It uses CLASP both as an ASP and as a SAT solver (for unfounded set checking) and exploits its SMT interface for implementing the learning techniques. As an alternative to GRINGO and

CLASP, the reasoner supports also DLV as solver backend. However, the learning techniques are not applicable in this case because DLV does not provide an appropriate interface. We further provide a grounder and solver which were implemented from scratch during the work on this thesis (mainly for testing purposes, due to missing optimizations).

5.4.2 Summary and Future Work

Our experiments use synthetic and application-driven problems. They have shown significant performance improvements due to the techniques developed in this thesis. We could even observe an exponential speedup in many cases, which turns HEX into a practically useful knowledge representation and reasoning formalism also for larger applications. Some problems can virtually not be solved with the traditional evaluation algorithms for HEX. In particular, programs with value invention often violate the criteria of strong safety and adding domain predicates is not only inconvenient, but merely impossible due to the large size of the domain in many applications.

One starting point for future work concerns the algorithms. Our experiments confirm that exploiting application-specific properties of external atoms has a strong influence on efficiency. Thus, the expectation is that if even more properties are exploited, performance can be further improved. As already mentioned in Chapter 3, another issue is the development of heuristics. We have introduced two encodings for unfounded set checking and observe that each of them might be more efficient in some cases. A heuristics for dynamically choosing between the two encodings might be subject to future work. Concerning unfounded set checking over partial interpretations, we have briefly presented two heuristics but came to the conclusion that they are both inferior to the implementation as a post-check, which is roughly the case because the unfounded set check is too expensive and should be done rarely. However, more advanced heuristics may have positive effects on efficiency.

On the implementation side, one possible improvement is an advanced passing of the non-ground program to the GRINGO grounder. Currently, the already parsed program is sent in string format, thus GRINGO needs to parse it again. Sending it in binary format by exploiting GRINGO's internal data structures can avoid the second parsing process and would be a more elegant implementation. However, as the non-ground input program is usually of moderate size and ASP parsers are simple, we do not expect that this leads to notable performance increase.

Currently, DLVHEX is available for Linux-based systems and for (Mac) OS X. Porting the system to Microsoft Windows systems might be a future goal. Although most people in academia have access to Linux-based systems anyway, this could increase the number of potential users. Moreover, a current issue is the simplification of the build and installation process of the system. Currently, the user needs to build the system from source, which is quite complicated for the average user. We want to provide pre-built packages for the most common platforms in the near future.

An important goal is the extension of our benchmark suite. We are continuously working on the realization of further applications which demonstrate the effectiveness of current and future evaluation algorithms.

Chapter 6

Applications and Extensions of HEX-Programs

In this chapter we discuss applications and extensions of HEX-programs. We focus on applications and extensions which emerged during the work on this thesis, or which have been significantly extended in the context of this research. In the latter case we first give an overview about the current state and then point out the improvements which were established during the work on this thesis. Afterwards, some traditional applications are briefly sketched and references to more elaborative discussions are given. Some of the applications already served as benchmarks in Chapter 5. Other applications have not been used as benchmarks because they can be efficiently realized with traditional algorithms for HEX and are thus not suited for showing computational improvements; this is in particular true for most programs without cyclic structure.

6.1 HEX-programs with Existential Quantification

An important feature of external atoms in HEX-programs is *value invention*, i.e., the introduction of new values that do not occur in the program. Such values may also occur in an answer set of a HEX-program, if we have a rule like

$$lookup(X, Y) \leftarrow p(X), \&do\_hash[X](Y),$$

where intuitively, the external atom $\&do\_hash[X](Y)$ generates a hash key Y for the input X and records it in the fact $lookup(X, Y)$. Here, the variable Y can be seen under existential quantification, i.e., as $\exists Y$, where the quantifier is externally evaluated, by taking domain-specific information into account; in the example above, this would be a procedure to calculate the hashkey. Such domain-specific quantification occurs frequently in applications, be it e.g. for built-in functions (just think of arithmetic), the successor of a current situation in situation calculus, retrieving the social security number of a person etc. To handle such quantifiers in ordinary

ASP is cumbersome; they amount to interpreted functions and require proper encoding and/or special solvers.

HEX-programs however provide a uniform approach to represent such domain-specific existentials. The external treatment allows to deal elegantly with data types (e.g., the social security number, or an IBAN[1] of a bank account, or strings and numbers like reals), to respect parameters, and to realize partial or domain-restricted quantification of the form $\exists Y : \phi(X) \supset p(X, Y)$ where $\phi(X)$ is a formula that specifies the domain of elements X for which an existential value needs to exist; clearly, also range-restricted quantification $\exists Y : \psi(Y) \supset p(X, Y)$ that limits the value of Y to elements that satisfy ψ can be conveniently realized.

In general, such value invention leads on an infinite domain (e.g., for strings) to infinite models, which can not be finitely generated. Under suitable restrictions on a program Π, this can be excluded, in particular if a finite portion of the grounding of Π is equivalent to its full, infinite grounding. This is exploited by various notions of *safety* of HEX-programs that generalize safety of logic programs.

Building on the grounding algorithm for liberally de-safe programs from Section 4.3, we can effectively evaluate HEX-programs with domain-specific existentials that fall in this class. In this application, we generalize this algorithm with *domain specific termination hooks*, such that for non-safe programs, a finitely *bounded grounding* is generated. Roughly speaking, such a bounded grounding amounts to domain-restricted quantification $\exists Y : \phi(X) \supset p(X, Y)$ where the domain condition $\phi(X)$ is dynamically evaluated during the grounding, and information about the grounding process may be also considered. This domain-specific termination produces a partial (bounded) grounding of the program, Π', that leads to *bounded models* of the program Π; the idea is that the grounding is faithful in the sense that every answer set of Π' can be extended to a (possibly infinite) answer set of Π, and considering bounded models is sufficient for an application. This may be fruitfully exploited for applications like query answering over existential rules, reasoning about actions, or to evaluate classes of logic programs with function symbols like *FDNC programs* [Eiter and Simkus, 2010]. Furthermore, even if bounded models are not faithful (i.e., may not be extensible to models of the full grounding), they might be convenient e.g. to provide strings, arithmetic, recursive data structures like lists, trees etc, or action sequences of bounded length resp. depth. The point is that the bound does not have to be 'coded' in the program (like the `maxint` in `DLV` to bound the integer range), but can be provided via termination criteria in the grounding, which gives greater flexibility.

6.1.1 HEX-Programs with Domain-Specific Existential Quantification

In this subsection, we consider HEX-programs with *domain-specific existential quantifiers*. This term refers to the introduction of new values in rule bodies which are propagated to the head such that they may appear in the answer sets of a program. Logical existential quantification is strictly an instance of our approach, where only the existence but not the structure of the values is of interest. Instead, in our work also the structure of the introduced values may be relevant and can be controlled by the external atoms. We consider logical existential quantification in Section 6.1.2.

[1] International Bank Account Number

We start by introducing a grounding algorithm for HEX-programs which extends GroundHEX from Section 4.3 by additional *hooks* that support the insertion of application-specific termination criteria. This may be exploited for computing a finite subset of the grounding of non-de-safe HEX-programs, which is sufficient for the considered reasoning task, e.g., for bounded model building. For instance, we discuss *queries* over (positive) programs with (logical) existential quantifiers in Section 6.1.3, which can be answered by computing a finite part of a canonical model.

HEX-Program Grounding

Intuitively, the bounded grounding Algorithm BGroundHEX can be explained as follows. As with Algorithm GroundHEX, program Π is the non-ground input program. Program Π_p is the non-ground ordinary ASP *prototype program*, which is an iteratively updated variant of Π enriched with additional rules. In each step, the *preliminary ground program* Π_{pg} is produced by grounding Π_p using a standard ASP grounding algorithm. Program Π_{pg} converges against a fixpoint from which the final *ground* HEX*-program* Π_g is eventually extracted. Compared to Algorithm GroundHEX, Algorithm BGroundHEX contains the loop at (b) and the check at (f), which introduce two *hooks* Evaluate and Repeat that allow for realizing application-specific termination criteria in the algorithm. They need to be substituted by concrete code fragments depending on the reasoning task. The remaining parts of the algorithm are equivalent to Algorithm GroundHEX, where *all* external atoms are considered to be de-safety-relevant.

We assume that the hooks are substituted by code fragments with access to all local variables. Moreover, the set PIT_i contains the input atoms for which the corresponding external atoms have been evaluated in iteration i. Evaluate decides for a given input atom $r_{inp}^{\&g[\mathbf{Y}](\mathbf{X})}(\mathbf{c})$ whether the corresponding external atom shall be evaluated with input $\mathbf{c}$. This allows for abortion of the grounding even if it is incomplete, which can be exploited for reasoning tasks over programs with infinite groundings where a finite subset of the grounding is sufficient. The second hook Repeat allows for repeating the core algorithm multiple times such that Evaluate can distinguish between input tuples processed in different iterations. Naturally, soundness and completeness of the algorithm cannot be shown in general, but must be done separately for different reasoning tasks with concrete instances for Repeat and Evaluate.

Domain-Specific Existential Quantification in HEX-Programs

We can realize domain-specific existential quantification naturally in HEX-programs by appropriate external atoms that introduce new values to the program. Realizing existentials by external atoms also allows to use constants different from Skolem terms, i.e., data types with a specific semantics. The values introduced may depend on input parameters passed to the external atom.

Example 74. Consider the following rule:

$$iban(B, I) \leftarrow country(B, C), bank(B, N), \&iban[C, N, B](I)$$

Suppose $bank(b, n)$ models financial institutions b with their associated national number n, and $country(b, c)$ holds for an institution b its home country c. Then one can use $\&iban[C, N, B](I)$ to generate an IBAN from the country, the bank name and number. □

Algorithm BGroundHEX

Input: A HEX-program Π
Output: A ground HEX-program Π_g

(a) $\Pi_p = \Pi \cup \{r^a_{inp} \mid a = \&g[\mathbf{Y}](\mathbf{X}) \text{ in } r \in \Pi\}$
Replace all external atoms $\&g[\mathbf{Y}](\mathbf{X})$ in all rules r in Π_p by $e_{r,\&g[\mathbf{Y}]}(\mathbf{X})$
$i \leftarrow 0$
(b) **while** *Repeat()* **do**
  `// remember already processed input tuples at iteration` i
  $i \leftarrow i + 1$
(c) Set $NewInputTuples \leftarrow \emptyset$ and $PIT_i \leftarrow \emptyset$
  repeat
    $\Pi_{pg} \leftarrow GroundASP(\Pi_p)$ `// partial grounding`
(d) `// evaluate all external atoms`
    for $a = \&g[\boldsymbol{Y}](\boldsymbol{X})$ *in a rule* $r \in \Pi$ **do**
      Let $g^{\&g}_{inp}$ be the unique predicate in the head of r^a_{inp}
(e) `// do this wrt. all relevant assignments`
      $\mathbf{A}_{ma} \leftarrow \{\mathbf{T}p(\mathbf{c}) \mid p(\mathbf{c}) \in A(\Pi_{pg}), p \in \mathbf{Y}_m\} \cup \{\mathbf{F}p(\mathbf{c}) \mid p(\mathbf{c}) \in A(\Pi_{pg}), p \in \mathbf{Y}_a\}$
      for $\mathbf{A}_{nm} \subseteq \left\{\mathbf{T}p(\boldsymbol{c}), \mathbf{F}p(\boldsymbol{c}) \;\middle|\; \begin{array}{l} p \in \boldsymbol{Y}_n, \\ p(\boldsymbol{c}) \in A(\Pi_{pg}) \end{array}\right\}$ *s.t.* $\nexists a : \mathbf{T}a, \mathbf{F}a \in \mathbf{A}_{nm}$ **do**
        $\mathbf{A} \leftarrow \left(\mathbf{A}_{ma} \cup \mathbf{A}_{nm} \cup \{\mathbf{T}a \mid a \leftarrow . \in \Pi_{pg}\}\right) \setminus \{\mathbf{F}a \mid a \leftarrow . \in \Pi_{pg}\}$
(f) **for** $\boldsymbol{y} \in \{\boldsymbol{c} \mid g^{\&g}_{inp}(\boldsymbol{c}) \in A(\Pi_{pg})$ *s.t.* *Evaluate*$(g^{\&g}_{inp}(\boldsymbol{c})) = true\}$ **do**
(g) `// add ground guessing rules`
          `// and remember evaluation`
          $\Pi_p \leftarrow \Pi_p \cup \{e_{r,\&g[\mathbf{y}]}(\mathbf{x}) \vee ne_{r,\&g[\mathbf{y}]}(\mathbf{x}) \leftarrow . \mid f_{\&g}(\mathbf{A}, \mathbf{y}, \mathbf{x}) = 1\}$
          $NewInputTuples \leftarrow NewInputTuples \cup \{g^{\&g}_{inp}(\mathbf{c})\}$
    $PIT_i \leftarrow PIT_i \cup NewInputTuples$
  until Π_{pg} *did not change*
(h) $\Pi_g \leftarrow \Pi_{pg}$
Remove input auxiliary rules and external atom guessing rules from Π_g
Replace all $e_{\&g[\mathbf{y}]}(\mathbf{x})$ in Π_g by $\&g[\mathbf{y}](\mathbf{x})$
return Π_g

Here, the structure of the introduced value is relevant, but an algorithm which computes it can be hidden from the user. The introduction of new values may also be subject to additional conditions which cannot be easily expressed in the program.

Example 75. Consider the following rule:

$$lifetime(M, L) \leftarrow machine(M, C), \&lifetime[M, C](L)$$

It informally expresses that each purchased machine m with cost c ($machine(m, c)$) higher

than a given limit has assigned an expected lifetime l ($lifetime(m, l)$) used for fiscal purposes, whereas purchases below that limit are fully tax deductible in the year of acquirement. Then testing for exceeding of the limit might involve real numbers and cannot easily be done in the logic program. However, the external atom can easily be extended in such a way that the value is only introduced if this side constraint holds. □

Also *counting quantifiers* may be realized in this way, i.e., expressing that there exist *exactly* k or *at least* k elements, which is relevant e.g. in description logics. While a direct implementation of existentials requires changes in the reasoner, a simulation using external atoms is easily extensible.

6.1.2 HEX$^{\exists}$-Programs

We now realize the logical existential quantifier as a specific instance of our approach, which can also be written in the usual syntax; a rewriting then simulates it by using external atoms which return dedicated *null values* to represent a representative for the unnamed values introduced by existential quantifiers.

We start by introducing a language for HEX-programs with logical existential quantifiers, called HEX$^{\exists}$*-programs*, as follows.

Definition 82. A HEX$^{\exists}$*-program* is a finite set of rules of form

$$\forall \mathbf{X} \exists \mathbf{Y} : p(\mathbf{X}', \mathbf{Y}) \leftarrow \mathbf{conj}[\mathbf{X}], \tag{6.1}$$

where $\mathbf{X}$ and $\mathbf{Y}$ are disjoint sets of variables, $\mathbf{X}' \subseteq \mathbf{X}$, $p(\mathbf{X}', \mathbf{Y})$ is an atom, and $\mathbf{conj}[\mathbf{X}]$ is a conjunction of default literals or default external literals containing all and only the variables $\mathbf{X}$; without confusion, we also omit $\forall \mathbf{X}$.

Intuitively speaking, whenever $\mathbf{conj}[\mathbf{X}]$ holds for some vector of constants $\mathbf{X}$, then there should exist a vector $\mathbf{Y}$ such that $p(\mathbf{X}', \mathbf{Y})$ holds. Existential quantifiers are simulated by using *new* null values which represent the introduced unnamed individuals. Formally, we assume that $\mathcal{N} \subseteq \mathcal{C}$ is a set of dedicated null values, denoted by ω_i with $i \in \mathbb{N}$, which do not appear in the program. We transform HEX$^{\exists}$-programs to HEX-programs as follows.

Definition 83. For a HEX$^{\exists}$-program Π, let $T_{\exists}(\Pi)$ be the HEX-program with each rule r of form (6.1) replaced by

$$p(\mathbf{X}', \mathbf{Y}) \leftarrow \mathbf{conj}[\mathbf{X}], \mathit{\&exists}^{|\mathbf{X}'|,|\mathbf{Y}|}[r, \mathbf{X}'](\mathbf{Y}),$$

where $f_{\mathit{\&exists}^{n,m}}(\mathbf{A}, r, \mathbf{x}, \mathbf{y}) = 1$ if $\mathbf{y} = \omega_1, \ldots, \omega_m$ is a vector of *fresh, unique null values* for $r, \mathbf{x}$[2], and $f_{\mathit{\&exists}^{n,m}}(\mathbf{A}, r, \mathbf{x}, \mathbf{y}) = 0$ otherwise; if $n = 0$ we may omit it.

Each existential quantifier is replaced by an external atom $\mathit{\&exists}^{|\mathbf{X}'|,|\mathbf{Y}|}[r, \mathbf{X}'](\mathbf{Y})$ of appropriate input and output arity which exploits value invention for simulating the logical existential quantifier similar to the *parsimonious chase* algorithm (see below).

[2]That is, for each rule r and input vector $\mathbf{x}$ there is exactly one vector $\mathbf{y} = \omega_1, \ldots, \omega_m$ of null values that do not occur in the input program.

We call a HEX$^\exists$-program Π liberally de-safe if $T_\exists(\Pi)$ is liberally de-safe. Various notions of cyclicity have been introduced, e.g., by Grau et al. (2012) (who also survey further notions); here we use b_{synsem} from Section 4.2.2.

Example 76. The following set of rules is a HEX$^\exists$-program Π:

$$\begin{aligned} & employee(john); && employee(joe) \\ r_1\colon\ & \exists Y : office(X,Y) \leftarrow && employee(X) \\ r_2\colon\ & room(Y) \leftarrow && office(X,Y) \end{aligned}$$

Then $T_\exists(\Pi)$ is the following de-safe program:

$$\begin{aligned} & employee(john); && employee(joe) \\ r_1'\colon\ & office(X,Y) \leftarrow && employee(X), \&exists^{1,1}[r_1,X](Y) \\ r_2\colon\ & room(Y) \leftarrow && office(X,Y) \end{aligned}$$

Intuitively, each employee X has some unnamed office Y of X, which is a room. The unique answer set of program $T_\exists(\Pi)$ is $\{employee(john), employee(joe), office(john,\omega_1), office(joe,\omega_2), room(\omega_1), room(\omega_2)\}$. □

For grounding de-safe programs, we simply let Repeat test for $i < 1$ and Evaluate return $true$. Explicit model computation is in general infeasible for non-de-safe programs as the grounding and the models might be infinite. The resulting algorithm GroundDESafeHEX always terminates for de-safe programs. For non-de-safe programs, we can support bounded model generation by other hook instantiations. This is exploited e.g. for query answering over cyclic programs, as described next.

One can show that the algorithm indeed computes all models of the input program.

Proposition 6.1. *For de-safe programs* Π, $\mathcal{AS}\big(\mathit{GroundDESafeHEX}(\Pi)\big) \equiv^{pos} \mathcal{AS}(\Pi)$.

Proof. By definition of the hooks, GroundDESafeHEX behaves like Algorithm GroundHEX and the claim follows from Theorem 6. □

6.1.3 Query Answering over Positive HEX$^\exists$-Programs

The basic idea for query answering over programs with possibly infinite models is to compute a ground program with a single answer set that can be used for answering the query. Positive programs with existential variables are essentially grounded by simulating the *parsimonious chase procedure* in the form used by Leone et al. (2012), which uses null values for each existential quantification. However, for termination we need to use specific instances of the hooks in Algorithm BGroundHEX.

We start by restricting the discussion to a fragment of HEX$^\exists$-programs, called $Datalog^\exists$-programs [Leone et al., 2012].

Definition 84. A $Datalog^\exists$-program is a HEX$^\exists$-program where every rule body $\mathbf{conj}[\mathbf{X}]$ consists of positive ordinary atoms.

Thus, compared to HEX$^{\exists}$-programs, default negation and external atoms are excluded.

Example 77. The following set of rules is a $Datalog^{\exists}$-program:

$$\begin{array}{rl} & person(john); \quad person(joe) \\ r_1\colon & \exists Y : father(X,Y) \leftarrow person(X) \\ r_2\colon & person(Y) \leftarrow father(X,Y) \end{array}$$

□

Next, we define *homomorphisms* for use in $Datalog^{\exists}$-semantics and query answering over $Datalog^{\exists}$-programs.

Definition 85. A *homomorphism* is a mapping $h\colon \mathcal{N} \cup \mathcal{V} \to \mathcal{C} \cup \mathcal{V}$.

For a homomorphism h, let $h|_S$ be its restriction to $S \subseteq \mathcal{N} \cup \mathcal{V}$, i.e., $h|_S(X) = h(X)$ if $X \in S$ and let it be undefined otherwise. For any atom a, let $h(a)$ be the atom where each variable and null value V in a is replaced by $h(V)$; this is likewise extended to $h(S)$ for sets S of atoms and/or vectors of terms. A homomorphism h is a *substitution*, if $h(N) = N$ for all $N \in \mathcal{N}$. An atom a is *homomorphic* (*substitutive*) to atom b, if some homomorphism (substitution) h exists such that $h(a) = b$. An isomorphism between two atoms a and b is a bijective homomorphism h s.t. $h(a) = b$ and $h^{-1}(b) = a$.

A set M of atoms is a model of a $Datalog^{\exists}$-program Π, denoted $M \models \Pi$, such that whenever $h\big(B(r)\big) \subseteq M$ for some substitution h and $r \in \Pi$ of form (6.1), then $h|_{\mathbf{X}}\big(H(r)\big)$ is substitutive to some atom in M; the set of all models of Π is denoted by $mods(\Pi)$.

Next, we can introduce queries over $Datalog^{\exists}$-programs.

Definition 86. A *conjunctive query* (CQ) q is an expression of form $\exists \mathbf{Y} : \leftarrow \mathbf{conj}[\mathbf{X} \cup \mathbf{Y}]$, where $\mathbf{Y}$ and $\mathbf{X}$ (the free variables) are disjoint sets of variables and $\mathbf{conj}[\mathbf{X} \cup \mathbf{Y}]$ is a conjunction of ordinary atoms containing all and only the variables $\mathbf{X} \cup \mathbf{Y}$.

The answer of a CQ q with free variables $\mathbf{X}$ wrt. a model M is defined as follows:

$$ans(q, M) = \left\{ h|_{\mathbf{X}} \;\middle|\; \begin{array}{l} h \text{ is a substitution such that for all atoms } a \text{ in } q, \\ \text{it holds that } h|_{\mathbf{X}}(a) \text{ is substitutive to some } a' \in M \end{array} \right\}$$

Intuitively, this is the set of assignments to the free variables such that the query holds wrt. the model. The answer of a CQ q wrt. a program Π is then defined as the set

$$ans(q, \Pi) = \bigcap_{M \in mods(\Pi)} ans(q, M).$$

Query answering can be carried out over some *universal model* U of the program that is embeddable into each of its models by applying a suitable homomorphism. Formally,

Definition 87. A model U of a program Π is called *universal* if, for each $M \in mods(\Pi)$, there is a homomorphism h s.t. $h(U) \subseteq M$.

The universal model uses null values for unnamed individuals introduced by existential quantifiers. It is then used to answer the query using the following result, which was first shown by Fagin et al. (2005); we use the formalization of Leone et al. (2012):

Proposition 6.2 (Proposition 2.4 by Leone et al. (2012))**.** *Let U be a universal model of some $Datalog^{\exists}$-program Π. Then for each CQ q, $h \in ans(q, \Pi)$ if $h \in ans(q, U)$ and $h\colon\ \mathcal{V} \to \mathcal{C} \setminus \mathcal{N}$.*

Intuitively, the set of all answers to q wrt. U which map all variables to non-null constants is exactly the set of answers to q wrt. Π.

Example 78. Let Π be the program from Example 77. Then the CQ $\exists Y : \leftarrow person(X), father(X, Y)$ asks for all persons who have a father. A universal model is $U = \{person(john), person(joe), father(john, \omega_1), father(joe, \omega_2), person(\omega_1), person(\omega_2), \ldots\}$ and $ans(q, \Pi)$ contains the answers $h_1(X) = john$ and $h_2(X) = joe$. □

Thus, computing a universal model is a key issue for query answering. A common approach for this step is the chase procedure [Johnson and Klug, 1984]. Intuitively, it starts from an empty interpretation and iteratively adds the head atoms of all rules with satisfied bodies, where existentially quantified variables are substituted by fresh nulls. However, in general this procedure does not terminate. Thus, a restricted *parsimonious chase procedure* was used by Leone et al. (2012), which derives less atoms, and which is guaranteed to terminate for the class of *Shy-programs*. Moreover, it was shown that the interpretation computed by the parsimonious chase procedure is, although not a model of the program in general, still sound and complete for query answering, and a *bounded model* in our view.

For query answering over $Datalog^{\exists}$-programs we reuse the translation in Section 6.1.2.

Example 79. Consider the $Datalog^{\exists}$-program Π and its HEX translation $T_{\exists}(\Pi)$:

Π:

$$person(john); \qquad person(joe)$$
$$\exists Y : father(X, Y) \leftarrow person(X)$$
$$person(Y) \leftarrow father(X, Y)$$

$T_{\exists}(\Pi)$:

$$person(john); \qquad person(joe)$$
$$father(X, Y) \leftarrow person(X), \&exists^{1,1}[r_1, X](Y)$$
$$person(Y) \leftarrow father(X, Y)$$

Intuitively, each person X has some unnamed father Y of X who is also a person. □

Note that $T_{\exists}(\Pi)$ is *not* de-safe in general. However, with the hooks in Algorithm BGround-HEX we can still guarantee termination. Let $\mathsf{GroundDatalog}^{\exists}(\Pi, k) = \mathrm{BGroundHEX}\big(T_{\exists}(\Pi)\big)$ with Repeat testing $i < k + 1$, where k is the number of existentially quantified variables in the query, and $\mathsf{Evaluate}(x) = true$ if atom x is *not* homomorphic to any $a \in PIT_i$, and $\mathsf{Evaluate}(x) = false$ otherwise.

The produced program has a single answer set, which essentially coincides with the result of pChase [Leone et al., 2012] that in turn can be used for query answering. Thus, query answering over Shy-programs is reduced to grounding and solving of a HEX-program. More formally:

Proposition 6.3. *For a shy program* Π*, the program produced by* GroundDatalog$^{\exists}(\Pi, k)$ *has a unique answer set which is sound and complete for answering CQs with up to* k *existential variables against* Π*.*

Proof. See Appendix B, page 229.

The main difference to pChase by Leone et al. (2012) is essentially where the homomorphism check is done. pChase instantiates existential variables in rules with satisfied body to new null values only if the resulting head atom is not homomorphic to an already derived atom. In contrast, our method performs the homomorphism check for the input to the $\&exists^{n,m}$ atoms. Thus, homomorphisms are detected when constants are cyclically sent to the external atom. Consequently, our algorithm may need one iteration more than pChase, but allows for reusing the basic idea of the basic algorithm from Chapter 4.

Example 80. For the program and query from Example 79, the algorithm computes a program with answer set $\{person(john), person(joe), father(john, \omega_1), father(joe, \omega_2),\ person(\omega_1), person(\omega_2)\}$. In contrast, pChase would stop already earlier with the result $\{person(john), person(joe), father(john, \omega_1), father(joe, \omega_2)\}$ because there is a homomorphism which maps $person(\omega_1), person(\omega_2)$ to $person(john), person(joe)$. □

More formally, one can show that GroundDatalog$^{\exists}(\Pi, k)$ yields for a Shy-program Π a program with a single answer set that is equivalent to pChase$(\Pi, k+1)$ by Leone et al. (2012). Lemma 4.9 by Leone et al. (2012) implies that the resulting answer set can be used for answering queries with k different existentially quantified variables, which proves Proposition 6.3.

While pChase intermingles grounding and computing a universal model, our algorithm cleanly separates the two stages as common in ASP; modularized program evaluation by the solver will however effect such intermingling. We expect this to be advantageous for extending Shy-programs to programs involving both existential quantifiers and other external atoms, which we leave for future work.

6.1.4 HEX-Programs with Function Symbols

In this subsection we show how to process terms with function symbols by a rewriting to de-safe HEX-programs. We will briefly discuss advantages of our approach compared to a direct implementation of function symbols.

We consider HEX-programs, where the arguments X_i for $1 \leq i \leq \ell$ of ordinary atoms $p(X_1, \ldots, X_\ell)$, and the constant input arguments in $\mathbf{Y}$ and the output $\mathbf{X}$ of an external atom $\&g[\mathbf{Y}](\mathbf{X})$ come from a set of *terms* $\mathcal{T}$ that is the least set $\mathcal{T} \supseteq \mathcal{V} \cup \mathcal{C}$ such that $f \in \mathcal{C}, t_1, \ldots, t_n \in \mathcal{T}$ implies $f(t_1, \ldots, t_n) \in \mathcal{T}$.

Following Calimeri et al. (2007), and as already recapitulated in Section 4.6.1, we introduce for each $k \geq 0$ two external predicates $\&compose_k$ and $\&decompose_k$ with $ar_{\text{I}}(\&compose_k) = 1 + k$, $ar_{\text{O}}(\&compose_k) = 1$, and $ar_{\text{I}}(\&decompose_k) = 1$, $ar_{\text{O}}(\&decompose_k) = 1 + k$. We define

$$f_{\&compose_k}(\mathbf{A}, f, X_1, \ldots, X_k, T) = f_{\&decompose_k}(\mathbf{A}, T, f, X_1, \ldots, X_k) = v,$$

with $v = 1$ if $T = f(X_1, \ldots, X_k)$ and $v = 0$ otherwise. Composition and decomposition of function terms can be simulated using these external predicates. Function terms are replaced by new variables and appropriate external atoms with predicate $\&compose_k$ or $\&decompose_k$ are added in rule bodies to compute their values. More formally, we introduce the following rewriting.

Definition 88. For any HEX-program Π with function symbols, let $T_f(\Pi)$ be the HEX-program where each occurrence of a term $t = f(t_1, \ldots, t_n)$ in a rule r is recursively replaced by a new variable V, and if V occurs afterwards in $H(r)$ or the input list of an external atom in $B(r)$, we add $\&compose_n[f, t_1, \ldots, t_n](V)$ to $B(r)$; otherwise (i.e., V occurs afterwards in some ordinary body atom or the output list of an external atom), we add $\&decompose_n[V](f, t_1, \ldots, t_n)$ to $B(r)$.

Intuitively, $\&compose_n$ is used to construct a nested term from a function symbol and arguments, which might be nested terms themselves, and $\&decompose_n$ is used to extract the function symbol and the arguments from a nested term. The translation can be optimized wrt. evaluation efficiency, but we leave this for future work.

Example 81. Consider the HEX-program Π with function symbols and its translation:

$$
\begin{array}{lll}
\Pi\colon & q(z);\ q(y) & \\
& p(f(f(X))) \leftarrow q(X) & \\
& r(X) \leftarrow p(X) & \\
& r(X) \leftarrow r(f(X)) &
\end{array}
\qquad
\begin{array}{lll}
T_f(\Pi)\colon & q(z);\ q(y) & \\
& p(V) \leftarrow q(X), \&compose_1[f, X](U), & \\
& \qquad \&compose_1[f, U](V) & \\
& r(X) \leftarrow p(X) & \\
& r(X) \leftarrow r(V), \&decompose_1[V](f, X) &
\end{array}
$$

Intuitively, $T_f(\Pi)$ first builds $f(f(X))$ for all X on which q holds using two atoms over $\&compose_1$, and then extracts the X from derived $r(f(X))$ facts using a $\&decompose_1$-atom. □

Note that $\&decompose_n$ supports a well-ordering on term depth such that its output has always a strictly smaller depth than its inputs. This is an important property for proving finite groundability of a program by exploiting the TBFs introduced in Section 4.2.2.

Example 82. The $\Pi = \{q(f(f(a)));\ q(X) \leftarrow q(f(X))\}$ is translated to program $T_f(\Pi) = \{q(f(f(a)));\ q(X) \leftarrow q(V), \&decompose_1[V](f, X)\}$. As $\&decompose_1$ supports a well-ordering, the cycle is benign, i.e., it cannot introduce infinitely many values because the nesting depth of terms is strictly decreasing with each iteration. □

The realization of function symbols via external atoms has the advantage that their processing can be controlled (and can in fact also be seen as domain-specific existential quantifiers). For instance, the introduction of new nested terms may be restricted by additional conditions which can be integrated in the semantics of the external predicates $\&compose_k$ and $\&decompose_k$. A concrete example is *data type checking*, i.e., testing whether the arguments of a function term come from a certain domain. In particular, values might also be rejected, e.g., bounded generation up to a maximal term depth is possible. Another example is to compute some of the term

arguments automatically from others, e.g., constructing the functional term $num(7, vii)$ from 7, where the second argument is the Roman representation of the first one.

Another advantage is that the use of external atoms for functional term processing allows for exploiting de-safety of HEX-programs to guarantee finiteness of the grounding. This allows for reusing the expressive framework for domain-expansion safety and does not need safety criteria specific for function terms.

6.2 HEX-Programs with Nested Program Calls

In procedural programming, the idea of calling subprograms and processing their output is in permanent use. Also in functional programming this kind of modularity is popular. This helps reducing development time (e.g., by using third-party libraries), the length of source code, and, last but not least, makes code human-readable. Reading, understanding, and debugging a typical size application written in a monolithic program is cumbersome. In this section we present a subsystem of DLVHEX, which can be used to 'call' HEX-programs from other HEX-programs, called the *called program* and the *host program*, respectively. This allows for a reasoning over the set of answer sets of a different program. For this purpose we will provide several external atoms for evaluating programs, retrieving the predicates which occur in a specific answer set of the program and for retrieving the arguments of a specific atom within an answer set; objects such as answer sets and atoms are identified by handles.

Modular extensions of ASP have been considered for instance by Janhunen et al. (2009) and Eiter et al. (1997) with the aim of building an overall answer set from program modules; however, multiple results of subprograms (as typical for ASP) are respected, and no reasoning about such results is supported. XASP [Swift and Warren, 2012] is an SMODELS interface for XSB-Prolog. This system is related to our work but less expressive, as it is designed for host programs under well-founded semantics. Moreover, our system allows the seamless integration of queries over subprograms with other external sources. Both is important for some applications, e.g., for the MELD belief set merging system [Redl et al., 2011], which require on the one hand choices and on the other hand access to arbitrary external sources in order to query the data sources to be merged. Adding such nesting to available approaches is not easy and requires to adapt systems similar to our approach. Another approach for modularity was presented by Faber and Woltran (2011). It allows answer set programs for reasoning over the answer sets of other programs. However, unlike our approach, they compile both the host and the called program into a single one, which is called *manifold programs*. In contrast, our approach keeps independent programs which interact via external atoms only.

We realized (possibly parameterized) program calls on top of HEX-programs and call it *nested* HEX-*programs*. It is the nature of nested HEX-programs to have multiple programs which reason over the answer sets of each individual subprogram. This can be done in a user-friendly way and enables the user to write purely *declarative* applications consisting of multiple interacting modules. Notably, call results and answer sets are *objects* that can be accessed by identifiers and processed in the calling program. Thus, different from the works of Janhunen et al. (2009) and Eiter et al. (1997) and related formalisms, this enables *(meta)-reasoning about the set of answer sets* of a program. In contrast to the work of Swift and Warren (2012), both the calling

and the called program are in the same formalism. In particular, the calling program has also a declarative semantics. As an important difference to the formalism of Analyti et al. (2011), nested HEX-programs do not require extending the syntax and semantics of the underlying formalism, which is the HEX-semantics. The integration is, instead, by defining external atoms, making the approach simple and user-friendly for many applications. Furthermore, as nested HEX-programs are based on HEX-programs, they additionally provide access to external sources other than logic programs. This makes nested HEX-programs a powerful formalism, which has been implemented using the DLVHEX reasoner for HEX-programs; applications like belief set merging [Redl et al., 2011] show its potential and usefulness. Moreover, we will show how nested programs can be used for external source simulation. This allows for rapid prototyping without actually implementing plugins for the reasoner, which is time-consuming.

Nested HEX-programs are realized as a *set of external atoms* and an *answer cache* for the results of subprograms. They have initially been developed by Redl (2010) and presented in more detail by Redl et al. (2011) and Eiter et al. (2011b) as part of the belief set merging system MELD. However, it turned out that calling subprograms is useful for many applications beyond belief set merging. In particular, during the work on this thesis, the *simulation of external sources* became relevant. Thus, the original plugin was integrated directly into the core of DLVHEX and extended by new features.

When a subprogram call (corresponding to the evaluation of a special external atom) is encountered in the host program, the subsystem for evaluating nested HEX-programs creates another instance of the reasoner to evaluate the subprogram. Its result is then stored in the *answer cache* and identified with a unique *handle*, which can later be used to reference the result and access its components (e.g., predicate names, literals, arguments) via other special external atoms. For economic memory management, the implementation may remove answer cache entries dynamically in the style of a *least frequently used* heuristics, and reevaluate the corresponding program if it is later accessed again.

There are two possible sources for the called subprogram: (1) either it is *directly embedded* in the host program, or (2) it is *stored in a separate file*. In (1), the rules of the subprogram must be represented within the host program. To this end, they are encoded as string constants.

An embedded program must not be confused with a subset of the rules of the host program. Even though the embedded program is syntactically part of the host program, it is logically separated to allow independent evaluation. In (2), merely the *path* to the external program in the file system is given. Compared to embedded subprograms, code can be reused without the need to copy, which is clearly advantageous when the subprogram is changed or extended. This might be used to provide libraries for solving problems which often reoccur as sub-problems in ASP applications, e.g., graph problems or combinatorial optimization problems. For maintenance of the library and the depending programs (e.g., for bug fixing or for extending the library towards a more general type of the problem), it is of great interest to have a clear interface and not to hard-code the sub-programs within the host programs.

We now present external atoms $\&callhex_n$, $\&callhexfile_n$, $\&answersets$, $\&predicates$, and $\&arguments$ which are used to realize nested HEX-programs.

6.2.1 External Atoms for Subprogram Handling

We start with two families of external atoms

$$\&callhex_n[\mathtt{P}, p_1, \ldots, p_n](H) \quad \text{and} \quad \&callhexfile_n[\mathtt{FN}, p_1, \ldots, p_n](H)$$

that allow to execute a subprogram given by a string `P` respectively in a file `FN`; here n is an integer specifying the number of predicate names p_i, $1 \leq i \leq n$, used to define the input facts. When evaluating such an external atom relative to an interpretation $\mathbf{A}$, the system adds all facts $\{p_i(t_1, \ldots, t_\ell) \leftarrow\mid \mathbf{T}p_i(t_1, \ldots, t_\ell) \in \mathbf{A}\}$ to the specified program, creates another instance of the reasoner to evaluate it, and returns a symbolic handle H as result. A *handle* is a unique integer representing a certain program answer cache entry. For convenience, we do not write n in $\&callhex_n$ and $\&callhexfile_n$ as it is understood from the usage.

Example 83. In the following program, we use two predicates p_1 and p_2 to define the input to the subprogram `sub.hex` ($n = 2$), i.e., all atoms over these predicates are added to the subprogram prior to evaluation. The call derives a handle H as result.

$$\begin{array}{rcl} p_1(x, y); & & p_2(a); \quad p_2(b) \\ handle(H) & \leftarrow & \&callhexfile[\mathtt{sub.hex}, p_1, p_2](H) \end{array}$$

In the implementation, handles are consecutive numbers starting with 0. Hence, the unique answer set of the program is $\{\mathbf{T}handle(0)\}$ (neglecting facts). □

Formally, given an interpretation $\mathbf{A}$, $f_{\&callhexfile_n}(\mathbf{A}, file, p_1, \ldots, p_n, h) = v$ with $v = 1$ if h is the handle to the result of the program in file *file* extended by the facts over predicates $p_1, \ldots, p_n$ that are true in $\mathbf{A}$, and $v = 0$ otherwise. The formal notion and use of $\&callhex_n$ to call embedded subprograms is analogous to $\&callhexfile_n$.

Example 84. Consider the following program:

$$\begin{array}{rcl} h_1(H) & \leftarrow & \&callhexfile[\mathtt{sub.hex}](H) \\ h_2(H) & \leftarrow & \&callhexfile[\mathtt{sub.hex}](H) \\ h_3(H) & \leftarrow & \&callhex[\mathtt{a};\ \mathtt{b}](H) \end{array}$$

□

The rules execute the program `sub.hex` and the embedded program $\Pi_e = \{a;\ b\}$, with no facts being added. The single answer set is $\{\mathbf{T}h_1(0), \mathbf{T}h_2(0), \mathbf{T}h_3(1)\}$ or $\{\mathbf{T}h_1(1), \mathbf{T}h_2(1), \mathbf{T}h_3(0)\}$ depending on the order in which the subprograms are executed (which is irrelevant). While $h_1(X)$ and $h_2(X)$ will have the same value for X, $h_3(Y)$ will be such that $Y \neq X$. Our implementation realizes that the result of the program in `sub.hex` is referred to twice but executes it only once; Π_e is (possibly) different from `sub.hex` and thus evaluated separately.

Now we want to determine how many (and subsequently which) answer sets it has. For this purpose, we define external atom $\&answersets[PH](AH)$ which maps handles PH to call results to sets of respective answer set handles. Formally, for an interpretation $\mathbf{A}$, we have $f_{\&answersets}(\mathbf{A}, h_{Prog}, h_{AS}) = v$ with $v = 1$ if h_{AS} is a handle to an answer set of the program with program handle h_{Prog}, and $v = 0$ otherwise.

Example 85. The single rule

$$ash(PH, AH) \quad \leftarrow \quad \&callhex[\mathtt{a} \vee \mathtt{b} \leftarrow](PH), \&answersets[PH](AH)$$

calls the embedded subprogram $\Pi_e = \{a \vee b \leftarrow\}$ and retrieves pairs (PH, PA) of handles to its answer sets. $\&callhex$ returns a handle $PH = 0$ to the result of Π_e, which is passed to $\&answersets$. This atom returns a *set* of answer set handles (0 and 1, as Π_e has two answer sets, viz. $\{\mathbf{T}a, \mathbf{F}b\}$ and $\{\mathbf{F}a, \mathbf{T}b\}$). The overall program has thus the single answer set $\{\mathbf{T}ash(0,0), \mathbf{T}ash(0,1)\}$. As for each program the answer set handles start with 0, only a pair of program and answer set handles uniquely identifies an answer set. □

We are now ready to solve our example of counting shortest paths from above.

Example 86. Suppose `paths.hex` is the search program and encodes each shortest path in a separate answer set. Consider the following program:

$$\begin{aligned} as(AH) \quad &\leftarrow \quad \&callhexfile[\mathtt{paths.hex}](PH), \&answersets[PH](AH) \\ number(D) \quad &\leftarrow \quad as(C), D = C + 1, \mathrm{not}\, as(D) \end{aligned}$$

The second rule computes the first free handle D; the latter coincides with the number of answer sets of `paths.hex` (assuming that some path between the nodes exists). □

At this point we still treat answer sets of subprograms as black boxes. We now define an external atom to investigate them.

Given an interpretation $\mathbf{A}$, $f_{\&predicates}(\mathbf{A}, h_{Prog}, h_{AS}, p, a) = v$ with $v = 1$ if p occurs as an a-ary predicate in the answer set identified by h_{Prog} and h_{AS}, and $v = 0$ otherwise. Intuitively, the external atom maps pairs of program and answer set handles to the predicates names with their associated arities occurring in the according answer set.

Example 87. We illustrate the usage of $\&predicates$ with the following program:

$$\begin{aligned} preds(P, A) \quad \leftarrow \quad &\&callhex[\mathtt{node(a)};\ \mathtt{node(b)};\ \mathtt{edge(a,b)}](PH), \\ &\&answersets[PH](AH), \&predicates[PH, AH](P, A) \end{aligned}$$

It extracts all predicates (and their arities) occurring in the answer of the embedded program Π_e, which specifies a graph. The answer set is $\{\mathbf{T}preds(node, 1), \mathbf{T}preds(edge, 2)\}$ as the answer set of Π_e has atoms with predicate $node$ (unary) and $edge$ (binary). □

The final step to gather all information from the answer of a subprogram is to extract the *literals* and their *parameters* occurring in a certain answer set. This can be done with external atom $\&arguments$, which is best demonstrated with an example.

Example 88. Consider the following program:

$$\begin{aligned} h(PH, AH) \quad &\leftarrow \quad \&callhex[\mathtt{node(a)};\ \mathtt{node(b)};\ \mathtt{node(c)};\ \mathtt{edge(a,b)};\ \mathtt{edge(c,a)}](PH), \\ &\quad\ \&answersets[PH](AH) \\ edge(W, V) \quad &\leftarrow \quad h(PH, AH), \&arguments[PH, AH, \mathtt{edge}](I, 0, V), \\ &\quad\ \&arguments[PH, AH, \mathtt{edge}](I, 1, W) \\ node(V) \quad &\leftarrow \quad h(PH, AH), \&arguments[PH, AH, \mathtt{node}](I, 0, V) \end{aligned}$$

It extracts the directed graph given by the embedded subprogram Π_e and reverses all edges; the answer set is $\{\mathbf{T}h(0,0), \mathbf{T}node(a), \mathbf{T}node(b), \mathbf{T}node(c), \mathbf{T}edge(b,a), \mathbf{T}edge(a,c)\}$. Indeed, Π_e has a single answer set, identified by $PH = 0, AH = 0$; via $\&arguments$ we can access in the second resp. third rule the facts over $edge$ resp. $node$ in it, which are identified by a unique literal id I; the second output term of $\&arguments$ is the argument position, and the third the actual value at this position. If the predicates of a subprogram were unknown, we can determine them using $\&predicates$. □

6.2.2 External Atoms for External Source Prototyping

Our system provides another family of external atoms for rapid prototyping of (simple) external sources directly in ASP. For this purpose, the input-output behavior of hypothetical external sources is encoded by ASP rules. This is useful for quick experiments before a new external source is actually implemented. It comes with less implementation overhead compared to a native implementation in C++. This gives the user the possibility to see how the planned external atom will behave in a program even before it is developed. This application of nested HEX-programs was of great interest throughout the work on this thesis, because the development of the algorithms presented in Chapters 3 and 4 required exhaustive experiments with various types of external sources, and it would have been cumbersome to develop all of them natively in C++.

However, it is clear that the possibility of simulating external sources cannot replace the plugin mechanism of DLVHEX as it cannot access real external sources. Moreover, simulation is less efficient than a native implementation in C++.

For simulation our system supports the external atom:

$$\&simulator_{n,m}[\mathtt{F}, p_1, \ldots, p_n](X_1, \ldots X_m)$$

The simulator atom takes as arguments a filename F, which refers to the ASP program defining the input-output behavior of the prototypical external source, and predicate inputs $p_1, \ldots, p_n$. The output list $X_1, \ldots, X_m$ is used to retrieve the tuples produced by the simulated external source.

When a simulator atom is encountered in the host program, it will evaluate the ASP-program in $\mathtt{F}$ extended by the input atoms defined over predicates $p_1, \ldots, p_n$. In particular, the system will add for all $1 \leq i \leq n$ each input atom $p_i(a_1, \ldots, a_\ell)$ a fact of form $in_i(a_1, \ldots, a_\ell)$ to $\mathtt{F}$. The renaming of the predicates is necessary in order to make the program $\mathtt{F}$ independent of the input predicate names in the host program. The result of $\mathtt{F}$ is expected to consist of exactly one answer set, where all atoms of form $out(o_1, \ldots, o_\ell)$ define the output of the simulated external source.

Example 89. Consider the following program P given by the rules:

$$\begin{array}{rcl} dom(a); & & dom(b); \quad dom(c) \\ sel(X) & \leftarrow & dom(X), \&simulator_{2,1}[\mathtt{Q}, dom, nsel](X) \\ nsel(X) & \leftarrow & dom(X), \&simulator_{2,1}[\mathtt{Q}, dom, sel](X) \end{array}$$

Let further $\mathtt{Q}$ refer to the program:

$$out(X) \quad \leftarrow \quad in_1(X), \text{not } in_2(X)$$

Then $\mathbb{Q}$ simulates an external source which computes the set difference, where the extension of the second predicate input in_2 is subtracted from the extension of the first predicate input in_1. The program P computes then the two sets sel and $nsel$, corresponding to all partitionings of $\{a, b, c\}$ into two subsets. □

6.2.3 Interface for External Source Developers

As an important difference compared to the initial version of nested HEX-programs [Redl, 2010; Redl et al., 2011; Eiter et al., 2011b], the subsystem for subprogram handling is now not only available through special external atoms for the writer of HEX-program, but also as a C++ API for external source developers. That is, the DLVHEX core system offers methods to the developers of plugins which allow them for evaluating subprograms during the evaluation of an external source.

An application of this API is the argumentation benchmark from Section 5.2. The external sources in this benchmark need to check candidate extensions of argumentation frameworks. This check is again encoded as an ASP program, which is realized using the API for nested HEX-programs.

6.2.4 Applications

We conclude this section with a brief discussion of some concrete applications of nested HEX-programs.

MELD

The MELD system deals with merging *collections of belief sets* [Redl, 2010; Redl et al., 2011]. Roughly, a belief set is a set of classical ground literals. Practical examples of belief sets include explanations in abduction problems, encodings of decision diagrams, and relational data. The merging strategy is defined by tree-shaped *merging plans*, whose leaves are the collections of belief sets to be merged, and whose inner nodes are *merging operators* (provided by the user). The structure is akin to syntax trees of terms.

The automatic evaluation of tree-shaped merging plans is based on nested HEX-programs; it proceeds bottom-up, where every step requires inspection of the subresults, i.e., accessing the answer sets of subprograms. The meta program at the root node generates then one answer set for each integrated belief set. For this purpose, guessing rules select an integrated belief set of the top-level merging operator. The meta program then inherits the conclusions of the chosen belief set in order to make it visible to the user. Note that XASP [Swift and Warren, 2012] is thus not appropriate for such unstratified host programs, as it can only compute the well-founded semantics.

Aggregate Functions

Nested HEX-programs can also be used to emulate aggregate functions [Faber et al., 2011] (see e.g. `#sum`, `#count`, `#max`) where the (user-defined) host program computes the function given the result of a subprogram. This can be generalized to aggregates over *multiple* answer sets of the subprogram; e.g., to answer set counting, or to find the minimum/maximum of some predicate over all answer sets (which may be exploited for global optimization).

Generalized Quantifiers

Nested HEX-programs make the implementation of brave and cautious reasoning for query answering tasks very easy, even if the backend reasoner only supports answer set enumeration. Furthermore, extended and user-defined types of query answers (cf. Eiter et al. (1997)) are definable in a very user-friendly way, e.g., majority decisions (at least half of the answer sets support a query), or minimum and/or maximum number based decisions (qualified number restrictions).

Preferences

Answer sets as accessible objects can be easily compared wrt. user-defined preference rules, and used for filtering as well as ranking results (cf. Delgrande et al. (2004)): a host program selects appropriate candidates produced by a subprogram, using preference rules. The latter can be elegantly implemented as ordinary integrity constraints (for filtering), or as rules (possibly involving further external calls) to derive a rank. A popular application are online shops, where the past consumer behavior is frequently used to filter or sort search results. Doing the search via an ASP program, that delivers the matches in answer sets, a host program can reason about them and act as a filter or ranking algorithm.

Nested Programs as a Development Tool for DLVHEX

The further development of our system DLVHEX uses the idea of annotated external sources. That is, known properties like monotonicity and functionality shall be exploited for speeding up the reasoning process. Developing appropriate algorithms and heuristics requires empirical experiments with a variety of external sources. As it would be cumbersome to implement all of them as real plugins to DLVHEX, simulating them via our $\&simulator_{n,m}$ atom is a good alternative.

6.2.5 Improvements

We summarize the improvements to nested HEX-programs which were done during the work on this thesis compared to the initial version of the techniques [Redl, 2010; Redl et al., 2011].

First, input parameters to subprograms, as supported by the external atoms introduced in Section 6.2.1, were not allowed in the initial version. Redl (2010) used nested HEX-programs for evaluating tree-sharped merging plans of the MELD system, which did not require this feature.

However, it is natural to support argument passing if we consider subprogram calls, thus the feature was added.

Next, external atom simulation as illustrated in Section 6.2.2 was also out of the scope of the initial work on the topic. However, this extension is highly relevant in the context of this thesis as it eases experimenting with various external sources significantly.

Finally, while nested HEX-programs in its initial form were implemented as a plugin, the current version is directly embedded in the system core. This step was chosen because program calls are not only useful for the HEX-programmer, but also for the developer of external sources, which sometimes need to solve declarative subproblems during evaluation. Then it is convenient to have an API for solving nested HEX-programs.

6.3 ACTHEX

ACTHEX [Basol et al., 2010] is an extension of HEX-programs which allows for the execution of declaratively scheduled actions. To this end, *action atoms* are introduced to rule heads, which operate on an *environment* and may modify it. The environment can be seen as an abstraction of realms outside the logic program. Thus, in contrast to ASP and HEX-programs, which are stateless, ACTHEX allows for actual modifications of the external environment without wrapping the solver in a procedural language.

Intuitively, the evaluation of an ACTHEX-program is by first evaluating it similar to an ordinary HEX-program and then selecting a set of action atoms based on the answer sets of the program. The associated actions are then executed in a given sequence, and possibly modify the environment.

During the work on this thesis the ACTHEX formalism was extended in various ways. We will now describe the basics of ACTHEX and then point out the improvements.

6.3.1 ACTHEX Syntax

The ACTHEX language uses, besides the signature introduced in Chapter 2, a set $\mathcal{A}$ of *action predicate names*, which are prefixed with $\#$.

Definition 89 (Action Atom)**.** An action atom is of the form

$$a = \#g[Y_1, \ldots, Y_n]\{o, r\}[w : l],$$

where $\#g \in \mathcal{A}$ is an action predicate name, $Y_1, \ldots, Y_n$ is the input list of length n, $o \in \{b, c, c_p\}$ is the *action option* which declares actions as *brave*, *cautious* or *preferred cautious*, respectively, and the optional integer attributes r, w, and l are called *precedence*, *weight*, and *level*, denoted by $prec(a)$, $weight(a)$, and $level(a)$, respectively.

Rules and programs are then defined as in Chapter 2 but may contain action atoms in rule heads.

6.3.2 ACTHEX Semantics

We first give an intuitive overview about the evaluation of an ACTHEX program. Basically, the following steps are performed:

1. Determine the answer sets of Π wrt. to a snapshot of the environment.
2. Select a subset of all answer sets, called *best models*, using an objective function.
3. Select one of the best models using a *best model selector*.
4. For the chosen best model, determine an *execution schedule*, i.e., a sequence of actions.
5. Execute the execution schedule, which yields an updated environment.
6. Iterate the process.

Formally, we introduce the semantics as follows. First, we generalize the semantics of external atoms such that the environment may influence its truth value. To this end, we introduce for a ground external atom $\&g[\mathbf{y}](\mathbf{x})$ with k-ary input and l-ary output a $2{+}k{+}l$-ary Boolean *oracle function* $f_{\&g}$ and say that $\&g[\mathbf{y}](\mathbf{x})$ is true wrt. assignment $\mathbf{A}$ and environment E if $f_{\&g}(\mathbf{A}, E, \mathbf{y}, \mathbf{x}) = 1$. Satisfaction of ordinary and action atoms a is independent of the environment: a is true if $\mathbf{T}a \in \mathbf{A}$ and false if $\mathbf{F}a \in \mathbf{A}$. Satisfaction of ground rules and programs is then naturally defined as in Chapter 2. We denote by $\mathcal{AS}(\Pi, E)$ the set of all answer sets of program Π wrt. environment E.

Next, we define the *best models* of a program.

Definition 90 (Best Models [Schüller, 2012])**.** The *best models* $\mathcal{BM}(P, E) \subseteq \mathcal{AS}(P, E)$ of Π are those answer sets which minimize the following objective function.

Let $AA^g_w(\Pi)$ denote the set of action atoms in $grnd_{\mathcal{C}}(\Pi)$ with explicit weight and level values. Let further

$$w^{\Pi}_{max} = \max_{a \in AA^g_w(\Pi)} weight(a) \quad \text{and} \quad l^{\Pi}_{max} = \max_{a \in AA^g_w(\Pi)} level(a)$$

denote the maximum weight and maximum level over weighted action atoms in $grnd_{\mathcal{C}}(\Pi)$, respectively; and let

$$M^{\Pi}_i(\mathbf{A}) = \big\{ \#b[\mathbf{Y}]\{o, r\}[w : i] \mid \mathbf{T}\#b[\mathbf{Y}]\{o, r\}[w : i] \in \mathbf{A} \big\}$$

denote the set of action atoms true in $\mathbf{A}$ with level i.

An auxiliary function f_{Π} is then recursively defined as follows:

$$\begin{aligned} f_{\Pi}(1) &= 1, \\ f_{\Pi}(n) &= f_{\Pi}(n-1) \cdot \big|AA^g_w(\Pi)\big| \cdot w^{\Pi}_{max} + 1, \qquad \text{for } n > 1. \end{aligned}$$

Given an answer set $\mathbf{A}$, the objective function $H_{\Pi}(\mathbf{A})$ is then defined as

$$H_{\Pi}(\mathbf{A}) = \sum_{i=1}^{l^{\Pi}_{max}} \big(f_{\Pi}(i) \cdot \sum_{a \in M^{\Pi}_i(I)} weight(a) \big).$$

The intuition is that an answer set is in the set of best models, if no other answer set contains only actions with a lower level, and no other answer set which contains only actions on the same level has a smaller weight of all actions. Note that there are in general multiple best models. However, a possibly customized *best model selector*, as introduced by Fink et al. (2013), is used to select a single one. Once a best model has been selected, the *executability* of action atoms is determined as follows.

Definition 91 (Executable Action Atoms)**.** An action atom $a = \#b[\mathbf{y}]\{o, r\}[w : i]$ is *executable* wrt. a best model $\mathbf{A}$, if:

(i) $o = b$ and $\mathbf{T}a \in \mathbf{A}$; or

(ii) $o = c$ and $\mathbf{T}a \in \mathbf{A}'$ for all $\mathbf{A}' \in \mathcal{AS}(\Pi, E)$; or

(iii) $o = c_p$ and $\mathbf{T}a \in \mathbf{A}'$ for all $\mathbf{A}' \in \mathcal{BM}(\Pi, E)$.

Finally, we define a sequence of all actions executable in the selected best model as follows.

Definition 92 (Execution Schedule [Basol et al., 2010; Schüller, 2012])**.** An *execution schedule* $S_{\mathbf{A}}$ for a best model $\mathbf{A}$ is a sequence of all actions executable in $\mathbf{A}$, such that for all pairs of action atoms a, b with $\mathbf{T}a, \mathbf{T}b \in \mathbf{A}$, if $prec(a) < prec(b)$ then a must precede b in $S_{\mathbf{A}}$.

The set of all execution schedules of a best model $\mathbf{A}$ can be formalized as follows:

$$\mathcal{ES}_{\Pi,E}(\mathbf{A}) = \big\{[a_1, \ldots, a_n] \mid prec(a_i) \leq prec(a_j) \text{ for all } 1 \leq i < j \leq n\big\}$$

Although there can be many execution schedules in general, one is usually interested in a single one which is to be actually executed on the environment. For this purpose, customizable *execution schedule builders* have been introduced by Fink et al. (2013), but also some predefined ones are available in the system.

We now illustrate ACTHEX with the following example [Fink et al., 2013].

Example 90. Consider the following ACTHEX-program:

$$\#robot[goto, charger]\{b, 1\}[1:1] \leftarrow \&sensor[bat](low)$$
$$\#robot[clean, kitchen]\{c, 2\}[1:1] \leftarrow night$$
$$\#robot[clean, bedroom]\{c, 2\}[1:1] \leftarrow day$$
$$night \vee day \leftarrow$$

It uses action atom *#robot* to control a robot and an external atom $\&sensor$ to access sensor data. Intuitively, precedence 1 of action atom $\#robot[goto, charger]\{b, 1\}$ should make the robot recharging its battery, if necessary, before cleaning actions. The cleaning action depends on the time of day. □

The effect of executing a ground action $\#b[\mathbf{y}]\{o, r\}[w : i]$ on an environment E is modeled by a $(2+n)$-ary function $f_{\#b}$, where n is the length of $\mathbf{y}$, which computes an updated environment $E' = f_{\#b}(\mathbf{A}, E, \mathbf{y})$. Similarly, for an execution schedule $S = [a_1, \ldots, a_n]$, the *execution outcome* of S in environment E is $EX(S, \mathbf{A}, E) = E_n$, where $E_0 = E$ and $E_{i+1} = f_{\#b}(\mathbf{A}, E_i, \mathbf{y}^i)$, where $\mathbf{y}^i$ is the input vector of action atom a_i. Intuitively, the environment is iteratively updated following the order of actions in the execution schedule.

6.3.3 Applications

We finally discuss some applications of ACTHEX-programs. For a more elaborative discussion we refer to the works of Basol et al. (2010) and Fink et al. (2013).

Action Languages

Action languages, such as the one by Giunchiglia et al. (2004), are used to describe the relations between *actions* and *fluents*, where the latter describe the state of the world at specified times which is modified by the former. It is not surprising that such languages can be captured by ACTHEX, exploiting the precedence attribute of action atoms to model time.

Knowledge-Base Updates

Adding and removing statements is an issue when maintaining knowledge bases and can be modeled by action atoms. This allows the ACTHEX-programmer to reason over knowledge bases and modify them declaratively depending on the current content. This gives raise to various use cases such as belief revision, belief merging or the implementation of observe-think-act cycles [Kowalski and Sadri, 1999].

Iterative Agent Strategies

The iteration feature introduced by Fink et al. (2013) allows for the stepwise computation of solutions to a problem. This might be exploited to implement agent strategies, e.g., for logic games such as Sudoku or Reversi, in an easily extensible way. For instance, an existing Sudoku agent based on ACTHEX adds in each iteration numbers to a cell or excludes them from the set of possible values. It was observed by Fink et al. (2013) that this strategy has potential to solve instances with a size beyond that which is computable in pure ASP.

6.3.4 Improvements

One improvement in the work of Fink et al. (2013) over the one of Basol et al. (2010) is the generalization of the HEX-semantics. While Basol et al. (2010) associated a $1{+}k{+}l$-ary Boolean oracle function to external atoms $\&g[\mathbf{y}](\mathbf{x})$ with k-ary input and l-ary output, Fink et al. (2013) use a $2{+}k{+}l$-ary function to allow the external atom evaluation to depend on the environment. Previously, the environment was only used for action atoms.

Next, as practical applications are usually interested in a single execution, best model selectors and execution schedule builders were introduced by Fink et al. (2013). The system implementation provides some predefined variants (e.g., lexicographical ordering), but allows also for customizing them.

One of the most important improvements is the possibility of *iteration*. That is, after executing an execution schedule, the whole program may be evaluated again on the updated environment. Termination can be controlled by command-line or (with higher priority) by built-in constants, which set either a fixed number of iterations, a maximum execution time, or iteration ad infinitum. Alternatively, the use of dedicated action atoms $\#acthexContinue$ and $\#acthexStop$

is possible which decide about continuation or termination of iteration. If one of these action atoms is true in the selected best model, then the iteration options set by command-line or built-in constants are overruled in case. Moreover, in order to capture dynamic environments, the environment E'_i used as input for the i+1-th iteration may differ from the outcome E_i from iteration i.

6.4 Multi-Context Systems

Multi-context systems (MCSs) [Brewka and Eiter, 2007] are a formalism for interlinking multiple knowledge based systems, which are called *contexts*. The formalism abstracts from the knowledge representation language and identifies the contexts by their accepted *belief sets*. The latter are abstractly defined as collections of elements. More specifically, the definition of the interlinked systems hinges on the notion of a *logic*, i.e., a tuple $L = (\mathbf{KB}_L, \mathbf{BS}_L, \mathbf{ACC}_L)$ consisting of a set $\mathbf{KB}_L$ of well-formed knowledge bases in some knowledge representation formalism (which may be a different one for each logic), a set $\mathbf{BS}_L$ of possible belief sets, and a function $\mathbf{ACC}_L\colon\ \mathbf{KB}_L \rightarrow 2^{\mathbf{BS}_L}$ which assigns to each knowledge base a set of acceptable belief sets.

The interlinking of contexts is described by *bridge rules*. For a sequence of logics $L = (L_1, \ldots, L_n)$, an L_k-bridge rule over L is of form

$$(k : s) \leftarrow (c_1 : p_1), \ldots, (c_j : p_j), \mathrm{not}(c_{j+1} : p_{j+1}), \ldots, \mathrm{not}(c_m : p_m), \tag{6.2}$$

where $1 \leq k \leq n$ and for each $1 \leq i \leq m$ we have $1 \leq c_i \leq n$ and p_i is an element of some belief set of L_{c_i}.

A *context* is then a tuple $C_i = (L_i, kb_i, br_i)$, where $L_i = (\mathbf{KB}_i, \mathbf{BS}_i, \mathbf{ACC}_i)$ is a *logic*, $kb_i \in \mathbf{KB}_i$ and br_i is a set of L_i-bridge rules over $(L_1, \ldots, L_n)$. A multi-context system is a sequence $C = (C_1, \ldots, C_n)$ of contexts.

The semantics of an MCS $M = (C_1, \ldots, C_n)$ is given by *accepted belief states* $S = (S_1, \ldots, S_n)$, which choose for each context L_i a belief set $S_i \in \mathbf{BS}_i$ for $1 \leq i \leq n$. We call a bridge rule of form (6.2) *applicable* wrt. an accepted belief state S, if $p_i \in S_i$ for all $1 \leq i \leq j$ and $p_i \notin S_i$ for all $j + 1 \leq i \leq m$. A belief state is called an *equilibrium* if for all $1 \leq i \leq n$, $S_i \in \mathbf{ACC}_i\big(kb_i \cup h(r) \mid r \in app(br_i, S)\big)$. Here, $app(br_i, S)$ denotes the set of applicable L_i-bridge rules wrt. S and $h(r)$ denotes the head atom of the respective rule which must be derived in context i. Intuitively, an equilibrium is an accepted belief state $S = (S_1, \ldots, S_n)$ such that the accepted belief sets in all contexts also respect the semantics of bridge rules, i.e., for all applicable L_i-bridge rules with head atom s we have $s \in S_i$.

Example 91. Suppose we have two contexts C_1 and C_2 using ASP syntax and semantics as logics s.t. $kb_1 = \{a \vee b \leftarrow\}$ and $kb_2 = \{x \vee y \leftarrow\}$. Let the only bridge rule be $b_1\colon\ (1 : a) \leftarrow (2 : x)$. Then the equilibria are $S^1 = (\{a\}, \{x\})$, $S^2 = (\{a\}, \{y\})$ and $S^3 = (\{b\}, \{y\})$. In contrast, $S^4 = (\{b\}, \{x\})$ is not an equilibrium because b_1 is applicable wrt. S^4 and thus derives a in context C_1, but is not contained in the respective belief set $\{b\}$. □

Besides computing equilibria, an important reasoning task for MCSs is *inconsistency analysis*. That is, for an MCS M with bridge rules br_M without equilibria, a reason for this incon-

sistency shall be computed. To this end, the notion of *inconsistency explanations (IEs)* has been introduced by Eiter et al. (2010). An IE is a pair (E_1, E_2) of bridge rules $E_1 \subseteq br_M, E_2 \subseteq br_M$ with the following intuition. The set E_1 is relevant to cause an inconsistency, i.e., whenever the bridge rules br_M are replaced by E_1, then the system is already inconsistent and it is not possible to get a consistent MCS by adding further bridge rules from br_M. This inconsistency can however be repaired by adding at least one of the bridge rules in E_2 unconditionally, i.e., for a rule of form (6.2) in E_2 one adds $(k : s) \leftarrow$. For a formal discussion of IEs, we refer to Schüller (2012).

Inconsistency explanations can be computed using an encoding as HEX-program developed by Bögl et al. (2010), which we used as a benchmark problem in Section 5.2. The idea is essentially to guess the membership of rules in sets E_1 and E_2 using disjunctive rules and checking each candidate explanation using the saturation programming technique, which is necessary due to the nature of the definition of inconsistency explanations. External atoms are used for checking if a context accepts a certain belief set. The use of external atoms is highly cyclic because of the saturation encoding. This lifts the problem to the second level of the polynomial hierarchy.

6.5 Description Logic Knowledge-Bases

Description logics (DLs) are a knowledge representation formalism which is well-suited for ontologies as used in the Semantic Web [Heflin and Munoz-Avila, 2002] or in medical applications [Hoehndorf et al., 2007]. Intuitively, ontologies represent classes of objects, referred to as *concepts*, and the relations between objects, called *roles*. Concepts and roles correspond to unary and binary predicates in first-order logic, respectively. A *description logic knowledge base* consists of a *Tbox* (*terminological knowledge* or *intensional part*) and an *Abox* (*assertions* or *extensional part*), cf. Baader et al. (2003). The Tbox defines concepts and roles and represents relations between them, whereas the Abox contains specific knowledge about membership of individuals in concepts or pairs of individuals in roles.

Example 92. Let $PhDStudent$, $Student$ and $Professor$ be concepts and $isAssistantOf$ be a role. Then the Tbox may contain the *concept inclusion axiom* $PhDStudent \sqsubseteq Student$, representing that the class of PhD students is a subclass of all students. The Abox contains concept membership assertions like $Professor(smith)$ and $PhDStudent(johnson)$, representing that $smith$ is a professor and $johnson$ a PhD student. A role membership assertion of form $isAssistantOf(johnson, smith)$ represents that $johnson$ is an assistant of professor $smith$. □

Typical reasoning tasks over description logic knowledge bases include concept and role retrieval, i.e., listing all individuals or pairs of individuals which are members of a given concept or role, respectively. In the example above one may ask for all members of $Student$ and expects as answer the individual $johnson$ because he is a $PhDStudent$ and thus, by the terminological knowledge, also a $Student$.

The combination of ontologies and answer set programming is especially valuable as it allows for accessing existing domain knowledge from logic programs. To this end, *DL-programs* have been developed by Eiter et al. (2008) which can be implemented on top of HEX-programs with dedicated external atoms [Dao-Tran et al., 2009b]. The external source features external atoms for concept and role queries. During a query, the set of members of a concept or role can

be extended by additional individuals defined in the ASP program. This allows for advanced reasoning tasks such as default reasoning over description logic knowledge bases. We explain this using the example which was also used as a benchmark problem in Chapter 5.

Example 93. Consider the program on the left and the terminological part of a DL knowledge base on the right. They encode the Tweety bird example:

$$\begin{aligned} \mathit{birds}(X) &\leftarrow \mathit{DL}[\mathit{Bird}](X) \\ \mathit{flies}(X) &\leftarrow \mathit{birds}(X), \text{not } \mathit{neg\_flies}(X) \\ \mathit{neg\_flies}(X) &\leftarrow \mathit{birds}(X), \mathit{DL}[\mathit{Flier} \uplus \mathit{flies}; \neg \mathit{Flier}](X) \end{aligned} \qquad \begin{aligned} \mathit{Flier} &\sqsubseteq \neg \mathit{NonFlier} \\ \mathit{Penguin} &\sqsubseteq \mathit{Bird} \\ \mathit{Penguin} &\sqsubseteq \mathit{NonFlier} \end{aligned}$$

The ontological knowledge expresses that Flier and $\mathit{NonFlier}$ are disjoint, and that penguins are birds and do not fly. The rules express that birds fly by default. Intuitively the encoding works as follows; a detailed description can be found in the work of Dao-Tran et al. (2009b). Suppose the assertional part of the DL knowledge base contains $\mathit{Penguin}(\mathit{tweety})$. Then the truth of $\mathit{DL}[\mathit{Flier} \uplus \mathit{flies}; \neg \mathit{Flier}](\mathit{tweety})$ is guessed by our evaluation algorithm for HEX-programs. Thus, $\mathit{flies}(\mathit{tweety})$ and $\mathit{neg\_flies}(\mathit{tweety})$ are either true or false. However, if $\mathit{flies}(\mathit{tweety})$ is true, then the DL-program becomes inconsistent because $\mathit{Flier} \uplus \mathit{flies}$ extends the concept Flier by the individual tweety, which contradicts the knowledge derivable from $\mathit{Penguin}(\mathit{tweety})$ and the Tbox. As in classical logic, an inconsistent description knowledge base implies everything, thus $\mathit{neg\_flies}(\mathit{tweety})$ is derived. But then $\mathit{flies}(\mathit{tweety})$ loses its support. Therefore the candidate where $\mathit{flies}(\mathit{tweety})$ is true is rejected. In contrast, if $\mathit{flies}(\mathit{tweety})$ is false, then the description logic knowledge base remains consistent and $\mathit{DL}[\mathit{Flier} \uplus \mathit{flies}; \neg \mathit{Flier}](\mathit{tweety})$ and $\mathit{neg\_flies}(\mathit{tweety})$ are true. Thus, the only answer set correctly identifies tweety as a non-flier. □

Since description logics are purely monotonic, default reasoning can only be realized by the interaction of rules and the DL knowledge base with a highly cyclic structure. For more background and a formal discussion, we refer to Dao-Tran et al. (2009b) and Schindlauer (2006).

Another reasoning task for DL-programs is the repair of inconsistencies which might appear due to the interaction of the program with the ontology [Eiter et al., 2013e]. A repair is a set of changes in the program or in the ontology in order to restore consistency and might be computed by a variant of the evaluation algorithms for HEX-programs.

6.6 Route Planning

While there exist numerous commercial and open route planning applications, with the currently most popular one being probably Google Maps[3], the types of supported queries are usually limited. In contrast, an implementation in a declarative formalism like HEX-programs allows for easy addition of side constraints and thus tailoring to very specific use cases.

We have presented two route planning scenarios in Section 5.3 and explained how side constraints may look like. While the application was mainly developed to serve as a benchmark problem, the results show that the techniques from this thesis allow the application to scale to realistic map material. Thus, the application is ready to be used for practical purposes.

[3] https://maps.google.com

6.7 Summary and Future Work

We now summarize the chapter, discuss some relations to other applications from the literature and give a brief outlook on further developments.

6.7.1 Related Work

Comparable to HEX is SAT modulo theories (SMT) [Barrett et al., 2009], which is satisfiability checking with dedicated *theory atoms*. The background theory (e.g., bit vectors) constraints the truth values of theory atoms. However, unless SMT, HEX also supports programs with variables and unfounded set checking. In this sense, HEX relates to SMT like ASP to SAT.

Constraint ASP [Gebser et al., 2009; Ostrowski and Schaub, 2012] is an extension of ASP by constraint atoms. It has been directly implemented in the CLINGCON system, but is in fact a particular instance of HEX and can be translated to plain HEX. A prototypical implementation of this translation has been developed by Stashuk (2013).

6.7.2 Summary and Future Work

HEX-programs have been successfully used to implement a variety of applications. Notably, the concept of external atoms is very expressive and allows for realizing even syntactic extensions of HEX-programs by translation to plain HEX (e.g., aggregation functions). In this sense, HEX is a generic basis for many knowledge representation and reasoning tasks.

We are continuously looking for further applications which can be effectively implemented using HEX-programs. A convincing application suite can not only serve as benchmark suite, but also helps to establish HEX as a popular knowledge representation formalism.

A concrete important topic is the improvement of the constraint ASP realization on top of HEX. The current implementation is not yet comparable to direct implementations, as in the CLINGCON system, because important optimizations are missing (thus the application was not discussed in more detail in this chapter). While user-defined learning functions have already been used by Stashuk (2013), application-specific heuristics for external atom evaluation (cf. Section 5.1.3) are missing, thus effective *theory propagation* (partial assignments and the theory deterministically imply further truth values of constraint atoms) is not possible. Thus, theory propagation is an important part of future extensions.

Chapter 7

Conclusion and Outlook

We now summarize the thesis and recapitulate the main contributions. Afterwards we give an outlook on possible starting points for future work.

7.1 Conclusion

In this thesis we have developed evaluation and grounding algorithms for HEX-programs, which is an extension of ASP with external sources represented by *external atoms*. The traditional evaluation strategy guesses the truth values of external atoms to produce model candidates. Subsequently, each guess is checked against the real semantics of the external atoms. This usually leads to a large number of independent guesses and limits scalability. Therefore, a main goal of this thesis was the development of *genuine* evaluation algorithms which avoid blind guessing and turn HEX-programs into a practically usable knowledge representation formalism.

As in ASP, non-ground HEX-programs are translated into ground programs by a *grounding procedure*. However, unlike in ASP, the grounding of HEX-programs may need to contain constants which are not part of the input program. This is due to *value invention*, i.e., the introduction of new constants into the program by external atoms. Traditionally, strong syntactic safety criteria are used to restrict value invention and guarantee finite groundability. However, these criteria are in many cases unnecessarily restrictive, thus the traditional approach suffers not only scalability but also expressiveness problems. Overcoming them by relaxing the safety criteria and the development of a suitable grounding algorithm was the second main goal of this thesis.

We first developed a new evaluation strategy for ground HEX-programs in Chapter 3, which makes use of advanced learning techniques. Intuitively, our new algorithm learns additional clauses while the search space is traversed, which prevent the algorithm later from running into the same conflicts again. These clauses reflect the behavior of external sources, and the learning technique is thus called *external behavior learning (EBL)*. The basic idea of our algo-

rithm is related to *constraint ASP solving* presented by Gebser et al. (2009), which is realized in the CLINGCON system. External atom evaluation in our algorithm can superficially be seen in place of constraint propagation. However, while Gebser et al. (2009) consider a particular application, we deal with a more abstract interface to external sources. An important difference between CLINGCON and EBL is that the constraint solver is seen as a black box, whereas we exploit known properties of external sources. Moreover, we support *user-defined learning*, i.e., customization of the default construction of learned clauses to incorporate specific knowledge about the sources, as discussed in Section 3.1.2. Another difference is the construction of learned clauses. Constraint ASP has special constraint atoms, which may be contradictory, e.g., $\mathbf{T}(X > 10)$ and $\mathbf{T}(X = 5)$. The learned clauses are sets of constraint literals, which are kept as small as possible. In our algorithm we have usually no conflicts between ground external atoms as output atoms are mostly independent of each other (excepting e.g. functional sources). Instead, we have a strong relationship between the input and the output. This is reflected by clauses which usually consist of (relevant) input atoms and the negation of one output atom. As in constraint ASP solving, the key for efficiency is keeping clauses small.

Ensuring minimality of answer sets is in general non-trivial and requires special attention. We have designed a minimality check which is based on unfounded sets [Faber, 2005], realized as a separate search problem encoded as a SAT instance. We have then shown several optimizations of the basic minimality check and tightly coupled the minimality check and the search for model candidates by *nogood exchanging*, i.e., nogoods learned in one search problem can be reused for the other one. We have then presented a decision criterion which allows for skipping the entire minimality check for certain program classes.

In Chapter 4 we considered programs with variables and value invention in particular. Since naive value invention can lead to infinite groundings and infinite answer sets, it must be restricted by appropriate safety criteria. In this thesis we have replaced the traditional notion of *strong safety* by the less restrictive notion of *liberal domain-expansion safety*, which increases the freedom of the HEX-programmer when modeling a search problem. We have compared our notion to several other notions of safety and concluded that it is strictly more liberal. Our new notion of safety exploits both syntactic and semantic criteria to guarantee the existence of finite groundings. Moreover, it is designed in an extensible fashion such that additional safety criteria can be easily integrated. This also includes application-specific criteria which can be added to the system using a plugin interface.

Based on the relaxed notion of safety, we have designed a novel efficient grounding algorithm. The new algorithm for HEX-program evaluation can ground *any* program which satisfies the new safety criteria, while the traditional grounding algorithm relies on a program decomposition step for subdividing the program into groundable fragments. However, program decomposition is still advantageous in some cases, although not necessary anymore. This gives the designer of evaluation heuristics more freedom, which we exploited when we designed a novel heuristics capturing the insights from this thesis.

Our theoretical research is confirmed by an implementation and promising benchmark results in Chapter 5. We have seen that the learning-based algorithms lead to significant speedup during evaluation because of effective pruning of the search space. In some cases we even observed an exponential speedup. The minimality check based on unfounded sets often results in

an additional speedup, compared to naive minimality checking using an explicit construction of the FLP-reduct. For programs with variables and domain expansion, we experienced that our new grounding algorithm does not only relieve the user from the burden of writing domain predicates manually, but sometimes also speeds up the grounder significantly compared to a pre-computation of the domain. The worst-case scenario for our new grounding algorithm can be effectively avoided using our new evaluation heuristics for the evaluation framework, which has been extended during the work on this thesis.

Finally, we have also briefly discussed some new and some traditional applications of HEX-programs in Chapter 6, including HEX-programs with (domain-specific) existential quantification [Eiter et al., 2013b], HEX-programs with nested program calls [Eiter et al., 2011b], ACTHEX [Fink et al., 2013], multi-context systems [Brewka and Eiter, 2007], and programs with description knowledge bases [Dao-Tran et al., 2009b].

7.2 Outlook

For ground HEX-program evaluation, the identification of further properties for informed learning is an important topic. Our experiments in Chapter 5 confirm that exploiting application-specific properties of external atoms may have strong influence on efficiency. Thus, the expectation is that if even more properties are exploited, then performance can be further improved. Another issue is the development of heuristics for several purposes. First, our algorithm evaluates external atoms whenever their input is complete. However, this is only one possible strategy. It is also possible to delay external atom evaluation although the input is already complete, which may be advantageous for external sources with high computational costs. Second, our algorithm can perform minimality checks for model candidates already during the search. However, the development of a concrete heuristics for deciding when to do such a check, was out of the scope of this thesis. Third, we have introduced two encodings for unfounded set checking. A heuristics for dynamically choosing between the two encodings might also be subject to future work.

For non-ground HEX-programs, the identification of further (syntactic or semantic) properties, which allow for finite groundability, is an issue. Of particular interest are external atoms that provide built-in functions and simulate, in a straightforward manner, function symbols. On the implementation side, further refinement and optimizations are an issue, as well as the setup of a library of term bounding functions (TBFs) which exploit specific properties of concrete external sources. The grounding algorithm may be extended in the future such that the worst-case can be avoided in more cases. Also other optimizations to the algorithm are possible, e.g., by reusing previous results of the grounding step instead of iterative regrounding of the whole program. Moreover, the new evaluation heuristics for the (extended) evaluation framework may be refined. In the long term, a tighter integration of the grounding and the solving algorithm is desirable. This might be realized as interleaved grounding and solving phases, which reason over an incrementally extended program. The algorithms from this thesis pave the way for future development in this direction.

Bibliography

[Analyti et al., 2011] Analyti, A., Antoniou, G., and Damasio, C. V. (2011). MWeb: a principled framework for modular web rule bases and its semantics. *ACM Transactions on Computational Logic*, 12(2):17:1–17:46.

[Antoniou et al., 2007] Antoniou, G., Baldoni, M., Bonatti, P. A., Nejdl, W., and Olmedilla, D. (2007). *Secure Data Management in Decentralized Systems*, volume 33 of *Advances in Information Security*, chapter Rule-based Policy Specification, pages 1–57. Springer.

[Asparagus Website, 2014] Asparagus Website (2014). http://asparagus.cs.uni-potsdam.de/.

[Baader et al., 2003] Baader, F., Calvanese, D., McGuinness, D. L., Nardi, D., and Patel-Schneider, P. F. (2003). *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, New York, NY, USA.

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

[Barrett et al., 2009] Barrett, C., Sebastiani, R., Seshia, S. A., and Tinelli, C. (2009). *Satisfiability Modulo Theories*, chapter 26, pages 825–885. Volume 185 of [Biere et al., 2009].

[Bartholomew and Lee, 2010] Bartholomew, M. and Lee, J. (2010). A decidable class of groundable formulas in the general theory of stable models. In *Proceedings of the Twelfth International Conference on the Principles of Knowledge Representation and Reasoning (KR 2010), Toronto, Canada, May 9-13, 2010*, pages 477–485. AAAI Press.

[Basol et al., 2010] Basol, S., Erdem, O., Fink, M., and Ianni, G. (2010). HEX programs with action atoms. In *Technical Communications of the Twenty-Sixth International Conference on Logic Programming, ICLP 2010, July 16-19, 2010, Edinburgh, Scotland, UK*, pages 24–33.

[Biere et al., 2009] Biere, A., Heule, M. J. H., van Maaren, H., and Walsh, T., editors (2009). *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press.

[Bikakis and Antoniou, 2008] Bikakis, A. and Antoniou, G. (2008). Alternative strategies for contextual reasoning with conflicts in ambient computing. In Calvanese, D. and Lausen, G., editors, *Web Reasoning and Rule Systems*, volume 5341 of *Lecture Notes in Computer Science*, pages 234–235. Springer.

[Bikakis and Antoniou, 2010] Bikakis, A. and Antoniou, G. (2010). Defeasible contextual reasoning with arguments in ambient intelligence. *IEEE Transactions on Knowledge and Data Engineering*, 22(11):1492–1506.

[Bögl et al., 2010] Bögl, M., Eiter, T., Fink, M., and Schüller, P. (2010). The MCS-IE system for explaining inconsistency in multi-context systems. In *In Proceedings of the Twelfth European Conference on Logics in Artificial Intelligence (JELIA 2010)*, pages 356–359.

[Bonatti, 2001] Bonatti, P. A. (2001). Reasoning with infinite stable models. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI 2001)*, pages 603–610. Morgan Kaufmann.

[Bonatti, 2002] Bonatti, P. A. (2002). Reasoning with infinite stable models II: Disjunctive programs. In *Proceedings of the Eighteenth International Conference on Logic Programming (ICLP 2002)*, volume 2401 of *LNCS*, pages 333–346. Springer.

[Brewka and Eiter, 2007] Brewka, G. and Eiter, T. (2007). Equilibria in heterogeneous nonmonotonic multi-context systems. In *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence (AAAI 2007), July 22–26, 2007, Vancouver, British Columbia, Canada*, pages 385–390. AAAI Press.

[Brewka et al., 2011] Brewka, G., Eiter, T., and Truszczyński, M. (2011). Answer set programming at a glance. *Communications of the ACM*, 54(12):92–103. doi>10.1145/2043174.2043195.

[Brown, 2003] Brown, F. (2003). *Boolean Reasoning: The Logic of Boolean Equations*. Dover Books on Mathematics. Dover Publications.

[Buccafurri et al., 1997] Buccafurri, F., Leone, N., and Rullo, P. (1997). Strong and weak constraints in disjunctive datalog. In *Proceedings of the Fourth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 1997)*, pages 2–17, London, UK. Springer.

[Cabalar et al., 2009] Cabalar, P., Pearce, D., and Valverde, A. (2009). A revised concept of safety for general answer set programs. In *Proceedings of the Tenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2009)*, volume 5753 of *LNCS*, pages 58–70. Springer.

[Calimeri et al., 2007] Calimeri, F., Cozza, S., and Ianni, G. (2007). External sources of knowledge and value invention in logic programming. *Annals of Mathematics and Artificial Intelligence*, 50(3–4):333–361.

[Calimeri et al., 2008a] Calimeri, F., Cozza, S., Ianni, G., and Leone, N. (2008a). Computable functions in ASP: theory and implementation. In *Proceedings of the Twenty-Fourth International Conference on Logic Programming (ICLP 2008)*, volume 5366 of *LNCS*, pages 407–424. Springer.

[Calimeri et al., 2013a] Calimeri, F., Fink, M., Germano, S., Ianni, G., Redl, C., and Wimmer, A. (2013a). AngryHEX: an angry birds-playing agent based on HEX-programs. Angry-Birds Competition 2013, August 6-9, 2013, Beijing, China.

[Calimeri et al., 2013b] Calimeri, F., Fink, M., Germano, S., Ianni, G., Redl, C., and Wimmer, A. (2013b). AngryHEX: an artificial player for angry birds based on declarative knowledge bases. In Baldoni, M., Chesani, F., Mello, P., and Montali, M., editors, *National Workshop and Prize on Popularize Artificial Intelligence, Turin, Italy, December 5, 2013*, pages 29–35.

[Calimeri et al., 2008b] Calimeri, F., Perri, S., and Ricca, F. (2008b). Experimenting with parallelism for the instantiation of ASP programs. *Journal of Algorithms*, 63(1-3):34–54.

[Clark, 1977] Clark, K. L. (1977). Negation as failure. In *Logic and Data Bases*, pages 293–322.

[CLASP Website, 2014] CLASP Website (2014). http://www.cs.uni-potsdam.de/clasp.

[Dao-Tran et al., 2009a] Dao-Tran, M., Eiter, T., Fink, M., and Krennwallner, T. (2009a). Modular nonmonotonic logic programming revisited. In Hill, P. M. and Warren, D. S., editors, *Proceedings of the Twenty-Fifth International Conference on Logic Programming (ICLP 2009), Pasadena, California, USA, July 14–17, 2009*, volume 5649 of *LNCS*, pages 145–159. Springer.

[Dao-Tran et al., 2009b] Dao-Tran, M., Eiter, T., and Krennwallner, T. (2009b). Realizing default logic over description logic knowledge bases. In Sossai, C. and Chemello, G., editors, *Symbolic and Quantitative Approaches to Reasoning with Uncertainty*, volume 5590 of *Lecture Notes in Computer Science*, pages 602–613. Springer Berlin / Heidelberg.

[Davis et al., 1962] Davis, M., Logemann, G., and Loveland, D. (1962). A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397.

[Delgrande et al., 2004] Delgrande, J. P., Schaub, T., Tompits, H., and Wang, K. (2004). A classification and survey of preference handling approaches in nonmonotonic reasoning. *Computational Intelligence*, 20(2):308–334.

[DLV Website, 2014] DLV Website (2014). http://www.dlvsystem.com.

[Drescher et al., 2008] Drescher, C., Gebser, M., Grote, T., Kaufmann, B., König, A., Ostrowski, M., and Schaub, T. (2008). Conflict-driven disjunctive answer set solving. In *Proceedings of the Eleventh International Conference on Principles of Knowledge Representation and Reasoning (KR 2008)*, pages 422–432. AAAI Press.

[Dung et al., 2007] Dung, P., Mancarella, P., and Toni, F. (2007). Computing ideal sceptical argumentation. *Artificial Intelligence*, 171:642–674.

[Dung, 1995] Dung, P. M. (1995). On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games. *Artificial Intelligence*, 77(2):321–357.

[Dunne, 2009] Dunne, P. E. (2009). The computational complexity of ideal semantics. *Artificial Intelligence*, 173(18):1559–1591.

[Egly et al., 2010] Egly, U., Gaggl, S. A., and Woltran, S. (2010). Answer-set programming encodings for argumentation frameworks. *Argument and Computation*, 1(2):147–177.

[Eiter et al., 2011a] Eiter, T., Fink, M., Ianni, G., Krennwallner, T., and Schüller, P. (2011a). Pushing efficient evaluation of HEX programs by modular decomposition. In Delgrande, J. and Faber, W., editors, *Proceedings of the Eleventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2011), Vancouver, BC, Canada, May 16-19, 2011*, volume 6645 of *LNCS*, pages 93–106. Springer.

[Eiter et al., 2012a] Eiter, T., Fink, M., Krennwallner, T., and Redl, C. (2012a). Conflict-driven ASP solving with external sources. *Theory and Practice of Logic Programming: Special Issue Twenty-Eighth International Conference on Logic Programming (ICLP 2012)*, 12(4–5):659–679. Published online: September 5, 2012.

[Eiter et al., 2013a] Eiter, T., Fink, M., Krennwallner, T., and Redl, C. (2013a). Grounding HEX-Programs with Expanding Domains. In Pearce, D., Tasharrofi, S., Ternovska, E., and Vidal, C., editors, *Second Workshop on Grounding and Transformations for Theories with Variables (GTTV 2013), Corunna, Spain, September 15, 2013*, pages 3–15.

[Eiter et al., 2013b] Eiter, T., Fink, M., Krennwallner, T., and Redl, C. (2013b). HEX-Programs with Existential Quantification. In Rocha, R., editor, *Proceedings of the Twentieth International Conference on Applications of Declarative Programming and Knowledge Management (INAP 2013), Kiel, Germany, September 11-13, 2013*.

[Eiter et al., 2013c] Eiter, T., Fink, M., Krennwallner, T., and Redl, C. (2013c). Liberal safety for answer set programs with external sources. In desJardins, M. and Littman, M., editors, *Proceedings of the Twenty-Seventh AAAI Conference (AAAI 2013), July 14–18, 2013, Bellevue, Washington, USA*. AAAI Press.

[Eiter et al., 2014a] Eiter, T., Fink, M., Krennwallner, T., and Redl, C. (2014a). HEX-programs with existential quantification. In Rocha, R., editor, *Proceedings of the Twentieth International Conference on Applications of Declarative Programming and Knowledge Manage-*

ment (INAP 2013), Kiel, Germany, September 11-13, 2013. Post proceedings. Accepted for publication.

[Eiter et al., 2012b] Eiter, T., Fink, M., Krennwallner, T., Redl, C., and Schüller, P. (2012b). Eliminating Unfounded Set Checking for HEX-Programs. In Fink, M. and Lierler, Y., editors, *Fifth Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP 2012), September 4, 2012, Budapest, Hungary*, pages 83–97.

[Eiter et al., 2012c] Eiter, T., Fink, M., Krennwallner, T., Redl, C., and Schüller, P. (2012c). Exploiting Unfounded Sets for HEX-Program Evaluation. In del Cerro, L. F., Herzig, A., and Mengin, J., editors, *Proceedings of the Thirteenth European Conference on Logics in Artificial Intelligence (JELIA 2012), September 26-28, 2012, Toulouse, France*, volume 7519 of *LNCS*, pages 160–175. Springer.

[Eiter et al., 2013d] Eiter, T., Fink, M., Krennwallner, T., Redl, C., and Schüller, P. (2013d). Improving HEX-program evaluation based on unfounded sets. Technical Report INFSYS RR-1843-12-08, Institut für Informationssysteme, Technische Universität Wien, A-1040 Vienna, Austria.

[Eiter et al., 2014b] Eiter, T., Fink, M., Krennwallner, T., Redl, C., and Schüller, P. (2014b). Efficient HEX-program evaluation based on unfounded sets. *Journal of Artificial Intelligence Research*, 49:269–321.

[Eiter et al., 2010] Eiter, T., Fink, M., Schüller, P., and Weinzierl, A. (2010). Finding explanations of inconsistency in multi-context systems. In *Proceedings of the Twelfth International Conference on the Principles of Knowledge Representation and Reasoning (KR 2010), Toronto, Canada, May 9-13, 2010*, pages 329–339. AAAI Press.

[Eiter et al., 2012d] Eiter, T., Fink, M., Schüller, P., and Weinzierl, A. (2012d). Finding explanations of inconsistency in nonmonotonic multi-context systems. Technical Report INFSYS RR-1843-12-09, INFSYS RR-1843-03-08, Institut für Informationssysteme, TU Wien.

[Eiter et al., 2013e] Eiter, T., Fink, M., and Stepanova, D. (2013e). Data repair of inconsistent dl-programs. In *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence*, IJCAI 2013, pages 869–876. AAAI Press.

[Eiter et al., 1997] Eiter, T., Gottlob, G., and Veith, H. (1997). Modular logic programming and generalized quantifiers. In *Proceedings of the Fourth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 1997)*, volume 1265 of *LNCS*, pages 290–309. Springer.

[Eiter et al., 2009] Eiter, T., Ianni, G., and Krennwallner, T. (2009). Answer set programming: A primer. In Tessaris, S., Franconi, E., Eiter, T., Gutierrez, C., Handschuh, S., Rousset, M.-C., and Schmidt, R. A., editors, *Fifth International Reasoning Web Summer School (RW 2009), Brixen/Bressanone, Italy, August 30–September 4, 2009*, volume 5689 of *LNCS*, pages 40–110. Springer.

[Eiter et al., 2008] Eiter, T., Ianni, G., Krennwallner, T., and Schindlauer, R. (2008). Exploiting conjunctive queries in description logic programs. *Annals of Mathematics and Artificial Intelligence: Logic in AI: A Special Issue Dedicated to Victor W. Marek on the Occasion of His 65th birthday*, 53(1–4):115–152. Published online: January 27, 2009.

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

[Eiter et al., 2006a] Eiter, T., Ianni, G., Schindlauer, R., and Tompits, H. (2006a). Effective integration of declarative rules with external evaluations for semantic-web reasoning. In *Proceedings of the Third European Conference on Semantic Web (ESWC 2006)*, volume 4011 of *LNCS*, pages 273–287. Springer.

[Eiter et al., 2006b] Eiter, T., Ianni, G., Schindlauer, R., and Tompits, H. (2006b). Effective integration of declarative rules with external evaluations for semantic-web reasoning. In *Proceedings of the Third European Conference on Semantic Web (ESWC 2006)*, pages 273–287.

[Eiter et al., 2006c] Eiter, T., Ianni, G., Schindlauer, R., Tompits, H., and Wang, K. (2006c). Forgetting in managing rules and ontologies. In *IEEE/WIC/ACM International Conference on Web Intelligence (WI 2006), Hongkong, December 2006*, pages 411–419. IEEE Computer Society. Preliminary version at ALPSWS 2006.

[Eiter et al., 2011b] Eiter, T., Krennwallner, T., and Redl, C. (2011b). Nested HEX-programs. *CoRR*, abs/1108.5626.

[Eiter et al., 2013f] Eiter, T., Krennwallner, T., and Redl, C. (2013f). HEX-Programs with Nested Program Calls. In Tompits, H., editor, *Nineteenth International Conference on Applications of Declarative Programming and Knowledge Management (INAP 2011)*, volume 7773 of *LNAI*, pages 1–10. Springer.

[Eiter and Simkus, 2010] Eiter, T. and Simkus, M. (2010). FDNC: decidable nonmonotonic disjunctive logic programs with function symbols. *ACM Transactions on Computational Logic*, 11(2):14:1–14:50.

[Erdogan et al., 2010] Erdogan, H., Oztok, U., Erdem, Y., and Erdem, E. (2010). Querying biomedical ontologies in natural language using answer set programming. In Paschke, A., Burger, A., Splendiani, A., Marshall, M. S., and Romano, P., editors, *3rd International Workshop on Semantic Web Applications and Tools for the Life Sciences (SWAT4LS 2010), Berlin, Germany, December 8-10, 2010*, volume abs/1012.1899 of *CoRR*, page 4.

[Faber, 2005] Faber, W. (2005). Unfounded sets for disjunctive logic programs with arbitrary aggregates. In *Proceedings of the Eighth International Conference on Logic Programming*

and Nonmonotonic Reasoning (LPNMR 2005), Diamante, Italy, September 5-8, 2005, volume 3662, pages 40–52. Springer.

[Faber et al., 1999] Faber, W., Leone, N., Mateis, C., and Pfeifer, G. (1999). Using database optimization techniques for nonmonotonic reasoning. In *Seventh International Workshop on Deductive Databases and Logic Programming (DDLP 1999)*, pages 135–139. Prolog Association of Japan.

[Faber et al., 2011] Faber, W., Leone, N., and Pfeifer, G. (2011). Semantics and complexity of recursive aggregates in answer set programming. *Artificial Intelligence*, 175(1):278–298.

[Faber and Woltran, 2011] Faber, W. and Woltran, S. (2011). Manifold answer-set programs and their applications. In Balduccini, M. and Son, T. C., editors, *Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning*, volume 6565 of *Lecture Notes in Computer Science*, pages 44–63. Springer.

[Fages, 1994] Fages, F. (1994). Consistency of Clark's completion and existence of stable models. *Journal of Methods of Logic in Computer Science*, 1:51–60.

[Fagin et al., 2005] Fagin, R., Kolaitis, P., Miller, R., and Popa, L. (2005). Data exchange: Semantics and query answering. *Theoretical Computer Science*, 336(1):89–124.

[Ferraris, 2005] Ferraris, P. (2005). Answer sets for propositional theories. In Baral, C., Greco, G., Leone, N., and Terracina, G., editors, *Proceedings of the Eighth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2005), Diamante, Italy, September 5-8, 2005*, volume 3662 of *Lecture Notes in Computer Science*, pages 119–131. Springer.

[Ferraris, 2011] Ferraris, P. (2011). Logic programs with propositional connectives and aggregates. *ACM Transactions on Computational Logic (TOCL)*, 12(4):44.

[Fink et al., 2013] Fink, M., Germano, S., Ianni, G., Redl, C., and Schüller, P. (2013). ActHEX: implementing HEX programs with action atoms. In Cabalar, P. and Son, T., editors, *Proceedings of the Twelfth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2013)*, volume 8148 of *Lecture Notes in Computer Science*, pages 317–322. Springer Berlin Heidelberg.

[Gebser et al., 2011a] Gebser, M., Kaminski, R., König, A., and Schaub, T. (2011a). Advances in gringo series 3. In Delgrande, J. and Faber, W., editors, *Logic Programming and Nonmonotonic Reasoning*, volume 6645 of *Lecture Notes in Computer Science*, pages 345–351. Springer Berlin Heidelberg.

[Gebser et al., 2011b] Gebser, M., Kaufmann, B., Kaminski, R., Ostrowski, M., Schaub, T., and Schneider, M. (2011b). Potassco: The potsdam answer set solving collection. *AI Communications*, 24(2):107–124.

[Gebser et al., 2007a] Gebser, M., Kaufmann, B., Neumann, A., and Schaub, T. (2007a). *c*lasp: a conflict-driven answer set solver. In Baral, C., Brewka, G., and Schlipf, J. S., editors, *Proceedings of the Ninth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2007), Tempe, Arizona, USA, May 15-17, 2007*, volume 4483 of *Lecture Notes in Computer Science*, pages 260–265. Springer.

[Gebser et al., 2012] Gebser, M., Kaufmann, B., and Schaub, T. (2012). Conflict-driven answer set solving: From theory to practice. *Artificial Intelligence*, 187-188(0):52–89.

[Gebser et al., 2013] Gebser, M., Kaufmann, B., and Schaub, T. (2013). Advanced conflict-driven disjunctive answer set solving. In *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence*, IJCAI 2013, pages 912–918. AAAI Press.

[Gebser et al., 2009] Gebser, M., Ostrowski, M., and Schaub, T. (2009). Constraint answer set solving. In Hill, P. and Warren, D., editors, *Proceedings of the Twenty-Fifth International Conference on Logic Programming (ICLP 2009)*, volume 5649 of *Lecture Notes in Computer Science*, pages 235–249. Springer.

[Gebser et al., 2007b] Gebser, M., Schaub, T., and Thiele, S. (2007b). Gringo: A new grounder for answer set programming. In *Nineteenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2007)*, volume 4483, pages 266–271. Springer.

[Gelfond and Lifschitz, 1988] Gelfond, M. and Lifschitz, V. (1988). The stable model semantics for logic programming. In Kowalski, R. and Bowen, K., editors, *Logic Programming: Proceedings of the Fifth International Conference and Symposium*, pages 1070–1080. MIT Press.

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

[Giunchiglia et al., 2004] Giunchiglia, E., Lee, J., Lifschitz, V., McCain, N., Turner, H., and Lifschitz, J. L. V. (2004). Nonmonotonic causal theories. *Artificial Intelligence*, 153:2004.

[Giunchiglia et al., 2008] Giunchiglia, E., Leone, N., and Maratea, M. (2008). On the relation among answer set solvers. *Annals of Mathematics and Artificial Intelligence*, 53:169–204.

[Giunchiglia and Serafini, 1994] Giunchiglia, F. and Serafini, L. (1994). Multilanguage hierarchical logics or: How we can do without modal logics. *Artificial Intelligence*, 65(1):29–70.

[Goldberg and Novikov, 2007] Goldberg, E. and Novikov, Y. (2007). Berkmin: A fast and robust SAT-solver. *Discrete Applied Mathematics*, 155(12):1549–1561.

[Grau et al., 2012] Grau, B. C., Horrocks, I., Krötzsch, M., Kupke, C., Magka, D., Motik, B., and Wang, Z. (2012). Acyclicity conditions and their application to query answering in description logics. In Brewka, G., Eiter, T., and McIlraith, S. A., editors, *Proceedings of the Thirteenth International Conference on Principles of Knowledge Representation and Reasoning*. AAAI Press.

[Greco et al., 2013] Greco, S., Molinaro, C., and Trubitsyna, I. (2013). Bounded programs: A new decidable class of logic programs with function symbols. In *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence (IJCAI 2013)*, IJCAI 2013, pages 926–931. AAAI Press.

[GRINGO Website, 2014] GRINGO Website (2014). http://www.cs.uni-potsdam.de/gringo.

[Halevy et al., 2003] Halevy, A. Y., Ives, Z. G., Suciu, D., and Tatarinov, I. (2003). Schema mediation in peer data management systems. In *Proceedings of the Ninteenth International Conference on Data Engineering (ICDE 2003)*, pages 505–517.

[Heflin and Munoz-Avila, 2002] Heflin, J. and Munoz-Avila, H. (2002). Lcw-based agent planning for the semantic web. In Pease, A., editor, *Ontologies and the Semantic Web*, number WS-02-11 in AAAI Technical Report, pages 63–70, Menlo Park, CA. AAAI Press.

[Heymans et al., 2004] Heymans, S., Nieuwenborgh, D., and Vermeir, D. (2004). Semantic web reasoning with conceptual logic programs. In Antoniou, G. and Boley, H., editors, *Rules and Rule Markup Languages for the Semantic Web*, volume 3323 of *Lecture Notes in Computer Science*, pages 113–127. Springer Berlin Heidelberg.

[Heymans and Toma, 2008] Heymans, S. and Toma, I. (2008). Ranking services using fuzzy HEX-programs. In Calvanese, D. and Lausen, G., editors, *Proceedings of the Second International Conference on Web Reasoning and Rule Systems (RR 2008)*, volume 5341 of *LNCS*, pages 181–196. Springer.

[Hoehndorf et al., 2007] Hoehndorf, R., Loebe, F., Kelso, J., and Herre, H. (2007). Representing default knowledge in biomedical ontologies: Application to the integration of anatomy and phenotype ontologies. *BMC Bioinformatics*, 8(1):377.

[Janhunen et al., 2009] Janhunen, T., Oikarinen, E., Tompits, H., and Woltran, S. (2009). Modularity aspects of disjunctive stable models. *Journal of Artificial Intelligence Research*, 35:813–857.

[Järvisalo et al., 2009] Järvisalo, M., Oikarinen, E., Janhunen, T., and Niemelä, I. (2009). A module-based framework for multi-language constraint modeling. In Erdem, E., Lin, F., and Schaub, T., editors, *Proceedings of the Tenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2009), Potsdam, Germany, 14-18 September, 2009*, pages 155–168. Springer.

[Johnson and Klug, 1984] Johnson, D. and Klug, A. (1984). Testing containment of conjunctive queries under functional and inclusion dependencies. *Journal of Computer and System Sciences*, 28(1):167 – 189.

[Klop, 1992] Klop, J. W. (1992). Handbook of logic in computer science (vol. 2). In Abramsky, S., Gabbay, D. M., and Maibaum, S. E., editors, *Handbook of Logic in Computer Science*, chapter Term Rewriting Systems, pages 1–116. Oxford University Press, Inc., New York, NY, USA.

[Koch et al., 2003] Koch, C., Leone, N., and Pfeifer, G. (2003). Enhancing disjunctive logic programming systems by SAT checkers. *Artificial Intelligence*, 151(1–2):177–212.

[Kowalski and Sadri, 1999] Kowalski, R. and Sadri, F. (1999). From logic programming towards multi-agent systems. *Annals of Mathematics and Artificial Intelligence*, 25(3-4):391–419.

[Krishnamurthy et al., 1996] Krishnamurthy, R., Ramakrishnan, R., and Shmueli, O. (1996). A framework for testing safety and effective computability. *Journal of Computer and System Sciences*, 52(1):100–124.

[Lee et al., 2008] Lee, J., Lifschitz, V., and Palla, R. (2008). Safe formulas in the general theory of stable models (preliminary report). In *Proceedings of the Twenty-Fourth International Conference on Logic Programming (ICLP 2008)*, volume 5366 of *LNCS*, pages 672–676. Springer.

[Lee and Meng, 2009] Lee, J. and Meng, Y. (2009). On reductive semantics of aggregates in answer set programming. In Erdem, E., Lin, F., and Schaub, T., editors, *Proceedings of the Tenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2009), Potsdam, Germany, 14-18 September, 2009*, volume 5753 of *LNCS*, pages 182–195. Springer.

[Lefèvre and Nicolas, 2009] Lefèvre, C. and Nicolas, P. (2009). The first version of a new ASP solver: ASPeRiX. In Erdem, E., Lin, F., and Schaub, T., editors, *Proceedings of the Tenth International Conference on Logic Programming an Nonmonotonic Reasoning (LPNMR 2009), Potsdam, Germany, September 14-18, 2009*, volume 5753 of *LNCS*, pages 522–527. Springer.

[Leone et al., 2012] Leone, N., Manna, M., Terracina, G., and Veltri, P. (2012). Efficiently computable datalog$^{\exists}$ programs. In *Proceedings of the Thirteenth International Conference on Principles of Knowledge Representation and Reasoning (KR 2012)*, pages 13–23.

[Leone et al., 2001] Leone, N., Perri, S., and Scarcello, F. (2001). Improving ASP instantiators by join-ordering methods. In *Proceedings of the Sixth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2001)*, volume 2173 of *LNCS*, pages 280–294. Springer.

[Leone et al., 2006] Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., and Scarcello, F. (2006). The dlv system for knowledge representation and reasoning. *ACM Transactions on Computational Logic (TOCL)*, 7(3):499–562.

[Lierler, 2011] Lierler, Y. (2011). Abstract answer set solvers with backjumping and learning. *Theory and Practice of Logic Programming (TPLP)*, 11(2–3):135–169.

[Lierler and Lifschitz, 2009] Lierler, Y. and Lifschitz, V. (2009). One more decidable class of finitely ground programs. In *Proceedings of the Twenty-Fifth International Conference on Logic Programming (ICLP 2009)*, volume 5649 of *LNCS*, pages 489–493. Springer.

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

[Lifschitz et al., 1999] Lifschitz, V., Tang, L., and Turner, H. (1999). Nested expressions in logic programs. *Annals of Mathematics and Artificial Intelligence*, 25(3-4):369–389.

[Lifschitz and Turner, 1994] Lifschitz, V. and Turner, H. (1994). Splitting a logic program. In *Proceedings ICLP-94*, pages 23–38, Santa Margherita Ligure, Italy. MIT-Press.

[Lin and Zhao, 2004] Lin, F. and Zhao, Y. (2004). ASSAT: computing answer sets of a logic program by SAT solvers. *Artificial Intelligence*, 157(1–2):115–137.

[Lua Website, 2014] Lua Website (2014). http://www.lua.org.

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

[Marques-Silva and Sakallah, 1999] Marques-Silva, J. P. and Sakallah, K. A. (1999). GRASP: a search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521.

[Mitchell, 2005] Mitchell, D. G. (2005). A SAT solver primer. *EATCS Bulletin (The Logic in Computer Science Column)*, 85:112–133.

[Mosca and Bernini, 2008] Mosca, A. and Bernini, D. (2008). Ontology-driven geographic information system and dlvhex reasoning for material culture analysis. In *Italian Workshop RiCeRcA 2008*.

[Motik and Sattler, 2006] Motik, B. and Sattler, U. (2006). A comparison of reasoning techniques for querying large description logic aboxes. In *Proceedings of the Thirteenth international conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2006)*, pages 227–241. Springer.

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

[Nieuwenhuis and Oliveras, 2005] Nieuwenhuis, R. and Oliveras, A. (2005). DPLL(T) with exhaustive theory propagation and its application to difference logic. In *Seventeenth International Conference on Computer Aided Verification (CAV 2005)*, pages 321–334. Springer.

[Ostrowski and Schaub, 2012] Ostrowski, M. and Schaub, T. (2012). ASP modulo CSP: the clingcon system. *Theory and Practice of Logic Programming (TPLP)*, 12(4-5):485–503.

[Palù et al., 2009] Palù, A., Dovier, A., Pontelli, E., and Rossi, G. (2009). Answer set programming with constraints using lazy grounding. In Hill, P. and Warren, D. S., editors, *Proceedings of the Twenty-Fifth International Conference on Logic Programming (ICLP 2009)*, volume 5649 of *LNCS*, pages 115–129. Springer.

[Palù et al., 2009] Palù, A. D., Dovier, A., Pontelli, E., and Rossi, G. (2009). GASP: answer set programming with lazy grounding. *Fundamenta Informaticae*, 96(3):297–322.

[Papadimitriou, 1994] Papadimitriou, C. M. (1994). *Computational complexity*. Addison-Wesley, Reading, Massachusetts.

[Pelov et al., 2007] Pelov, N., Denecker, M., and Bruynooghe, M. (2007). Well-founded and stable semantics of logic programs with aggregates. *Theory and Practice of Logic Programming (TPLP)*, 7(3):301–353.

[Polleres, 2007] Polleres, A. (2007). From SPARQL to rules (and back). In *Proceedings of the Sixteenth World Wide Web Conference (WWW 2007)*, pages 787–796.

[Prud'hommeaux and Seaborne, 2007] Prud'hommeaux, E. and Seaborne, A. (2007). SPARQL query language for RDF. W3C Proposed Recommendation, World Wide Web Consortium.

[Ramakrishnan et al., 1987] Ramakrishnan, R., Bancilhon, F., and Silberschatz, A. (1987). Safety of recursive horn clauses with infinite relations. In *Sixth Symposium on Principles of Database Systems (PODS 1987)*, pages 328–339. ACM.

[Redl, 2010] Redl, C. (2010). Development of a belief merging framewerk for dlvhex. Master's thesis, Vienna University of Technology, Knowledge-based Systems Group, A-1040 Vienna, Karlsplatz 13.

[Redl et al., 2011] Redl, C., Eiter, T., and Krennwallner, T. (2011). Declarative belief set merging using merging plans. In Rocha, R. and Launchbury, J., editors, *Thirteenth International Symposium on Practical Aspects of Declarative Languages (PADL 2011), Austin, Texas, USA, January 24-25, 2011*, volume 6539 of *LNCS*, pages 99–114. Springer.

[Reiter, 1980] Reiter, R. (1980). Readings in nonmonotonic reasoning. In Ginsberg, M. L., editor, *Readings in Nonmonotonic Reasoning*, chapter A logic for default reasoning, pages 68–93. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[Sagiv and Vardi, 1989] Sagiv, Y. and Vardi, M. Y. (1989). Safety of datalog queries over infinite databases. In *Eighth Symposium on Principles of Database Systems (PODS 1989)*, pages 160–171. ACM.

[Schindlauer, 2006] Schindlauer, R. (2006). *Answer Set Programming for the Semantic Web*. PhD thesis, Vienna University of Technology, Vienna, Austria.

[Schüller, 2012] Schüller, P. (2012). *Inconsistency in Multi-Context Systems: Analysis and Efficient Evaluation*. PhD thesis, Vienna University of Technology, Vienna, Austria.

[Shen, 2011] Shen, Y.-D. (2011). Well-supported semantics for description logic programs. In *Twenty-Second International Joint Conference on Artificial Intelligence (IJCAI 2011)*, pages 1081–1086.

[Shen and Wang, 2011] Shen, Y.-D. and Wang, K. (2011). Extending logic programs with description logic expressions for the semantic web. In *International Semantic Web Conference (ISWC 2011)*, pages 633–648.

[Shen et al., 2014] Shen, Y.-D., Wang, K., Deng, J., Redl, C., Krennwallner, T., Eiter, T., and Fink, M. (2014). FLP answer set semantics without circular justifications for general logic programs. *Artificial Intelligence*. Accepted for publication.

[Simons et al., 2002] Simons, P., Niemelä, I., and Soininen, T. (2002). Extending and implementing the stable model semantics. *Artificial Intelligence*, 138:181–234.

[Sipser, 2012] Sipser, M. (2012). *Introduction to the Theory of Computation*. Course Technology, 3rd edition.

[SMODELS Website, 2014] SMODELS Website (2014). http://www.tcs.hut.fi/Software/smodels.

[Stashuk, 2013] Stashuk, O. (2013). Integrating constraint programming into answer set programming. Master's thesis, Vienna University of Technology, Knowledge-based Systems Group, A-1040 Vienna, Karlsplatz 13.

[Swift and Warren, 2012] Swift, T. and Warren, D. S. (2012). Xsb: Extending prolog with tabled logic programming. *Theory and Practice of Logic Programming (TPLP)*, 12(1–2):157–187.

[Syrjänen, 2001] Syrjänen, T. (2001). Omega-restricted logic programs. In *Sixth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2001)*, pages 267–279. Springer.

[Syrjänen, 2009] Syrjänen, T. (2009). *Logic Programs and Cardinality Constraints: Theory and Practice*. PhD thesis, Helsinki University of Technology, Espoo, Finland.

[Terracina et al., 2008] Terracina, G., Leone, N., Lio, V., and Panetta, C. (2008). Experimenting with recursive queries in database and logic programming systems. *Theory and Practice of Logic Programming (TPLP)*, 8(2):129–165.

[Van Nieuwenborgh et al., 2007] Van Nieuwenborgh, D., Eiter, T., and Vermeir, D. (2007). Conditional planning with external functions. In *Proceedings of the Ninth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2007)*, volume 4483 of *LNAI*, pages 214–227. Springer.

[Zantema, 1994] Zantema, H. (1994). Termination of term rewriting: Interpretation and type elimination. *Journal of Symbolic Computation*, 17(1):23–50.

[Zantema, 2001] Zantema, H. (2001). The termination hierarchy for term rewriting. *Applicable Algebra in Engineering, Communication and Computing*, 12(1-2):3–19.

[Zirtiloğlu and Yolum, 2008] Zirtiloğlu, H. and Yolum, P. (2008). Ranking semantic information for e-government: complaints management. In *Proceedings of the First International Workshop on Ontology-supported Business Intelligence (OBI 2008)*, number 5 in OBI 2008, page 7. ACM.

[Zuker and Sankoff, 1984] Zuker, M. and Sankoff, D. (1984). RNA secondary structures and their prediction. *Bulletin of Mathematical Biology*, 46(4):591–621.

Appendix A

Benchmark Encodings

In this appendix, we give details to some of those benchmark encodings which are not described in the existing literature or directly in Chapter 5. The encodings are very natural representations of the underlying problems in HEX and have not been optimized for our algorithms, thus they have been developed under realistic conditions; some of them have even existed before the techniques in this thesis were developed.

The encodings have been included to make the thesis self-contained. However, it is noted that some of them were not developed by the author of this thesis; in such cases the name of the actual author, both of the HEX-rules and of the informal explanation, is given in the respective section. Some of the encodings have been published by Eiter et al. (2014b).

A.1 Abstract Argumentation

The *Abstract Argumentation* benchmark results in Section 5.2.1 were obtained using the following encoding, which has been developed by Peter Schüller [Eiter et al., 2014b]. It is derived from encodings for admissible and preferred set extensions of an argumentation framework (A, att), given by Egly et al. (2010).

Input instances of this benchmark are defined over a set A of arguments encoded as facts $arg(a)$ for each $a \in A$ and a set att of attacks between arguments, encoded as facts $att(a, b)$ for some $(a, b) \in A \times A$. The encoding consists of the following rules where $x, y, z \in A$; very similar encodings are explained in detail by Egly et al. (2010) (but without the use of external atoms).

First, we define defeat from attacks.

$$defeat(x, y) \leftarrow att(x, y)$$

We guess a set $S \subseteq A$ using predicates in_S and out_S.

$$in_S(x) \leftarrow \text{not } out_S(x), arg(x); \qquad out_S(x) \leftarrow \text{not } in_S(x), arg(x)$$

Next, we require that all arguments in S are conflict-free and defended from S.

$$\begin{aligned} &\leftarrow in_S(x), in_S(y), defeat(x,y) \\ defeated(x) &\leftarrow in_S(y), defeat(y,x) \\ notDefended(x) &\leftarrow defeat(y,x), \text{not } defeated(y) \\ &\leftarrow in_S(x), notDefended(x) \end{aligned}$$

For saturation we define a linear order on arguments, including infimum and supremum.

$$\begin{aligned} lt(x,y) &\leftarrow arg(x), arg(y) && (x < y) \\ nsucc(x,z) &\leftarrow lt(x,y), lt(y,z) \\ succ(x,y) &\leftarrow lt(x,y), \text{not } nsucc(x,y) \\ ninf(x) &\leftarrow lt(y,x); & nsup(x) &\leftarrow lt(x,y) \\ inf(x) &\leftarrow \text{not } ninf(x), arg(x); & sup(x) &\leftarrow \text{not } nsup(x), arg(x) \end{aligned}$$

We perform a guess over a set $T \subseteq A$ using a disjunction.

$$in_T(x) \vee out_T(x) \leftarrow arg(x)$$

We check each argument of T whether it is in S and spoil the answer set if $S \subseteq T$.

$$\begin{aligned} sInT_{upto}(y) &\leftarrow inf(y), in_S(y), in_T(y) \\ sInT_{upto}(y) &\leftarrow inf(y), out_S(y) \\ sInT_{upto}(y) &\leftarrow succ(z,y), in_S(y), in_T(y), sInT_{upto}(z) \\ sInT_{upto}(y) &\leftarrow succ(z,y), out_S(y), sInT_{upto}(z) \\ sInT &\leftarrow sup(y), sInT_{upto}(y) \\ spoil &\leftarrow sInT \end{aligned}$$

We also spoil the answer set if T is not a preferred extension, determined by an external atom with semantics $f_{\&argSemExt}$ s.t. $f_{\&argSemExt}(\mathbf{A}, pref, arg, att, in_T, unused, spoil) = 1$ iff$_{\text{def}}$ $\mathbf{F}spoil \in \mathbf{A}$ or the extension of predicate in_T is a preferred set extension of the argumentation framework specified by the extension of predicates arg and att. Internally, the external atom uses another ASP program to compute the semantics. This check is performed using an ASP encoding for preferred extensions proposed by Egly et al. (2010).

$$\begin{aligned} tIsNotPref &\leftarrow \&argSemExt[pref, arg, att, in_T, unused, spoil]() \\ spoil &\leftarrow tIsNotPref \end{aligned}$$

Note that the parameters $pref$ and $unused$ support more general functionalities of $f_{\&argSemExt}$ which are not relevant for this benchmark. We create a unique answer set whenever $spoil$ is true and require that only spoiled answer sets are returned.

$$\begin{aligned} in_T(x) &\leftarrow spoil, arg(x); & out_T(x) &\leftarrow spoil, arg(x) \\ sInT &\leftarrow spoil; & tIsNotPref &\leftarrow spoil; & &\leftarrow \text{not } spoil \end{aligned}$$

Given an instance encoded as above, the above program computes all ideal sets of the argumentation framework, which correspond 1-1 to its answer sets.

A.2 Conformant Planning

The *Conformant Planning* benchmark results in Section 5.2.1 were obtained using the following encoding, which has also been developed by Peter Schüller [Eiter et al., 2014b].

Input instances of this benchmark are defined over a set R of robots, sets X and Y of valid x and y coordinates of the environment, and a maximum plan length l; an instance contains for each robot $r \in R$ the initial position (x, y) as facts $robo_X(r, x, 0)$ and $robo_Y(r, y, 0)$. The encoding consists of the following rules where, unless stated otherwise, $0 \leq t < l$, $r \in R$, $x \in X$, $y \in Y$.

For each robot we generate four possible moves in the environment.

$$move(r, x, y+1, t) \vee \overline{move}(r, x, y+1, t) \leftarrow robo_X(r, x, t), robo_Y(r, y, t) \quad (y+1 \in Y)$$
$$move(r, x, y-1, t) \vee \overline{move}(r, x, y-1, t) \leftarrow robo_X(r, x, t), robo_Y(r, y, t) \quad (y-1 \in Y)$$
$$move(r, x+1, y, t) \vee \overline{move}(r, x+1, y, t) \leftarrow robo_X(r, x, t), robo_Y(r, y, t) \quad (x+1 \in X)$$
$$move(r, x-1, y, t) \vee \overline{move}(r, x-1, y, t) \leftarrow robo_X(r, x, t), robo_Y(r, y, t) \quad (x-1 \in X)$$

We disallow moving to multiple locations and standing still (the latter is not strictly necessary but we obtained experiments results that way).

$$\leftarrow move(r, x_1, y_1, t), move(r, x_1, y_2, t) \quad (x_1, x_2 \in X,\ x_1 < x_2,\ y_1, y_2 \in Y)$$
$$\leftarrow move(r, x, y_1, t), move(r, x, y_2, t) \quad (y_1, y_2 \in Y,\ y_1 < y_2)$$
$$move_\exists(r, t) \leftarrow move(r, x, y, t)$$
$$\leftarrow \text{not } move_\exists(r, t)$$

The effect of moving is a deterministic change of location.

$$robo_X(r, x, t+1) \leftarrow move(r, x, y, t); \qquad robo_Y(r, y, t+1) \leftarrow move(r, x, y, t)$$

For saturation we guess the position of the object.

$$obj_X(x) \vee \overline{obj}_X(x) \leftarrow; \qquad obj_X(y) \vee \overline{obj}_Y(y) \leftarrow$$

We spoil the answer set if the object is at multiple locations.

$$spoil \leftarrow obj_X(x_1), obj_X(x_2) \quad (x_1, x_2 \in X,\ x_1 < x_2)$$
$$spoil \leftarrow obj_Y(y_1), obj_Y(y_2) \quad (y_1, y_2 \in Y,\ y_1 < y_2)$$

We spoil the answer set if the object is at no location.

$$objectHasNoXUpTo(1) \leftarrow \overline{obj}_X(1)$$
$$objectHasNoXUpTo(x) \leftarrow objectHasNoXUpTo(x-1), \overline{obj}_X(x) \quad (x-1 \in X)$$
$$spoil \leftarrow objectHasNoXUpTo(x_{max}) \quad (x_{max} = max(X))$$
$$objectHasNoYUpTo(1) \leftarrow \overline{obj}_Y(1)$$
$$objectHasNoYUpTo(y) \leftarrow objectHasNoYUpTo(y-1), \overline{obj}_Y(y) \quad (y-1 \in Y)$$
$$spoil \leftarrow objectHasNoYUpTo(y_{max}) \quad (y_{max} = max(Y))$$

We spoil the answer set if the object is sensed, which is determined by an external atom with the semantic function $f_{\&sense}$ such that $f_{\&sense}(\mathbf{A}, robo_X, robo_Y, obj_X, obj_Y, range, spoil) = 1$ iff$_{\text{def}}$ $\mathbf{T}spoil \in \mathbf{A}$ or the predicates $robo_X$, $robo_Y$, obj_X, obj_Y represent in $\mathbf{A}$ a state where the robot has a distance less than $range$ to the object, i.e., the robot can detect the object. The implementation of this external atom was realized in C++ and consists of verifying $range \leq \sqrt{\Delta_x^2 + \Delta_y^2}$ and bookkeeping code to extract Δ_x and Δ_y from $\mathbf{A}$.

$$spoil \leftarrow \&sense[robo_X, robo_Y, obj_X, obj_Y, range, spoil]()$$

We create a unique answer set whenever $spoil$ is true and require that only spoiled answer sets are returned.

$$\begin{aligned} obj_X(x) &\leftarrow spoil; & \overline{obj}_X(x) &\leftarrow spoil \\ obj_Y(x) &\leftarrow spoil; & \overline{obj}_Y(x) &\leftarrow spoil \\ objectHasNoXUpTo(x) &\leftarrow spoil; & objectHasNoYUpTo(y) &\leftarrow spoil \\ &\leftarrow \text{not } spoil \end{aligned}$$

Given an instance encoded as above, an answer set to the above program exists iff there exists a sequence of movements that ensures to detect the object no matter where it is located. Furthermore the movements required to detect the object, i.e., the conformant plan, is encoded in the answer set in the extension of the $move$ predicate.

A.3 Reachability

We assume that the instance consists of two facts $start(s)$ and $end(e)$. The goal is to check if node e is reachable from node s and to compute a witness in this case (i.e., a path from s to e).

The problem encoding uses the following set of rules to import the relevant part of the graph, consisting of all nodes and edges reachable from s. We use an external atom $\&out[X](Y)$ to get all nodes Y which are directly reachable from X.

$$\begin{aligned} edge(X, Y) &\leftarrow node(X), \&out[X](Y) \\ node(X) &\leftarrow start(X) \\ node(X) &\leftarrow end(X) \\ node(X) &\leftarrow edge(X, Y) \end{aligned}$$

We then guess the path as a set of pairs (n, i), where n is the node visited in step i. Constraints are used to ensure that the path is consecutive and reaches e in the last step.

$$
\begin{aligned}
path(X,0) &\leftarrow start(X)\\
&\leftarrow path(X,N), path(Y,N), X \neq Y\\
path(Y,N) \vee \overline{path}(Y,N) &\leftarrow node(Y), \#int(N)\\
&\leftarrow path(X,N), path(Y,N2), N2 = N+1, \text{not } edge(X,Y)\\
existspath(N) &\leftarrow path(X,N)\\
&\leftarrow \#int(N), \text{not } existspath(N), existspath(N2), N2 = N+1\\
lastnode(X) &\leftarrow path(X,N), N2 = N+1, \text{not } existspath(N2)\\
&\leftarrow end(Z), lastnode(Y), Z \neq Y
\end{aligned}
$$

The strongly safe version of the program needs to import the whole graph a priory because $\&reachable[X](Y)$ appears in a cycle and thus needs a domain predicate which bounds Y.

A.4 Mergesort

In this benchmark encoding, lists are encoded as string constants. A dedicated separator character allows for decoding the list representations in the external atoms.

We first use a set of rules to split the input list inp, given by a fact $list(inp)$, recursively into half. For each list l, its halves l_1 and l_2 are generated by the use of an external atom $\&splitHalf[l](l_1, l_2)$. We then derive an atom $sublist(l, l_1, l_2)$ that allows us later to reassemble the list from its (sorted) sublists. Let ϵ denote the empty list.

$$
\begin{aligned}
sublist(L, H1, H2) &\leftarrow list(L), \&splitHalf[L](H1, H2)\\
sublist(L, H1, H2) &\leftarrow sublist(\_, L, \_), \&splitHalf[L](H1, H2), H1 \neq \epsilon, H2 \neq \epsilon\\
sublist(L, H1, H2) &\leftarrow sublist(\_, \_, L), \&splitHalf[L](H1, H2), H1 \neq \epsilon, H2 \neq \epsilon
\end{aligned}
$$

Next, we declare each sublist of length 1 immediately as sorted, where we use an external atom $\&getLength[l](L)$ to get the length L of list l.

$$
\begin{aligned}
sorted(L, L) &\leftarrow sublist(L, \_, \_), \&getLength[L](1)\\
sorted(L, L) &\leftarrow sublist(\_, L, \_), \&getLength[L](1)\\
sorted(L, L) &\leftarrow sublist(\_, \_, L), \&getLength[L](1)
\end{aligned}
$$

Finally, we realize the merge step as follows. For each list l and its sublists l_1 and l_2, we determine the sorted versions s_1 and s_2 of l_1 and l_2, respectively, and merge them using external atom $\&merge[s_1, s_2](s)$ to get the sorted version of l.

$$
\begin{aligned}
sorted(L, S) \leftarrow &sublist(L, H1, H2), sorted(H1, H1s), sorted(H2, H2s),\\
&\&merge[H1s, H2s](S)\\
output(Sorted) \leftarrow &list(Final), sorted(Final, Sorted)
\end{aligned}
$$

Note that both the splitting and the merging part uses external atoms in cycles, thus the program is not strongly safe. In order to make it strongly safe, we need to generate all lists (i.e., permutations of the input list) a priori and add domain predicates.

A.5 Argumentation with Subsequent Processing

This benchmark uses the encoding from Section A.1 extended by an additional rule which performs further computations over the ideal sets. For instance, we might use a rule of form

$$latex(L) \leftarrow \&toLatex[in_S, out_S](L)$$

to generate a LaTeX representation of the arguments that are in resp. not in the extension.

A.6 Route Planning

We present the encodings for the two route planning scenarios separately. Once the single route planning scenario was introduced, the extension to pair route planning is quite simple. In fact, we will present a more general encoding which allows for planning tours for an arbitrary number of persons.

A.6.1 Single Route Planning

We start by guessing a sequence of the locations which are defined in the instance by facts of kind $location(loc)$. The following rules ensure that for a set L of n locations, the atoms $seq(i, loc_i)$ for $0 \leq i < n$, order the locations such that $L = \{loc_i \mid 0 \leq i < n\}$.

$$\begin{aligned}
seq(I, L) \vee \overline{seq}(I, L) &\leftarrow location(L), \#int(I) && \text{(R1)}\\
&\leftarrow seq(I1, L), seq(I2, L), I1 \neq I2\\
&\leftarrow seq(I, L1), seq(I, L1), L1 \neq L2\\
haveSeq(L) &\leftarrow seq(I, L)\\
&\leftarrow location(L), \text{not } haveSeq(L)\\
haveLoc(I) &\leftarrow seq(I, L)\\
&\leftarrow seq(I, L), I1 < I, \#int(I1), \text{not } haveLoc(I1)
\end{aligned}$$

The following rules choose exactly one restaurant and add it to the set of locations to visit, if necessary. It is assumed that the instance defines possible locations for having the lunch by facts of kind $possibleRestaurant(r)$.

$$
\begin{aligned}
restaurant(R) \vee \overline{restaurant}(R) &\leftarrow haveLunch, possibleRestaurant(R)\\
restaurantChosen &\leftarrow restaurant(R)\\
&\leftarrow restaurant(R1), restaurant(R2), R1 \neq R2\\
&\leftarrow haveLunch, \text{not } restaurantChosen\\
location(R) &\leftarrow restaurant(R)
\end{aligned}
$$

We further need rules to check if our tour has to include a restaurant. The constant $limit$ is to be replaced by an integer which defines the maximal costs of a tour without restaurant. The external atom $\&longerThan[path, limit]()$ is true iff$_{\text{def}}$ the path encoded in the extension of $path$ is longer than $limit$. Note that despite the rules (R3) and (R4), the choice between $haveLunch$ and $\overline{haveLunch}$ in rule (R2) is not redundant due to the use of the FLP-reduct[1].

$$
\begin{aligned}
haveLunch \vee \overline{haveLunch} &\leftarrow && \text{(R2)}\\
haveLunch &\leftarrow \&longerThan[path, limit]() && \text{(R3)}\\
\overline{haveLunch} &\leftarrow \text{not } \&longerThan[path, limit]() && \text{(R4)}
\end{aligned}
$$

The following rules plan the tour using the external atom $\&path[L1, L2](X, Y, Cost, Type)$. Atoms of form $path(L1, L2, X, Y, Cost, Type)$ are used to encode the path (consisting of edges (X, Y) with costs $Cost$ and type $Type$) from $L1$ to $L2$. Since the same station may be visited multiple times, the end points $L1$ to $L2$ must be included to make the representation unique.

$$
\begin{aligned}
path(L1, L2, X, Y, Cost, Type) \leftarrow & seq(Nr, L1), seq(NrNext, L2), && \text{(R5)}\\
& NrNext = Nr + 1,\\
& \&path[L1, L2](X, Y, Cost, Type)
\end{aligned}
$$

Finally, we ensure that all pairs of sequent locations are connected. Otherwise, the program would still have answer sets (leaving some locations unconnected) if a location is not reachable.

$$
\begin{aligned}
pathExists(L1, L2) \leftarrow & seq(Nr, L1), seq(NrNext, L2), NrNext = Nr + 1,\\
& path(L1, L2, X, Y, Cost, Type)\\
\leftarrow & seq(Nr, L1), seq(NrNext, L2), NrNext = Nr + 1, && \text{(R6)}\\
& \text{not } pathExists(L1, L2)
\end{aligned}
$$

[1] Constraints are never contained in the FLP-reduct because their body is unsatisfied. Therefore constraint (R6) does not check the existence of paths in the reduct. But then, without further techniques, the FLP check could fail because the reduct has a smaller model which does not connect all locations. However, since rule (R1) enforces the same sequence of locations in the reduct and rule (R5) deterministically computes the shortest paths between two successive locations, this can only happen if the set of locations is different in the model of the reduct. But this is only possible if the model candidate represents a tour with a restaurant and the model of the reduct represents one without a restaurant or vice versa. This case is excluded by rule (R2) which ensures that the choice between $haveLunch$ and $\overline{haveLunch}$ is the same in both models; without rule (R2), it might happen that neither $haveLunch$ nor $\overline{haveLunch}$ is true in the model of the reduct since either rule (R3) or (R4) will be missing in the reduct due to unsatisfied body.

A.6.2 Pair and Group Route Planning

Compared to single route planning from Section A.6.1, the predicates $location$, seq, $haveSeq$, $haveLoc$ and $path$ are extended by an additional argument P, which is inserted at argument position 1. It allows for discriminating multiple persons. While we considered the special case of two persons in our benchmarks in Chapter 5, the encoding is strictly more general and allows arbitrary many persons who need to be defined as facts of kind $person(p)$.

$$
\begin{aligned}
seq(P,I,L) \vee \overline{seq}(P,I,L) \leftarrow & person(P), location(L), \#int(I) \\
\leftarrow & person(P), seq(P,I1,L), seq(P,I2,L), \\
& I1 \neq I2 \\
\leftarrow & person(P), seq(P,I,L1), seq(P,I,L1), \\
& L1 \neq L2 \\
haveSeq(P,L) \leftarrow & person(P), seq(P,I,L) \\
\leftarrow & person(P), location(P,L), \text{not } haveSeq(P,L) \\
haveLoc(P,I) \leftarrow & person(P), seq(P,I,L) \\
\leftarrow & person(P), seq(P,I,L), I1 < I, \#int(I1), \\
& \text{not } haveLoc(P,I1)
\end{aligned}
$$

Computing the path and ensuring its existence is extended to multiple persons as follows.

$$
\begin{aligned}
path(P,L1,L2,X,Y,Cost,Type) \leftarrow & person(P), \\
& seq(P,Nr,L1), seq(P,NrNext,L2), \\
& NrNext = Nr+1, \\
& \&path[L1,L2](X,Y,Cost,Type) \\
pathExists(P,L1,L2) \leftarrow & person(P), \\
& seq(P,Nr,L1), seq(P,NrNext,L2), \\
& NrNext = Nr+1, \\
& path(P,L1,L2,X,Y,Cost,Type) \\
\leftarrow & person(P), \\
& seq(P,Nr,L1), seq(P,NrNext,L2), \\
& NrNext = Nr+1, \\
& \text{not } pathExists(P,L1,L2)
\end{aligned}
$$

The following rules choose a meeting point and include it in the tour of each person. It is assumed that the possible meeting locations are defined by facts of form $possibleMeeting(m)$.

$$
\begin{aligned}
meeting(M) \vee \overline{meeting}(M) &\leftarrow possibleMeeting(M) \\
meetingChosen &\leftarrow meeting(M) \\
&\leftarrow meeting(M1), meeting(M2), M1 \neq M2 \\
&\leftarrow \text{not } meetingChosen \\
location(P, M) &\leftarrow person(P), meeting(M)
\end{aligned}
$$

If the tour is longer than the given limit $limit$, then the meeting location should be a restaurant. The external atom $\&longerThanForPerson[path, p, limit]()$ is true iff$_{\text{def}}$ the path for person p encoded in the extension of $path$ is longer than $limit$.

$$
\begin{aligned}
haveLunch \vee \overline{haveLunch} &\leftarrow \\
haveLunch &\leftarrow person(P), \&longerThanForPerson[path, P, limit]() \\
\overline{haveLunch} &\leftarrow person(P), \text{not } \&longerThanForPerson[path, P, limit]() \\
&\leftarrow haveLunch, meeting(M), \text{not } possibleRestaurant(M) \\
&\leftarrow \overline{haveLunch}, meeting(M), possibleRestaurant(M)
\end{aligned}
$$

As for the encoding from Section A.6, the choice between $haveLunch$ and $\overline{haveLunch}$ is not redundant.

Appendix B

Proofs

We now show the lengthy proofs of some lemmas, propositions and theorems of this thesis.

B.1 Characterization of Answer Sets using Unfounded Sets (cf. Section 3.2)

To show Theorem 2 one can use the proofs by Faber (2005) mutatis mutandi, with external atoms in place of aggregates. To make the thesis self-contained we include the proofs and start with some lemmas. Lemma B.1 and Lemma B.3 are equivalent to Theorem 4 and Theorem 5 by Faber (2005), respectively, Lemma B.2 is equivalent to Proposition 1, and Theorem 2 corresponds to Corollary 3.

Lemma B.1. *Given a total interpretation* $\mathbf{A}$ *and program* Π*,* $\mathbf{A^F}$ *is an unfounded set for* Π *wrt.* $\mathbf{A}$ *iff* $\mathbf{A}$ *is a model of* Π*.*

Proof. ($\Rightarrow$) For any rule, either (1) $H(r) \cap \mathbf{A^F} = \emptyset$, or (2) $H(r) \cap \mathbf{A^F} \neq \emptyset$. If (1), then $H(r) \cap \mathbf{A^T} \neq \emptyset$, i.e. the head is true and r is satisfied wrt. $\mathbf{A}$. If (2), then one of the conditions of Definition 38 must hold. If Condition (i) holds, the body is false wrt. $\mathbf{A}$ and r is satisfied wrt. $\mathbf{A}$. If Condition (ii) holds, a body literal is false wrt. $\mathbf{A} \dot\cup \neg.\mathbf{A^F} = \mathbf{A}$, so it coincides with Condition (i). If Condition (iii) holds, $H(r) \cap \mathbf{A^T} \neq \emptyset$, and therefore the rule is satisfied wrt. $\mathbf{A}$. In total, if $\mathbf{A^F}$ is an unfounded set for Π wrt. $\mathbf{A}$, all rules are satisfied wrt. $\mathbf{A}$, hence $\mathbf{A}$ is a model of Π.

($\Leftarrow$) If $\mathbf{A}$ is a model, all rules are satisfied, so for any rule r, either (1) $H(r) \cap \mathbf{A^T} \neq \emptyset$ or (2) if $H(r) \cap \mathbf{A^T} = \emptyset$ then a body literal is false wrt. $\mathbf{A}$. So also for any rule r with $H(r) \cap \mathbf{A^F} \neq \emptyset$, either (1) or (2) holds. If (1), then Condition (iii) of Definition 38 applies. If (2), then Condition (i) (and also Condition (ii), since $\mathbf{A} \dot\cup \neg.\mathbf{A^F} = \mathbf{A}$) applies. Therefore $\mathbf{A^F}$ is an unfounded set. □

Lemma B.2. *If U_1 and U_2 are unfounded sets of a program Π wrt. $\mathbf{A}$ and both $U_1 \cap \mathbf{A}^{\mathbf{T}} = \emptyset$ and $U_2 \cap \mathbf{A}^{\mathbf{T}} = \emptyset$, then $U_1 \cup U_2$ is an unfounded set of Π wrt. $\mathbf{A}$.*

Proof. Consider a rule r where $H(r) \cap U_1 \neq \emptyset$ (symmetric arguments hold for U_2). At least one of the conditions of Definition 38 holds wrt. U_1. We will show that the conditions also hold wrt. $U_1 \cup U_2$.

If Condition (i) holds wrt. U_1, then it trivially holds also for $U_1 \cup U_2$. If Condition (ii) holds, a body literal is false wrt. $\mathbf{A} \dot{\cup} \neg.U_1$. Then it is also false wrt. $\mathbf{A} \dot{\cup} \neg.U_1 \dot{\cup} \neg.U_2 = \mathbf{A} \dot{\cup} \neg.(U_1 \cup U_2)$ since $\mathbf{A}^{\mathbf{T}} \cap U_2 = \emptyset$ and thus $\mathbf{A} \dot{\cup} \neg.U_1 \dot{\cup} \neg.U_2 = \mathbf{A} \dot{\cup} \neg.U_1$. If Condition (iii) holds, some atom $a \in H(r) \setminus U_1$ is true wrt. $\mathbf{A}$, so $a \in \mathbf{A}^{\mathbf{T}}$. It follows that $a \notin U_2$, and so $H(r) \setminus (U_1 \cup U_2)$ is still true wrt. $\mathbf{A}$. □

Lemma B.2 implies in particular that if U_1 and U_2 are unfounded sets of a program Π wrt. an unfounded-free interpretation $\mathbf{A}$, then also $U_1 \cup U_2$ is an unfounded set of Π wrt. $\mathbf{A}$ (Corollary 2 by Faber (2005)). This allows for a program Π and an unfounded-free interpretation $\mathbf{A}$ to define the *greatest unfounded set* $GUS_\Pi(\mathbf{A})$ (the GUS for Π wrt. $\mathbf{A}$) as the union of all unfounded sets for Π wrt. $\mathbf{A}$ (Definition 3 by Faber (2005)); for interpretations $\mathbf{A}$ which are not unfounded-free, $GUS_\Pi(\mathbf{A})$ is undefined.

Lemma B.3. *A total interpretation $\mathbf{A}$ is an answer set of Π iff $\mathbf{A}^{\mathbf{F}} = GUS_\Pi(\mathbf{A})$.*

Proof. ($\Rightarrow$) If $\mathbf{A}$ is an answer set, it is also a model of Π, so by Lemma B.1, $\mathbf{A}^{\mathbf{F}}$ is an unfounded set for Π wrt. $\mathbf{A}$. We next show that $\mathbf{A}$ is unfounded-free wrt. Π, from which $\mathbf{A}^{\mathbf{F}} = GUS_\Pi(\mathbf{A})$ follows. Let us assume an unfounded set U for Π wrt. $\mathbf{A}$ exists such that $\mathbf{A}^{\mathbf{T}} \cap U \neq \emptyset$. We can show that then $\mathbf{A} \dot{\cup} \neg.U$ is a model of $f\Pi^{\mathbf{A}}$, contradicting the fact that $\mathbf{A}$ is an answer set of Π. First note that for any rule r in $f\Pi^{\mathbf{A}}$, all body literals are true wrt. $\mathbf{A}$ (by construction of $f\Pi^{\mathbf{A}}$), and $H(r) \cap \mathbf{A}^{\mathbf{T}} \neq \emptyset$ (since $\mathbf{A}$ is a model of $f\Pi^{\mathbf{A}}$). We differentiate two cases: (1) $H(r) \cap (\mathbf{A} \dot{\cup} \neg.U)^{\mathbf{T}} \neq \emptyset$ and (2) $H(r) \cap (\mathbf{A} \dot{\cup} \neg.U)^{\mathbf{T}} = \emptyset$. For (1), r is trivially satisfied by $\mathbf{A} \dot{\cup} \neg.U$. For (2), since we know $H(r) \cap \mathbf{A}^{\mathbf{T}} \neq \emptyset$, $H(r) \cap U \neq \emptyset$ must hold. Since U is an unfounded set wrt. Π and $\mathbf{A}$ (and $r \in \Pi$), a body literal of r must be false wrt. $\mathbf{A} \dot{\cup} \neg.U$ (note that neither a body literal of r is false wrt. $\mathbf{A}$ since $r \in f\Pi^{\mathbf{A}}$, nor $(H(r) \setminus U) \cap \mathbf{A}^{\mathbf{T}} \neq \emptyset$ holds, otherwise $H(r) \cap (\mathbf{A} \dot{\cup} \neg.U)^{\mathbf{T}} \neq \emptyset$). So r is satisfied also in Case (2). $\mathbf{A} \dot{\cup} \neg.U$ is therefore a model of $f\Pi^{\mathbf{A}}$, and since $(\mathbf{A} \dot{\cup} \neg.U)^{\mathbf{T}} \subsetneq \mathbf{A}^{\mathbf{T}}$, $\mathbf{A}$ is not a minimal model of $f\Pi^{\mathbf{A}}$, contradicting that $\mathbf{A}$ is an answer set of Π.

($\Leftarrow$) By Lemma B.1 if $\mathbf{A}^{\mathbf{F}}$ is an unfounded set for Π wrt. $\mathbf{A}$, $\mathbf{A}$ is a model of Π, so it is also a model of $f\Pi^{\mathbf{A}}$. We show by contradiction that it is in fact a minimal model of $f\Pi^{\mathbf{A}}$. Assume that a total interpretation $\mathbf{A}'$, where $\mathbf{A}'^{\mathbf{T}} \subsetneq \mathbf{A}^{\mathbf{T}}$, is a model of $f\Pi^{\mathbf{A}}$. Since both $\mathbf{A}'$ and $\mathbf{A}$ are total, $\mathbf{A}'^{\mathbf{F}} \supsetneq \mathbf{A}^{\mathbf{F}}$. Again by Lemma B.1, $\mathbf{A}'^{\mathbf{F}}$ is an unfounded set for $f\Pi^{\mathbf{A}}$ wrt. $\mathbf{A}'$. We can then show that $\mathbf{A}'^{\mathbf{F}}$ is also an unfounded set for Π wrt. $\mathbf{A}$, contradicting the fact that $\mathbf{A}^{\mathbf{F}}$ is $GUS_\Pi(\mathbf{A})$. For any rule in $\Pi \setminus f\Pi^{\mathbf{A}}$, a body literal is false wrt. $\mathbf{A}$, so Condition (i) of Definition 38 holds. For a rule $r \in f\Pi^{\mathbf{A}}$ such that $H(r) \cap \mathbf{A}'^{\mathbf{F}} \neq \emptyset$, (1) a body literal of r is false wrt. $\mathbf{A}'$ (note that $\mathbf{A}' \dot{\cup} \neg.\mathbf{A}'^{\mathbf{F}} = \mathbf{A}'$) or (2) an atom a in $H(r) \setminus \mathbf{A}'^{\mathbf{F}}$ is true wrt. $\mathbf{A}'$. Concerning (1), observe that $\mathbf{A} \dot{\cup} \neg.\mathbf{A}'^{\mathbf{F}} = \mathbf{A}'$ so (1) holds iff a body atom is false wrt. $\mathbf{A} \dot{\cup} \neg.\mathbf{A}'^{\mathbf{F}}$. Concerning (2), since $\mathbf{A}'^{\mathbf{T}} \subsetneq \mathbf{A}^{\mathbf{T}}$, atom a is also true wrt. $\mathbf{A}$. In total,we have

shown that $\mathbf{A'^F}$ is an unfounded set for Π wrt. $\mathbf{A}$, a contradiction to $\mathbf{A^F} = GUS_\Pi(\mathbf{A})$. So $\mathbf{A}$ is indeed a minimal model of $f\Pi^{\mathbf{A}}$, and hence an answer set of Π. □

We can now show the main theorem.

Theorem 2. *A model* $\mathbf{A}$ *of a program* Π *is an answer set iff it is* unfounded-free *(in* Π*).*

Proof. ($\Rightarrow$) Since $\mathbf{A}$ is an answer set, by Lemma B.3 we have $\mathbf{A^F} = GUS_\Pi(\mathbf{A})$. But this implies that $GUS_\Pi(\mathbf{A})$ is defined, which is only the case if $\mathbf{A}$ is unfounded-free.

($\Leftarrow$) Since $\mathbf{A}$ is unfounded-free, we have that $\mathbf{A^T} \cap U = \emptyset$ and thus $U \subseteq \mathbf{A^F}$ for each unfounded set U of Π wrt. $\mathbf{A}$. But then $\mathbf{A^F} \supseteq GUS_\Pi(\mathbf{A})$. Moreover, since $\mathbf{A}$ is a model, by Lemma B.1 $\mathbf{A^F}$ is an unfounded set of Π wrt. $\mathbf{A}$ and thus $\mathbf{A^F} \subseteq GUS_\Pi(\mathbf{A})$. But then $\mathbf{A^F} = GUS_\Pi(\mathbf{A})$ and by Lemma B.3 $\mathbf{A}$ is an answer set of Π. □

B.2 Soundness and Completeness of the Grounding Algorithm (cf. Section 4.3.2)

We now formally prove that Algorithms GroundHEXNaive and GroundHEX are sound and complete as stated by Proposition 4.6 and Theorem 6, respectively. As the programs Π_p and Π_{pg} are iteratively updated in the algorithms, we make the following convention. Whenever we write Π_p or Π_{pg} in one of the proofs, we refer to the status just before Algorithm GroundHEXNaive or Algorithm GroundHEX returns.

A key concept in our proofs will be that of *representation* of external atoms in a ground program.

Definition 93. For a ground external atom $\&g[\mathbf{y}](\mathbf{x})$ in a rule r, its *representation degree in a program* Π is 0, if Π contains a rule $e_{r,\&g[\mathbf{y}]}(\mathbf{x}) \vee ne_{r,\&g[\mathbf{y}]}(\mathbf{x}) \leftarrow$. It is $n+1$, if Π contains a rule with head $e_{r,\&g[\mathbf{y}]}(\mathbf{x}) \vee ne_{r,\&g[\mathbf{y}]}(\mathbf{x})$ and the maximum representation degree of all $\&h[\mathbf{w}](\mathbf{v})$ s.t. $e_{s,\&h[\mathbf{w}]}(\mathbf{v})$ occurs in the body of this rule, is n. Otherwise (i.e., there is no rule with head $e_{r,\&g[\mathbf{y}]}(\mathbf{x}) \vee ne_{r,\&g[\mathbf{y}]}(\mathbf{x})$), the representation degree is *undefined*.

If the representation degree for some ground external atom is undefined, we also say that the external atom is *not represented*. Intuitively, if an external atom is represented, this means that the program contains a guessing rule for the respective replacement atom. The representation degree specifies on how many other external atom replacements this guess depends. Note that in general, an external atom can have multiple representation degrees simultaneously. However, in the following we will only use its *minimum representation degree* and can therefore drop the prefix *minimum*.

In the proofs of the grounding algorithms, we will usually denote assignments as sets of atoms which are true. All other atoms are then implicitly false.

Towards a proof of Proposition 4.6 we first prove the following lemma. Recall that, whenever we write Π_p or Π_{pg} in the proofs, we refer to the status just before Algorithm GroundHEXNaive returns.

Lemma B.4. *Let $\Pi_g = GroundHEXNaive(\Pi)$ and let C be the constants which appear in Π_g. Then for any $C' \supseteq C$ and each model $\mathbf{A}$ of $grnd_C(\Pi)$, $\mathbf{A} \not\models B(r)$ for all $r \in grnd_{C'}(\Pi) \setminus grnd_C(\Pi)$.*

Proof. Let $\mathbf{A}$ be a model of $grnd_C(\Pi)$. Then it can be extended to a model $\mathbf{A}_{pg}$ of Π_{pg} as follows:

- For all $e_{r,\&g[\mathbf{y}]}(\mathbf{x}) \in A(\Pi_{pg})$, add $e_{r,\&g[\mathbf{y}]}(\mathbf{x})$ if $f_{\&g}(\mathbf{A}, \mathbf{y}, \mathbf{x}) = 1$ and add $ne_{r,\&g[\mathbf{y}]}(\mathbf{x})$ otherwise.
- Add all $g^{\&g}_{inp}(\mathbf{y}) \in A(\Pi_{pg})$, for all predicates $g^{\&g}_{inp}$ occurring in the head of some r^a_{inp} (for an external atom $a = \&g[\mathbf{Y}](\mathbf{X})$).

This satisfies each ground instance of each input auxiliary rule r^a_{inp} because the head $g^{\&g}_{inp}(\mathbf{y})$ is true. Moreover, because $\mathbf{A}$ is a model of $grnd_C(\Pi) = \Pi_g$ and Π_{pg} contains $e_{r,\&g\mathbf{y}}(\mathbf{x})$ in place of $\&g[\mathbf{y}](\mathbf{x})$ and we set $e_{r,\&g[\mathbf{y}]}(\mathbf{x})$ to true iff $f_{\&g}(\mathbf{A}, \mathbf{y}, \mathbf{x}) = 1$, it satisfies also all remaining rules.

We show now that $\mathbf{A}$ is also a model of $grnd_{C'}(\Pi)$. Let $r \in grnd_{C'}(\Pi)$ and suppose $\mathbf{A} \not\models r$, then $\mathbf{A} \not\models H(r)$ and $\mathbf{A} \models B(r)$. Since $\mathbf{A} \models B(r)$, we have $\mathbf{A} \models a$ for each ordinary literal $a \in B(r)$. If there would be only ordinary literals in $B(r)$, then Π_g would also contain this rule instance because all constants in $B(r)$ must appear in the atoms which are true in $\mathbf{A}$ und thus in Π_g. Hence, $\mathbf{A}$ could not be a model of Π_g. Therefore there must be external atoms in $B(r)$.

We show now that each positive external atom in r is represented in Π_{pg} (with degree 0). Suppose there is an external atom in $B(r)$ which is not represented in Π_{pg}. Then, due to safety of r, which forbids cyclic passing of constant input within a rule body, there is also a 'first' unrepresented external atom $\&g[\mathbf{v}](\mathbf{u})$, i.e., one such that all its input constants in $\mathbf{v}$ either: (1) appear in a positive ordinary atom, (2) appear in the output list of a represented external atom, or (3) were already constants in the input program. In all three cases, the input auxiliary rule for $\&g[\mathbf{v}](\mathbf{u})$ is instantiated for this $\mathbf{v}$ because its body atoms are potentially true (they are ordinary atoms or replacement atoms of represented external atoms), i.e., $g^{\&g}_{inp}(\mathbf{v})$ appears in the program and is therefore true in $\mathbf{A}_{pg}$. Thus, the loop at (e) would evaluate $\&g$ with $\mathbf{A}_{pg}$ and $\mathbf{v}$ and determine all tuples $\mathbf{w}$ s.t. $f_{\&g}(\mathbf{A}_{pg}, \mathbf{v}, \mathbf{w}) = 1$. However, $f_{\&g}(\mathbf{A}_{pg}, \mathbf{v}, \mathbf{u}) = 0$, because otherwise rule $e_{r,\&g[\mathbf{v}]}(\mathbf{u}) \vee ne_{r,\&g[\mathbf{v}]}(\mathbf{u}) \leftarrow$ would have been added at (f) to Π_p and thus $\&g[\mathbf{v}](\mathbf{u})$ would be represented in Π_{pg}, which contradicts our assumption. But if $f_{\&g}(\mathbf{A}_{pg}, \mathbf{v}, \mathbf{u}) = 0$ then also $f_{\&g}(\mathbf{A}, \mathbf{v}, \mathbf{u}) = 0$ because $\mathbf{A}_{pg}$ and $\mathbf{A}$ differ only on input auxiliary atoms and external atom replacement atoms, which would imply $\mathbf{A} \not\models B(r)$.

Thus, all positive external atoms are represented in Π_{pg}. But as default-negated ones cannot introduce new values due to ordinary safety, all constants in r also appear in Π_g, thus $r \in \Pi_g$. But then $\mathbf{A}$ could not be a model of Π_g if $\mathbf{A} \models B(r)$, hence $\mathbf{A} \not\models B(r)$. □

Now we can now show the proposition.

Proposition 4.6. *If Π is a liberally de-safe* HEX*-program, then $\Pi \equiv^{pos} GroundHEXNaive(\Pi)$.*

Proof. Let $\Pi_g = \text{GroundHEXNaive}(\Pi)$. For the proof, observe that $\Pi_g = grnd_C(\Pi)$ where C is the set of all constants which appear in Π_g. We show now

$$grnd_C(\Pi) \equiv^{pos} grnd_{C'}(\Pi)$$

for any $C' \supseteq C$. Because $\Pi \equiv^{pos} grnd_{\mathcal{C}}(\Pi)$ for Herbrand universe $\mathcal{C} \supseteq C$ by definition of the HEX-semantics, this implies the proposition.

Termination of the algorithm follows from Theorem 6, where we will prove that an optimized version of the algorithm, which may produce a larger grounding (wrt. the number of constants) but need less iterations, terminates. As the grounding produced by this algorithm is even smaller, it terminates as well.

($\Rightarrow$) Let $\mathbf{A} \in \mathcal{AS}\big(grnd_C(\Pi)\big)$. By Lemma B.4 it is also a model of $grnd_{C'}(\Pi)$. It remains to show that it is also a subset-minimal model of $fgrnd_{C'}(\Pi)^{\mathbf{A}}$. Since $C \subseteq C'$, $fgrnd_C(\Pi)^{\mathbf{A}} \subseteq fgrnd_{C'}(\Pi)^{\mathbf{A}}$. By Lemma B.4, $\mathbf{A} \not\models B(r)$ for any $r \in grnd_{C'}(\Pi) \setminus grnd_C(\Pi)$, thus $fgrnd_C(\Pi)^{\mathbf{A}} = fgrnd_{C'}(\Pi)^{\mathbf{A}}$. But since $\mathbf{A} \in \mathcal{AS}\big(grnd_C(\Pi)\big)$, it is a minimal model of $fgrnd_C(\Pi)^{\mathbf{A}}$, thus also of $fgrnd_{C'}(\Pi)^{\mathbf{A}}$, i.e., $\mathbf{A} \in \mathcal{AS}\big(grnd_{C'}(\Pi)\big)$.

($\Leftarrow$) Let $\mathbf{A}' \in \mathcal{AS}\big(grnd_{C'}(\Pi)\big)$. We show that $\mathbf{A} = \mathbf{A}' \cap A\big(grnd_C(\Pi)\big)$ is an answer set of $grnd_C(\Pi)$. Because $grnd_C(\Pi) \subseteq grnd_{C'}(\Pi)$, it is trivial that $\mathbf{A}$ is a model of $grnd_C(\Pi)$. It remains to show that it is also a subset-minimal model of $fgrnd_C(\Pi)^{\mathbf{A}}$. By Lemma B.4, $\mathbf{A}$ is a model of $grnd_{C'}(\Pi)$. Clearly, $\mathbf{A} \subseteq \mathbf{A}'$. But $\mathbf{A} \subsetneq \mathbf{A}'$ would imply that $\mathbf{A}'$ is not subset-minimal, which contradicts the assumption that it is an answer set of $grnd_{C'}(\Pi)$, thus $\mathbf{A} = \mathbf{A}'$. Because $grnd_C(\Pi) \subseteq grnd_{C'}(\Pi)$ and $\mathbf{A} \not\models B(r)$ for all $r \in grnd_{C'}(\Pi) \setminus grnd_C(\Pi)$, we have $fgrnd_C(\Pi)^{\mathbf{A}} = fgrnd_{C'}(\Pi)^{\mathbf{A}}$. Because $\mathbf{A}$ is a subset-minimal model of $grnd_{C'}(\Pi)^{\mathbf{A}}$, it is a subset-minimal model of $fgrnd_C(\Pi)^{\mathbf{A}}$. Thus, $\mathbf{A}$ is an answer set of $grnd_C(\Pi)$. □

In order to prove soundness and completeness of our optimized algorithm in Theorem 6, we first show a lemma analogous to Lemma B.4. Recall that, whenever we write Π_p or Π_{pg} in the proofs, we refer to the status just before Algorithm GroundHEX returns.

Lemma B.5. *Let $\Pi_g = \text{GroundHEX}(\Pi)$ and let C be the constants which appear in Π_g. Then for any $C' \supseteq C$ and each model $\mathbf{A}$ of $grnd_C(\Pi)$, $\mathbf{A} \not\models B(r)$ for all $r \in grnd_{C'}(\Pi) \setminus grnd_C(\Pi)$.*

Proof. Let $\mathbf{A}$ be an model of $grnd_C(\Pi)$. Then it can be extended to a model $\mathbf{A}_{pg}$ of Π_{pg} as follows:

- For all $e_{r,\&g[\mathbf{y}]}(\mathbf{x}) \in A(\Pi_{pg})$, add $e_{r,\&g[\mathbf{y}]}(\mathbf{x})$ if $f_{\&g}(\mathbf{A}_g, \mathbf{y}, \mathbf{x}) = 1$ and add $ne_{r,\&g[\mathbf{y}]}(\mathbf{x})$ otherwise.
- Add all $g^{\&g}_{inp}(\mathbf{y}) \in A(\Pi_{pg})$, for all predicates $g^{\&g}_{inp}$ occurring in the head of some r^a_{inp} (for an external atom $a = \&g[\mathbf{Y}](\mathbf{X})$).

This satisfies each external atom guessing rule as for $\&g[\mathbf{y}](\mathbf{x})$ either $e_{r,\&g[\mathbf{y}]}(\mathbf{x})$ or $ne_{r,\&g[\mathbf{y}]}(\mathbf{x})$ is true, and each input auxiliary rule r^a_{inp} because the head $g^{\&g}_{inp}(\mathbf{y})$ is true. Moreover, because $\mathbf{A}$ is a model of $grnd_C(\Pi)$, it is also a model of the (possibly) less restrictive program Π_g. Since Π_{pg} contains $e_{r,\&g\mathbf{y}}(\mathbf{x})$ in place of $\&g[\mathbf{y}](\mathbf{x})$ and we set $e_{r,\&g[\mathbf{y}]}(\mathbf{x})$ in $\mathbf{A}_{pg}$ to true iff $f_{\&g}(\mathbf{A}, \mathbf{y}, \mathbf{x}) = 1$, $\mathbf{A}_{pg}$ satisfies also all remaining rules in program Π_{pg}.

We show now that $\mathbf{A}$ is a model of $grnd_{C'}(\Pi)$. Let $r \in grnd_{C'}(\Pi)$ and suppose $\mathbf{A} \not\models r$, i.e., $\mathbf{A} \not\models H(r)$ but $\mathbf{A} \models B(r)$.

As we have seen in the proof of Proposition 4.6, all de-safety relevant positive external atoms in r are represented with degree 0 in the program computed by Algorithm GroundHEXNaive. As such external atoms are handled equivalently by our optimized algorithm, they are also represented in Π_{pg}. We show that this holds also for positive external atoms which are not de-safety relevant.

Suppose r contains an external atom which is not de-safety relevant and which is not represented in Π_{pg}. Then there is a 'first' such external atom $\&g[\mathbf{v}](\mathbf{u})$ in $B(r)$, i.e., its input list only contains constants which (1) appear in ordinary atoms, (2) appear in de-safety relevant external atoms, or (3) were already constants in the input program. In all three cases, the input auxiliary rule for $\&g[\mathbf{v}](\mathbf{u})$ is instantiated for this $\mathbf{v}$ because its body atoms are potentially true (ordinary atoms appear also in $B(r)$ and are potentially true, otherwise r would not have been added to Π_g; external atoms are all not de-safety relevant and are potentially true since they are represented with degree 0), i.e., $g^{\&g}_{inp}(\mathbf{v})$ appears in the program. Moreover, the respective external atom guessing rule is instantiated for $\mathbf{v}$ and $\mathbf{u}$ because all its body atoms are potentially true (with the same argument as for input auxiliary rules). Thus, $\&g[\mathbf{v}](\mathbf{u})$ would be represented in Π_p with some degree > 0, and thus also in Π_{pg} and Π_g.

Thus, all positive external atoms are represented in Π_{pg}. But as default-negated ones cannot introduce new values due to ordinary safety, all constants in r also appear in Π_g, thus a strengthening of r would be in Π_g. But then $\mathbf{A}$ could not be a model of Π_g if $\mathbf{A} \models B(r)$, hence $\mathbf{A} \not\models B(r)$. □

We now show some additional lemmas to simplify the proof of soundness and completeness.

Lemma B.6. *Let $\Pi_g = GroundHEX(\Pi)$ and let C be the constants which appear in Π_g. Every answer set $\mathbf{A}$ of Π_g can be extended to an answer set $\mathbf{A}_{pg}$ of Π_{pg}.*

Proof. Let $\mathbf{A} \in \mathcal{AS}(\Pi_g)$. Then $\mathbf{A}_{pg}$ is constructed by iteratively adding additional atoms to $\mathbf{A}$ as follows:

- If the body B of a ground external atom guessing rule $e_{r,\&g[\mathbf{y}]}(\mathbf{x}) \vee ne_{r,\&g[\mathbf{y}]}(\mathbf{x}) \leftarrow B$ in Π_{pg} is satisfied by $\mathbf{A}_{pg}$, add $e_{r,\&g[\mathbf{y}]}(\mathbf{x})$ if $f_{\&g}(\mathbf{A}, \mathbf{y}, \mathbf{x}) = 1$ and add $ne_{r,\&g\mathbf{y}}(\mathbf{x})$ otherwise.
- Add all $g^{\&g}_{inp}(\mathbf{y}) \in A(\Pi_{pg})$ if the body of the respective input auxiliary rule is satisfied by $\mathbf{A}_{pg}$.

Note that this operation is monotonic because input auxiliary rules and external atom guessing rules contain only positive body literals.

Then the fixpoint of this operation $\mathbf{A}_{pg}$ is by construction a model of all input auxiliary rules and external atom guessing rules. Moreover, it is also a model of all remaining rules because $\mathbf{A}$ is a model of the corresponding rules in Π_g with external atoms in place of replacement atoms, and we set the truth values of the external atom replacement atoms exactly to the truth values of the external atoms in $\mathbf{A}$. Note that there might be external atoms in Π_g for which neither $e_{r,\&g[\mathbf{y}]}(\mathbf{x})$ nor $ne_{r,\&g[\mathbf{y}]}(\mathbf{x})$ is added to $\mathbf{A}_{pg}$, but then the body of the respective external atom

guessing rule is unsatisfied by $\mathbf{A}_{pg}$. But since the body of an external atom guessing rule is a subset of the body of the rule where this external atom occurs, also this rule is satisfied.

It remains to show that $\mathbf{A}_{pg}$ is also a subset-minimal model of $f\Pi_{pg}{}^{\mathbf{A}_{pg}}$. Suppose there is a smaller model $\mathbf{A}'_{pg} \subsetneq \mathbf{A}_{pg}$. Then $\mathbf{A}_{pg} \setminus \mathbf{A}'_{pg}$ must contain at least one atom which is not a replacement atom or an input auxiliary atom, because by construction of $\mathbf{A}_{pg}$ such atoms are only set to true if necessary, i.e., if they are supported by $\mathbf{A}$, and all rules used to derive such atoms are also in $f\Pi_{pg}{}^{\mathbf{A}_{pg}}$. We now show that the restriction of $\mathbf{A}$ to ordinary atoms $\mathbf{A}' \subsetneq \mathbf{A}$ (i.e., without replacement atoms $e_{\&g[\mathbf{y}]}(\mathbf{x})$ and $ne_{\&g[\mathbf{y}]}(\mathbf{x})$ and without external atom input atoms $g^{\&g}_{inp}(\mathbf{y})$) is a model of $f\Pi_g{}^{\mathbf{A}}$, which contradicts the assumption that $\mathbf{A}$ is an answer set of Π_g.

Observe that, except for the external atom guessing and input auxiliary rules, the reduct $f\Pi_{pg}{}^{\mathbf{A}_{pg}}$ contains the same rules as $f\Pi_g{}^{\mathbf{A}}$ with replacement atoms instead of external atoms. Thus, for $r \in f\Pi_g{}^{\mathbf{A}}$, the corresponding $r_{pg} \in f\Pi_{pg}{}^{\mathbf{A}_{pg}}$ contains the same ordinary literals in the head and body.

We show now that $\mathbf{A}'_{pg} \models r_{pg}$ implies $\mathbf{A}' \models r$. If $\mathbf{A}'_{pg}$ is a model of r_{pg}, then we have either (1) $\mathbf{A}'_{pg} \models h$ for some $h \in H(r_{pg})$, or (2) $\mathbf{A}'_{pg} \not\models b$ for some $b \in B(r_{pg})$. In Case (1), we also have $h \in H(r)$. Since $\mathbf{A}'_{pg}$ and $\mathbf{A}'$ coincide on non-replacement and non-input atoms, this implies $\mathbf{A}' \models r$. In Case (2), b is either (2a) a non-replacement literal, or (2b) a (positive or default-negated) external atom replacement. In Case (2a), we also have $b \in B(r)$. Since $\mathbf{A}'_{pg}$ and $\mathbf{A}'$ coincide on such atoms, this implies $\mathbf{A}' \models r$. In Case (2b), we either have (2b′) $\mathbf{A}_{pg} \not\models b$, or (2b″) b is positive (since a default-negated atom cannot become false by removing atoms from the interpretation) and some literal b' in the body of the external atom guessing or in the input rule for b is false in $\mathbf{A}'_{pg}$; in this case b is represented in Π_p with some degree n. In Case (2b′), $\mathbf{A}$ falsifies by construction of $\mathbf{A}_{pg}$ the external atom in $B(r)$ which corresponds to the replacement atom b. In Case (2b″), b' also appears in $B(r_{pg})$. Note that b' can be another external replacement atom. But in this case, the external atom corresponding to b' is represented with some degree $< n$. Thus, we start the case distinction for b' again. However, because the degree is reduced with every iteration, we will eventually end up in one of the other cases.

Thus, $\mathbf{A}'$ would be a model of $f\Pi_g{}^{\mathbf{A}}$, which contradicts the assumption that $\mathbf{A}$ is an answer set of Π_g. This shows that $\mathbf{A}_{pg}$ is an answer set of Π_{pg}. □

Lemma B.7. *Let $\Pi_g = GroundHEX(\Pi)$ and let C be the constants which appear in Π_g. Every answer set $\mathbf{A}$ of $grnd_C(\Pi)$ can be extended to an answer set $\mathbf{A}_p$ of $grnd_C(\Pi_p)$.*

Proof. Let $\mathbf{A} \in \mathcal{AS}\big(grnd_C(\Pi)\big)$. Then $\mathbf{A}_p$ is constructed by iteratively adding additional atoms to $\mathbf{A}$ as follows:

- If the body B of a ground external atom guessing rule $e_{r,\&g[\mathbf{y}]}(\mathbf{x}) \vee ne_{r,\&g[\mathbf{y}]}(\mathbf{x}) \leftarrow B$ in $grnd_C(\Pi_p)$ is satisfied by $\mathbf{A}_p$, add $e_{r,\&g[\mathbf{y}]}(\mathbf{x})$ if $f_{\&g}(\mathbf{A}, \mathbf{y}, \mathbf{x}) = 1$ and add $ne_{r,\&g[\mathbf{y}]}(\mathbf{x})$ otherwise.
- Add all $g^{\&g}_{inp}(\mathbf{y}) \in A\big(grnd_C(\Pi_p)\big)$ if the body of the respective input auxiliary rule is satisfied by $\mathbf{A}_p$.

Note that this operation is monotonic because input auxiliary rules and external atom guessing rules contain only positive body literals.

Then the fixpoint of this operation $\mathbf{A}_p$ is by construction a model of all ground input auxiliary rules and external atom guessing rules. Moreover, it is also a model of all remaining rules in $grnd_C(\Pi_p)$ because $\mathbf{A}$ is a model of the corresponding rules in $grnd_C(\Pi)$ with external atoms in place of replacement atoms, and we set the truth values of the external atom replacement atoms exactly to the truth values of the external atoms in $\mathbf{A}$. Note that there might be external atoms $\&g[\mathbf{y}](\mathbf{x})$ for which neither $e_{r,\&g[\mathbf{y}]}(\mathbf{x})$ nor $ne_{r,\&g[\mathbf{y}]}(\mathbf{x})$ is added to $\mathbf{A}_p$, but then the body of the respective external atom guessing rule is unsatisfied by $\mathbf{A}_p$. But since the body of an external atom guessing rule is a subset of the body of the rule where this external atom occurs, also this rule is satisfied.

Thus $\mathbf{A}_p$ is a model of $grnd_C(\Pi_p)$. It remains to show that it is also a subset-minimal model of $fgrnd_C(\Pi_p)^{\mathbf{A}_p}$. Suppose there is a smaller model $\mathbf{A}'_p \subsetneq \mathbf{A}_p$. Then $\mathbf{A}_p \setminus \mathbf{A}'_p$ must contain at least one atom which is not a replacement atom or an input auxiliary atom, because by construction of $\mathbf{A}_p$ such atoms are only set to true if necessary, i.e., if they are supported by $\mathbf{A}$, and all rules used to derive such atoms are also in $fgrnd_C(\Pi_p)^{\mathbf{A}_p}$. We now show that the restriction of $\mathbf{A}$ to ordinary atoms $\mathbf{A}' \subsetneq \mathbf{A}$ (i.e., without replacement atoms $e_{\&g[\mathbf{y}]}(\mathbf{x})$ and $ne_{\&g[\mathbf{y}]}(\mathbf{x})$ and without external atom input atoms $g^{\&g}_{inp}(\mathbf{y})$) is a model of $fgrnd_C(\Pi)^{\mathbf{A}}$, which contradicts the assumption that $\mathbf{A}$ is an answer set of $grnd_C(\Pi)$.

Observe that, except for the external atom guessing and input auxiliary rules, the reduct $fgrnd_C(\Pi_p)^{\mathbf{A}_p}$ contains the same rules as $fgrnd_C(\Pi)^{\mathbf{A}}$ with replacement atoms instead of external atoms. Thus, for $r \in fgrnd_C(\Pi)^{\mathbf{A}}$, the corresponding $r_p \in fgrnd_C(\Pi_p)^{\mathbf{A}_p}$ contains the same ordinary literals in the head and body.

We show now that $\mathbf{A}'_p \models r_p$ implies $\mathbf{A}' \models r$. If $\mathbf{A}'_p$ is a model of r_p, then we have either (1) $\mathbf{A}'_p \models h$ for some $h \in H(r_p)$, or (2) $\mathbf{A}'_p \not\models b$ for some $b \in B(r_p)$. In Case (1), we also have $h \in H(r)$. Since $\mathbf{A}'_p$ and $\mathbf{A}'$ coincide on non-replacement and non-input atoms, this implies $\mathbf{A}' \models r$. In Case (2), b is either (2a) a non-replacement literal, or (2b) a (positive or default-negated) external atom replacement. In Case (2a), we also have $b \in B(r)$. Since $\mathbf{A}'_p$ and $\mathbf{A}'$ coincide on ordinary atoms, this implies $\mathbf{A}' \models r$. In Case (2b), we either have (2b′) $\mathbf{A}_p \not\models b$, or (2b″) b is positive (since a default-negated atom cannot become false by removing atoms from the interpretation) and some literal b' in the body of the external atom guessing or in the input rule for b is false in $\mathbf{A}'_p$; in this case b is represented in $grnd_C(\Pi_p)$ with some degree n. In Case (2b′), $\mathbf{A}$ falsifies by construction of $\mathbf{A}_p$ the external atom in $B(r)$ which corresponds to the replacement atom b. In Case (2b″), b' also appears in $B(r_p)$. Note that b' can be another external replacement atom. But in this case, the external atom corresponding to b' is represented with some degree $< n$. Thus, we start the case distinction for b' again. However, because the degree is reduced with every iteration, we will eventually end up in one of the other cases.

Thus, $\mathbf{A}'$ would be a model of $fgrnd_C(\Pi)^{\mathbf{A}}$, which contradicts the assumption that $\mathbf{A}$ is an answer set of $grnd_C(\Pi)$. This shows that $\mathbf{A}_p$ is an answer set of $grnd_C(\Pi_p)$. □

Lemma B.8. *Let $\Pi_g = GroundHEX(\Pi)$ and let C be the constants which appear in Π_g. It holds that $\mathcal{AS}(\Pi_g) = \mathcal{AS}\big(grnd_C(\Pi)\big)$.*

Proof. ($\Rightarrow$) Let $\mathbf{A} \in \mathcal{AS}(\Pi_g)$. Then by Lemma B.6 it can be extended to an answer set $\mathbf{A}_{pg}$ of Π_{pg}. By Definition 75, $\mathbf{A}_{pg}$ is also an answer set of $grnd_C(\Pi_p)$.

Let $r \in grnd_C(\Pi)$ and let r' be the respective rule in $grnd_C(\Pi_p)$ with replacement atoms instead of external atoms. (1) If there is no strengthening of r in Π_g, then there is also no strengthening of r' in Π_{pg}. Then by Definition 75, every answer set of $grnd_C(\Pi_p)$ falsifies some ordinary body literal of r'. Thus this holds also for $\mathbf{A}_{pg}$. But all ordinary literals of r' are also in r and $\mathbf{A}$ coincides with $\mathbf{A}_{pg}$ on ordinary literals, thus $\mathbf{A}$ is a model of r. (2) If there is a strengthening $\bar{r}$ of r in Π_g, then there is also a strengthening $\bar{r'}$ of r' in Π_{pg} from which $\bar{r}$ was generated by replacing external replacement atoms by external atoms. Because $\mathbf{A}_{pg}$ is an answer set of $grnd_C(\Pi_p)$, it is also a model of $\bar{r'}$. Moreover, by Definition 75, it satisfies also all ordinary literals $B(r') \setminus B(\bar{r'})$. This is the same set as $B(r) \setminus B(\bar{r})$. Because $\mathbf{A}$ and $\mathbf{A}_{pg}$ coincide on ordinary literals, also $\mathbf{A}$ is a model of r. Thus, $\mathbf{A}$ is a model of $grnd_C(\Pi)$.

We show now that $\mathbf{A}$ is also a subset-minimal model of $fgrnd_C(\Pi)^{\mathbf{A}}$. Because we have seen that $\mathbf{A} \not\models B(r)$ for every $r \in grnd_C(\Pi)$ which has no strengthening in Π_g, it follows that $f\Pi_g{}^{\mathbf{A}}$ contains a strengthening $\bar{r}$ for every rule $r \in fgrnd_C(\Pi)^{\mathbf{A}}$. Conversely, by Definition 75 every rule in $f\Pi_g{}^{\mathbf{A}}$ is a strengthening of some rule $r \in fgrnd_C(\Pi)^{\mathbf{A}}$. Thus, the rules in $fgrnd_C(\Pi)^{\mathbf{A}}$ are even more restrictive, i.e., every model of $fgrnd_C(\Pi)^{\mathbf{A}}$ is also a model of $f\Pi_g{}^{\mathbf{A}}$. Thus, if there would be a smaller model $\mathbf{A}' \subsetneq \mathbf{A}$ of $fgrnd_C(\Pi)^{\mathbf{A}}$, it would also be a model of $f\Pi_g{}^{\mathbf{A}}$, which contradicts the assumption that $\mathbf{A}$ is an answer set of Π_g.

($\Leftarrow$) Let $\mathbf{A} \in \mathcal{AS}\big(grnd_C(\Pi)\big)$, then it is also a model of Π_g because this program is (possibly) less restrictive. It remains to show that $\mathbf{A}$ is also a subset-minimal model of $f\Pi_g{}^{\mathbf{A}}$. By Lemma B.7, $\mathbf{A}$ can be extended to an answer set $\mathbf{A}_p$ of $grnd_C(\Pi_p)$.

Let for every $r \in grnd_C(\Pi)$ be r' the respective rule in $grnd_C(\Pi_p)$ with replacement atoms instead of external atoms. Note that the rules in $f\Pi_g{}^{\mathbf{A}}$ are strengthenings of the rules in $fgrnd_C(\Pi)^{\mathbf{A}}$. Let $r \in grnd_C(\Pi)$. (1) If there is no strengthening of r in Π_g, then also r' has no strengthening in Π_{pg}. But this means, that every answer set of $grnd_C(\Pi_p)$ falsifies an ordinary body literal in r', thus also $\mathbf{A}_p$. Because $\mathbf{A}$ and $\mathbf{A}_p$ coincide on ordinary literals, also $\mathbf{A}$ falsifies some ordinary literal in $B(r)$, thus r is not in $fgrnd_C(\Pi)^{\mathbf{A}}$. (2) Now suppose there is a strengthening $\bar{r}$ of r in Π_g. Then $\mathbf{A} \models B(r)$ implies $\mathbf{A} \models B(\bar{r})$. Conversely, if $\mathbf{A} \models B(\bar{r})$, then the missing literals in $B(r) \setminus B(\bar{r})$ are satisfied as well because they are satisfied by all answer sets of $grnd_C(\Pi_p)$, including $\mathbf{A}_p$, which coincides with $\mathbf{A}$ on ordinary atoms (otherwise the literal would not have been removed by the optimizer).

Now suppose $f\Pi_g{}^{\mathbf{A}}$ has a smaller model $\mathbf{A}' \subsetneq \mathbf{A}$. Then $\mathbf{A}'$ is a model of $fgrnd_C(\Pi)^{\mathbf{A}}$ because we have seen that the missing literals are satisfied as well. □

Now we can now prove the following result.

Theorem 6 (Correctness of Algorithm GroundHEX)**.** *If* Π *is a liberally de-safe* HEX*-program, then GroundHEX*$(\Pi) \equiv^{pos} \Pi$.

Proof. Let Π_g = GuessAndCheckHexEvaluation(Π) and let C be the constants which appear in Π_g.

The differences to Algorithm GroundHEXNaive are that we (i) use a faithful (optimized) ASP grounding procedure to compute Π_g instead of the naive $grnd_C(\Pi_p)$; (ii) consider only de-safety relevant external atoms in Part (b); and (iii) a different set of interpretations in Part (c). We will now show that the algorithm is still correct.

We first ignore modification (iii) and show that the algorithm is still correct if only modifications (i) and (ii) are active.

We need to show that $\mathcal{AS}(\Pi_g) = \mathcal{AS}(\Pi)$. Recall that $\mathcal{AS}(\Pi_g) = \mathcal{AS}\big(grnd_C(\Pi)\big)$ by Lemma B.8, thus it suffices to show $\mathcal{AS}\big(grnd_C(\Pi)\big) = \mathcal{AS}\big(grnd_{C'}(\Pi)\big)$ for any $C' \supseteq C$.

($\Rightarrow$) Let $\mathbf{A} \in \mathcal{AS}\big(grnd_C(\Pi)\big)$. By Lemma B.5, $\mathbf{A}$ is a model of $grnd_{C'}(\Pi)$. It remains to show that it is also a subset-minimal model of $fgrnd_{C'}(\Pi)^{\mathbf{A}}$. As $C \subseteq C'$, $fgrnd_C(\Pi)^{\mathbf{A}} \subseteq fgrnd_{C'}(\Pi)^{\mathbf{A}}$. Moreover, by Lemma B.5 $\mathbf{A} \not\models B(r)$ for any $r \in grnd_{C'}(\Pi) \setminus grnd_C(\Pi)$, thus $fgrnd_C(\Pi)^{\mathbf{A}} = fgrnd_{C'}(\Pi)^{\mathbf{A}}$. But since $\mathbf{A} \in \mathcal{AS}\big(grnd_C(\Pi)\big)$, it is a subset-minimal model of $fgrnd_C(\Pi)^{\mathbf{A}}$, thus also of $fgrnd_{C'}(\Pi)^{\mathbf{A}}$, i.e., $\mathbf{A} \in \mathcal{AS}\big(grnd_{C'}(\Pi)\big)$.

($\Leftarrow$) Let $\mathbf{A}' \in \mathcal{AS}\big(grnd_{C'}(\Pi)\big)$. We show that $\mathbf{A} = \mathbf{A}' \cap A\big(grnd_C(\Pi)\big)$ is an answer set of $grnd_C(\Pi)$.

Because $grnd_C(\Pi) \subseteq grnd_{C'}(\Pi)$, it is trivial that $\mathbf{A}$ is a model of $grnd_C(\Pi)$. It remains to show that it is also a subset-minimal model of $fgrnd_C(\Pi)^{\mathbf{A}}$.

By Lemma B.5, $\mathbf{A}$ is also a model of $grnd_{C'}(\Pi)$. But then $\mathbf{A} = \mathbf{A}'$ because $\mathbf{A} \subsetneq \mathbf{A}'$ would imply that $\mathbf{A}'$ is not subset-minimal, which contradicts the assumption that it is an answer set of $grnd_{C'}(\Pi)$, thus $\mathbf{A} = \mathbf{A}'$. Because $grnd_C(\Pi) \subseteq grnd_{C'}(\Pi)$ and $\mathbf{A} \not\models B(r)$ for all $r \in grnd_{C'}(\Pi) \setminus grnd_C(\Pi)$ by Lemma B.5, we have $fgrnd_C(\Pi)^{\mathbf{A}} = fgrnd_{C'}(\Pi)^{\mathbf{A}}$. Because $\mathbf{A}$ is a subset-minimal model of $grnd_{C'}(\Pi)^{\mathbf{A}}$, it is a subset-minimal model of $fgrnd_C(\Pi)^{\mathbf{A}}$. Thus, $\mathbf{A}$ is an answer set of $grnd_C(\Pi)$.

Finally, consider modification (iii). While Algorithm GroundHEXNaive loops for all models of Π_{pg}, the optimized algorithm constructs the considered assignments such that the output of the external atoms is maximized: all monotonic input atoms are set to true, all antimonotonic input atoms to false, and for nonmonotonic input atoms all combinations are checked (except facts, which are always true). Every model of Π_{pg} considered by Algorithm GroundHEX, is contained in some assignment enumerated by Algorithm GroundHEXNaive. The output of the external atom wrt. this assignment may be larger, but never smaller. Thus, the optimized algorithm only produces larger but never smaller groundings wrt. the set of constants. As we have shown in Lemma B.5, this guarantees that the program has the same answer sets.

We also need to show that the algorithm terminates. But this follows from the observation that each run of the loop at (c) corresponds to a (restricted) application of operator G_Π; while G_Π instantiates rules whenever their positive body is satisfied by some of the enumerated assignments, our algorithm also respects the negative part of the rule body, i.e., it is even more restrictive. But by Corollary 4.2, $G_\Pi^\infty(\emptyset)$ is finite for liberally de-safe programs, thus the grounding produced by our algorithm is finite as well. Therefore the algorithm terminates. □

B.3 Query Answering over Positive HEX$^\exists$-Programs (cf. Section 6.1.3)

Towards a proof of Proposition 6.3, we first introduce some lemmas.

Lemma B.9. *If Π is a shy Datalog$^\exists$-program and $\Pi_g = \mathit{GroundDESafeHEX}\big(T_\exists(\Pi)\big)$, then the unique answer set of Π_g is a universal model of Π.*

Algorithm Chase

Input: A $Datalog^{\exists}$-program Π
Output: A universal model Chase(Π) for Π

(a) $C \leftarrow \{a \mid a \leftarrow . \in \Pi\}$
(b) $NewAtoms \leftarrow \emptyset$
for $r \in \Pi$ **do**
  Let **X** be the universal variables and let **Y** be the existential variables in r
  for *each firing substitution* σ *for* r *wrt.* C **do**
(c) **if** $C \cup NewAtoms \not\models \sigma\big(H(r)\big)$ **then**
    Let $\hat{\sigma}$ extend σ on $\mathbf{Y} \cup \mathbf{X}$ s.t. it associates each variable in **Y** a different null
    Add $\hat{\sigma}\big(H(r)\big)$ to $NewAtoms$
if $NewAtoms \neq \emptyset$ **then**
  $C \leftarrow C \cup NewAtoms$
  Goto (b)
return C

Proof. In this proof we refer to the Algorithm Chase as used by Leone et al. (2012)[1].

When we write *n-th iteration*, we mean the the n-th iteration of the outermost loop of algorithm GroundDESafeHEX. Let $\mathbf{A}$ be the unique answer set of Π_g and let $U = \text{Chase}(\Pi)$ be the universal model computed by the Chase procedure. We first show that there exists a homomorphism from $\mathbf{A}$ to U. Then we prove that $\mathbf{A}$ is a model of Π, which concludes the proof that it is a universal model as well.

For a rule $r \in \Pi$, let $r_{T_\exists} \in T_\exists(\Pi)$ be the corresponding rule in $T_\exists(\Pi)$. We stepwise construct an isomorphism h, beginning with the empty one, and show by induction that the following holds for this isomorphism. Whenever Chase adds an atom $a = \hat{\sigma}\big(H(r)\big)$ to C, then our algorithm adds an o-strengthening $r'_{T_\exists}$ of an instance of $r_{T_\exists}$ to the grounding Π_g s.t. $a = h\big(H(r'_{T_\exists})\big)$ (and $h^{-1}(a) = H(r'_{T_\exists})$) and such that for the unique answer set $\mathbf{A}$ of Π_g it holds that $\mathbf{A} \models B(r'_{T_\exists})$ (and thus, as it is an answer set, also $\mathbf{A} \models H(r'_{T_\exists})$).

Suppose some atom $p(\mathbf{u}, \mathbf{e})$ is added to C by Chase in the n-th iteration, where **u** are the substitutions for universally quantified variables and **e** the substitutions for existentially quantified ones.

For the base case $n = 0$, all (possibly existentially quantified) facts in Π are added to C. Facts without existential variables are trivially also part of the grounding Π_g, thus the proposition holds for any isomorphism, thus also for h. Facts with existential variables do not contain universal variables (otherwise they would not be safe), i.e., they are of form $r = \exists \mathbf{X} : p(\mathbf{X})$. Then Chase adds $p(\mathbf{e})$ for some vector of fresh nulls **e** to C. Thus $r_{T_\exists} = p(\mathbf{X}) \leftarrow \&exists^{|\mathbf{X}|}[](\mathbf{X})$. Since the external atom has no input parameters, the input auxiliary rule degenerates to fact $r^{\&exists^{|\mathbf{X}|}[r](\mathbf{X})}_{inp}()$ and the algorithm introduces at (g) a rule $e_{r,\&g[]}(\mathbf{x}) \vee ne_{r,\&g[]}(\mathbf{x}) \leftarrow$ for a

[1] The parsimonious chase algorithm pChase corresponds to Algorithm Chase if the check at (c) is changed from '$C \cup NewAtoms \not\models \sigma\big(H(r)\big)$' to '$\sigma\big(H(r)\big)$ is not homomorphic to an atom in $C \cup NewAtoms$'.

fresh vector of nulls $\mathbf{x}$. Since $\mathbf{x}$ is new in Π_p and $\mathbf{e}$ is new in C, we can easily extend h s.t. $h\big(p(\mathbf{x})\big) = p(\mathbf{e})$ and $h^{-1}\big(p(\mathbf{e})\big) = p(\mathbf{x})$.

For the induction step $n \mapsto n+1$, suppose there is a firing substitution σ for some r wrt. C and suppose $C \cup \mathit{NewAtoms} \not\models \sigma\big(H(r)\big)$, i.e., Chase adds $\hat{\sigma}\big(H(r)\big)$ to $\mathit{NewAtoms}$ and thus to C. Because σ is a firing substitution, $\sigma\big(B(r)\big) \subseteq C$, i.e., all body atoms of r under σ have been added to C in some earlier iteration. Thus, by the induction hypothesis, our algorithm adds for all $b \in \sigma\big(B(r)\big)$ a rule r_b with $h\big(H(r_b)\big) = b$ and $H(r_b) = h^{-1}(b)$ to Π_g, s.t. $B(r_b)$ and $H(r_b)$ are satisfied by the unique answer set of Π_g, i.e., r_b is the rule which derives atom $h^{-1}(b)$. All ordinary atoms in $B(r_{T_\exists})$ occur also in $B(r)$, thus $\mathbf{A}$ satisfies the ordinary atoms in $B(r_{T_\exists})$ under substitution $h^{-1} \circ \sigma$. For existentially quantified variables, $r_{T_\exists}$ contains an additional atom $\&\mathit{exists}[\mathbf{Y}](\mathbf{X})$ in the rule body and we show now that this atom is satisfied by $\mathbf{A}$ as well, for an appropriate vector $\mathbf{c}$ in place of $\mathbf{Y}$ and some fresh vector of nulls $\mathbf{x}$ in place of $\mathbf{X}$. Because all ordinary atoms in $B(r_{T_\exists})$ under substitution $h^{-1} \circ \sigma$ are satisfied by $\mathbf{A}$, our algorithm will add an o-strengthening of the input auxiliary rule of $\&\mathit{exists}[r, \mathbf{Y}](\mathbf{X})$ and the atom $r_{inp}^{\&\mathit{exists}[r,\mathbf{Y}](\mathbf{X})}(\mathbf{c})$ will appear in $A(\Pi_{pg})$, where $\mathbf{c} = h^{-1} \circ \sigma(\mathbf{u})$ is the vector of substitutions for the universally quantified variables in r as defined by $h^{-1} \circ \sigma$. But then by definition, $\&\mathit{exists}[r, \mathbf{c}](\mathbf{x})$ holds for a vector of nulls $\mathbf{x}$, which is unique for $\mathbf{c}$ and r and new in Π_p. The algorithm will add a guessing rule for $\&\mathit{exists}[\mathbf{c}](\mathbf{x})$ to Π_p and subsequently the desired instance $r'_{T_\exists}$ with $H(r'_{T_\exists}) = p(\mathbf{c}, \mathbf{x})$ will appear in Π_g. Because $\mathbf{x}$ is new in Π_p and $\mathbf{e}$ is new in C, h does not define any mapping for them so far, thus we can easily extend h s.t. $h^{-1}\big(p(\mathbf{u}, \mathbf{e})\big) = p(\mathbf{c}, \mathbf{x})$ and $p(\mathbf{u}, \mathbf{e}) = h\big(p(\mathbf{c}, \mathbf{x})\big)$. This concludes the induction step.

This shows that for any atom $a = \hat{\sigma}\big(H(r)\big)$ added to C in some interation $n \geq 0$, our algorithm adds an according rule to Π_g. Since U is just the fixpoint of this iteration, this holds also for the universal model U. The algorithm may add further rules to Π_g. However, their bodies remain unsatisfied by $\mathbf{A}$ because otherwise Chase$(\Pi, 0)$ would have identified firing substitutions and added the respective rule heads. Thus, the additional rules do not harm the procedure.

Moreover, the definition of $\&\mathit{exists}$ correctly models the semantics of the existential quantifier. Thus, $\mathbf{A}$ is also a model of Π. Since we have defined a homomorphism from $\mathbf{A}$ to U, we have shown that $\mathbf{A}$ is also a universal model of Π. □

Lemma B.10. *If Π is a shy $Datalog^\exists$-program, then* GroundDatalog$^\exists(\Pi, \infty)$ *yields the same (possibly infinite) program Π_g as* GroundDESafeHEX$\big(T_\exists(\Pi)\big)$.

Proof. Resetting PIT to $\emptyset$ at the beginning of each iteration of the loop at (b) has the same effect as disabling the homomorphism check. □

Lemma B.11. *Let Π by a Shy-program and let q be conjunctive query with n different existentially quantified variables. Then $ans(q, \Pi) \subseteq ans\big(q, pChase(\Pi, n+1)\big)$.*

Proof. The lemma is equivalent to Lemma 4.10 by Leone et al. (2012). □

Lemma B.12. *If Π is a shy $Datalog^\exists$-program and $\Pi_g =$* GroundDatalog$^\exists(\Pi, k)$*, then the unique answer set of Π_g is complete for conjunctive query answering of queries with up to k existentially quantified variables.*

Proof. Resetting PIT to $\emptyset$ after every iteration of the main loop at (b) behaves like freezing of nulls as used by Leone et al. (2012) and the loop at (b) runs $k+1$ times. Thus, the claim follows from Lemma B.11. □

Lemma B.13. *Algorithm* GroundDatalog$^{\exists}(\Pi, k)$ *terminates.*

Proof. Since all newly introduced values are null values, the loop at (f) introduces only finitely many new values because all remaining vectors $\mathbf{y}$ will eventually become homomorphic to some previously processed input vector. Thus each iteration of the loop at (c) terminates. As k is finite, also the loop at (b) terminates. □

We are now ready to show Proposition 6.3.

Proposition 6.3. *For a shy program* Π*, the program produced by* GroundDatalog$^{\exists}(\Pi, k)$ *has a unique answer set which is sound and complete for answering CQs with up to* k *existential variables against* Π*.*

Proof. Soundness follows from Lemmas B.9 and B.10 (GroundDatalog$^{\exists}(\Pi, k)$ yields a subset of a universal model of Π), in combination with Proposition 6.2. Completeness follows from Lemma B.12, and termination follows from Lemma B.13. □

Curriculum Vitae

Personal Information

Date and Place of Birth
July 3, 1986 in St. Pölten, Austria

Citizenship
Austria

Family Status
Unmarried, No Children

Affiliation
Knowledge-Based Systems Group
Institute of Information Systems
Technische Universität Wien (TU Vienna)
Vienna, Austria

Academic Degrees
Dipl.-Ing. ($\widehat{=}$ M.Sc.) in *Medical Informatics*
Dipl.-Ing. ($\widehat{=}$ M.Sc.) in *Computational Intelligence*
B.Sc. in *Software and Information Engineering*

Current Position
PhD Student and Research Assistant (FWF)

Research Interests
Knowledge Representation and Reasoning; Computational Logic;
Nonmonotonic Logic Programming and Databases;
Answer Set Programming and Extensions;
Reasoner Design; Algorithms

Education

2010–ongoing
PhD Student at TU Vienna, Institute of Information Systems

Program:	*Mathematical Logic in Computer Science*
Supervisor:	O. Univ.-Prof. Dipl.-Ing. Dr. techn. Thomas Eiter
Second Supervisor:	Associate Prof. Dipl.-Ing. Dr. techn. Stefan Woltran

2009–2010
Master Student of *Medical Informatics* at TU Vienna
Graduation with Distinction as a Dipl.-Ing. ($\widehat{=}$ M.Sc.)

2008–2010
Master Student of *Computational Intelligence* at TU Vienna
Graduation with Distinction as a Dipl.-Ing. ($\widehat{=}$ M.Sc.)

2005–2008
Bachelor Student of *Software and Information Engineering* at TU Vienna
Graduation with Distinction as a B.Sc.

2000–2005
Upper Secondary School in St. Pölten, Austria
Higher Technical Institute
Department of *Electronic Data Processing and Business Organization*
Graduation with Distinction

Career History

November 2010–ongoing
Research Assistant (FWF) at TU Vienna, Institute of Information Systems

Current Project:	*Evaluation of ASP Programs with External Source Access* (FWF P24090)
Previous Project: (2010–2012)	*Reasoning in Hybrid Knowledge Bases* (FWF P20840)

March 2007–June 2010
Tutor at TU Vienna for Several Courses

Summer 2004
Internship at Cincinnati Extrusion GmbH, Vienna, Austria
Employed in the IT Department: Help-Desk Tasks and Hardware Assembling

Summer 2002
Internship at A. Porr AG, Vienna, Austria
Office Tasks and Web Development

Journal Publications

[J3] Yi-Dong Shen, Kewen Wang, Jun Deng, Christoph Redl, Thomas Krennwallner, Thomas Eiter, and Michael Fink. FLP answer set semantics without circular justifications for general logic programs. *Artificial Intelligence*, 2014. Accepted for publication.

[J2] Thomas Eiter, Michael Fink, Thomas Krennwallner, Christoph Redl, and Peter Schüller. Efficient HEX-program evaluation based on unfounded sets. *Journal of Artificial Intelligence Research*, 49:269–321, February 2014.

[J1] Thomas Eiter, Michael Fink, Thomas Krennwallner, and Christoph Redl. Conflict-driven ASP solving with external sources. *TPLP*, 12(4-5):659–679, 2012.

Conference Publications

[C11] Thomas Eiter, Michael Fink, Christoph Redl, and Daria Stepanova. Exploiting support sets for answer set programs with external evaluations. In *Proceedings of the Twenty-Eighth AAAI Conference (AAAI 2014), July 27–31, 2014, Québec City, Québec, Canada*. AAAI Press, July 2014. Accepted for publication.

[C10] Thomas Eiter, Michael Fink, Thomas Krennwallner, and Christoph Redl. HEX-programs with existential quantification. In Ricardo Rocha, editor, *Proceedings of the Twentieth International Conference on Applications of Declarative Programming and Knowledge Management (INAP 2013), Kiel, Germany, September 11-13, 2013*, September 2014. Post proceedings. Accepted for publication.

[C9] Thomas Eiter, Thomas Krennwallner, and Christoph Redl. HEX-programs with nested program calls. In Hans Tompits, editor, *Proceedings of the Nineteenth International Conference on Applications of Declarative Programming and Knowledge Management (INAP 2011)*, volume 7773 of *LNAI*, pages 1–10. Springer, October 2013.

[C8] Thomas Eiter, Michael Fink, Thomas Krennwallner, and Christoph Redl. HEX-programs with existential quantification. In Ricardo Rocha, editor, *Proceedings of the Twentieth International Conference on Applications of Declarative Programming and Knowledge Management (INAP 2013), Kiel, Germany, September 11-13, 2013*, September 2013.

[C7] Michael Fink, Stefano Germano, Giovambattista Ianni, Christoph Redl, and Peter Schüller. ActHEX: implementing HEX programs with action atoms. In Pedro Cabalar and TranCao Son, editors, *Proceedings of the Twelfth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2013)*, volume 8148 of *Lecture Notes in Computer Science*, pages 317–322. Springer Berlin Heidelberg, 2013.

[C6] Mario Alviano, Francesco Calimeri, Günther Charwat, Minh Dao-Tran, Carmine Dodaro, Giovambattista Ianni, Thomas Krennwallner, Martin Kronegger, Johannes Oetsch, Andreas Pfandler, Jörg Pührer, Christoph Redl, Francesco Ricca, Patrik Schneider, Martin Schwengerer, Lara Katharina Spendier, Johannes Peter Wallner, and Guohui Xiao. The

fourth answer set programming competition: Preliminary report. In Pedro Cabalar and Tran Cao Son, editors, *Proceedings of the Twelfth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2013), Corunna, Spain, September 15-19, 2013*, volume 8148 of *LNCS*, pages 42–53. Springer, September 2013.

[C5] Günther Charwat, Giovambattista Ianni, Thomas Krennwallner, Martin Kronegger, Andreas Pfandler, Christoph Redl, Martin Schwengerer, Lara Spendier, Johannes Peter Wallner, and Guohui Xiao. VCWC: a versioning competition workflow compiler. In Pedro Cabalar and Tran Cao Son, editors, *Proceedings of the Twelfth ernational Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2013), Corunna, Spain, September 15-19, 2013*, volume 8148 of *LNCS*, pages 233–238. Springer, September 2013.

[C4] Thomas Eiter, Michael Fink, Thomas Krennwallner, and Christoph Redl. Liberal safety for answer set programs with external sources. In Marie desJardins and Michael Littman, editors, *Proceedings of the Twenty-Seventh AAAI Conference (AAAI 2013), July 14–18, 2013, Bellevue, Washington, USA*, pages 267–275. AAAI Press, July 2013.

[C3] Thomas Eiter, Michael Fink, Thomas Krennwallner, Christoph Redl, and Peter Schüller. Exploiting unfounded sets for HEX-program evaluation. In *Proceedings of the Thirteenth European Conference on Logics in Artificial Intelligence (JELIA 2012), Toulouse, France, September 26-28, 2012*, September 2012.

[C2] Thomas Eiter, Thomas Krennwallner, and Christoph Redl. Nested HEX-programs. In Hans Tompits, editor, *Proceedings of the Nineteenth International Conference on Applications of Declarative Programming and Knowledge Management (INAP 2011), Vienna, Austria, September 28–30, 2011*, number arXiv:1108.5626v1 in arXiv. Computing Research Repository (CoRR), September 2011.

[C1] Christoph Redl, Thomas Eiter, and Thomas Krennwallner. Declarative belief set merging using merging plans. In Ricardo Rocha and John Launchbury, editors, *Proceedings of the Thirteenth International Symposium on Practical Aspects of Declarative Languages (PADL 2011), Austin, Texas, USA, January 24-25, 2011*, volume 6539 of *LNCS*, pages 99–114. Springer, January 2011.

Workshop Publications

[W4] Francesco Calimeri, Michael Fink, Stefano Germano, Giovambattista Ianni, Christoph Redl, and Anton Wimmer. AngryHEX: an artificial player for angry birds based on declarative knowledge bases. In Matteo Baldoni, Federico Chesani, Paola Mello, and Marco Montali, editors, *National Workshop and Prize on Popularize Artificial Intelligence, Turin, Italy, December 5, 2013*, pages 29–35, December 2013.

[W3] Thomas Eiter, Michael Fink, Thomas Krennwallner, and Christoph Redl. Grounding HEX-programs with expanding domains. In David Pearce, Shahab Tasharrofi, Evgenia Ternovska, and Concepción Vidal, editors, *Second Workshop on Grounding and Transforma-*

tions for Theories with Variables (GTTV 2013), Corunna, Spain, September 15, 2013, pages 3–15, September 2013.

[W2] Thomas Eiter, Michael Fink, Thomas Krennwallner, Christoph Redl, and Peter Schüller. Eliminating unfounded set checking for HEX-programs. In Michael Fink and Yuliya Lierler, editors, *Fifth Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP 2012), September 4, 2012, Budapest, Hungary*, pages 83–97, September 2012.

[W1] Thomas Eiter, Thomas Krennwallner, and Christoph Redl. Declarative merging of and reasoning about decision diagrams. In Alessandro Dal Palù, Agostino Dovier, and Andrea Formisano, editors, *Workshop on Constraint Based Methods for Bioinformatics (WCB 2011), Perugia, Italy, September 12, 2011*, pages 3–15. Dipartimento di Matematica e Informatica, Universita degli Studi di Perugia, September 2011.

Doctoral Consortia

[D1] Christoph Redl. Answer set programming with external sources. In *Eighth ICLP Doctoral Consortium, Budapest, Hungary, September 4, 2012*, pages 469–475, 2012.

Technical Reports

[R1] Thomas Eiter, Michael Fink, Thomas Krennwallner, Christoph Redl, and Peter Schüller. Improving HEX-program evaluation based on unfounded sets. Technical Report INFSYS RR-1843-12-08, Institut für Informationssysteme, Technische Universität Wien, A-1040 Vienna, Austria, September 2013.

Poster Presentations

[P1] Francesco Calimeri, Michael Fink, Stefano Germano, Giovambattista Ianni, Christoph Redl, and Anton Wimmer. AngryHEX: an angry birds-playing agent based on HEX-programs. Angry-Birds Competition 2013, August 6-9, 2013, Beijing, China, August 2013.

Theses

[T2] Christoph Redl. Merging of biomedical decision diagrams. Master's thesis, Vienna University of Technology, Knowledge-Based Systems Group, A-1040 Vienna, Karlsplatz 13, October 2010.

[T1] Christoph Redl. Development of a belief merging framewerk for dlvhex. Master's thesis, Vienna University of Technology, Knowledge-based Systems Group, A-1040 Vienna, Karlsplatz 13, July 2010.

Scientific Talks

[S9] Thomas Eiter, Michael Fink, Thomas Krennwallner, and Christoph Redl. HEX-programs with existential quantification. In Ricardo Rocha, editor, *Twentieth International Conference on Applications of Declarative Programming and Knowledge Management (INAP 2013), Kiel, Germany, September 11-13, 2013*, September 2013.

[S8] Thomas Eiter, Michael Fink, Thomas Krennwallner, and Christoph Redl. Liberal safety criteria for HEX-programs. In Marie desJardins and Michael Littman, editors, *Twenty-Seventh AAAI Conference (AAAI 2013), July 14–18, 2013, Bellevue, Washington, USA*, pages 267–275. AAAI Press, July 2013.

[S7] Thomas Eiter, Michael Fink, Thomas Krennwallner, Christoph Redl, and Peter Schüller. Exploiting unfounded sets for HEX-program evaluation. In *Thirteenth European Conference on Logics in Artificial Intelligence, Toulouse, France, September 26-28, 2012*, September 2012.

[S6] Thomas Eiter, Michael Fink, Thomas Krennwallner, and Christoph Redl. Conflict-driven ASP solving with external sources. In *Eighth International Conference on Logic Programming (ICLP 2012), Budapest, Hungary, September 4–8, 2012*, pages 659–679, 2012.

[S5] Christoph Redl. Answer set programming with external sources. In *Eighth ICLP Doctoral Consortium, Budapest, Hungary, September 4, 2012*, pages 469–475, 2012.

[S4] Thomas Eiter, Michael Fink, Thomas Krennwallner, Christoph Redl, and Peter Schüller. Evaluation of ASP programs with external source access. In *University of Potsdam, Potsdam, Germany, February 1, 2012*.

[S3] Thomas Eiter, Thomas Krennwallner, and Christoph Redl. Nested HEX-programs. In Hans Tompits, editor, *Nineteenth International Conference on Applications of Declarative Programming and Knowledge Management (INAP 2011), Vienna, Austria, September 28–30, 2011*, number arXiv:1108.5626v1 in arXiv. Computing Research Repository (CoRR), September 2011.

[S2] Thomas Eiter, Thomas Krennwallner, and Christoph Redl. Declarative merging of and reasoning about decision diagrams. In Alessandro Dal Palù, Agostino Dovier, and Andrea Formisano, editors, *Workshop on Constraint Based Methods for Bioinformatics (WCB 2011), Perugia, Italy, September 12, 2011*, pages 3–15. Dipartimento di Matematica e Informatica, Universita degli Studi di Perugia, September 2011.

[S1] Christoph Redl, Thomas Eiter, and Thomas Krennwallner. Declarative belief set merging using merging plans. In Ricardo Rocha and John Launchbury, editors, *Thirteenth International Symposium on Practical ASPects of Declarative Languages (PADL 2011), Austin, Texas, USA, January 24-25, 2011*, volume 6539 of *LNCS*, pages 99–114. Springer, January 2011.