FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Integrating Constraint Programming into Answer Set Programming

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Master of Science (M.Sc.)

im Rahmen des Erasmus-Mundus-Studiums

Computational Logic

eingereicht von

Oleksandr Stashuk
Matrikelnummer 1228071

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: o.Univ.-Prof. Dipl.-Ing. Dr. techn. Thomas Eiter
Mitwirkung: Dipl.-Ing. Thomas Krennwallner
Dipl.-Ing. Christoph Redl

Wien, 28.09.2013

(Unterschrift Verfasserin) (Unterschrift Betreuung)

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

FAKULTÄT
FÜR !NFORMATIK

Faculty of Informatics

Integrating Constraint Programming into Answer Set Programming

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Master of Science (M.Sc.)

in

Computational Logic

by

Oleksandr Stashuk
Registration Number 1228071

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: o.Univ.-Prof. Dipl.-Ing. Dr. techn. Thomas Eiter

Vienna, 28.09.2013

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Oleksandr Stashuk
Lienfeldergasse 60C, 16-17, 1160 Wien

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.

(Ort, Datum)	(Unterschrift Verfasserin)

Acknowledgements

All of the people I have met these two years have added a small value to this thesis. These words are expressing my gratitude and respect for them and everything they have done for me.

I would like to take a chance to say thank you to my thesis supervisors, Thomas Eiter, Thomas Krennwallner and Christoph Redl, for the overall support and feedback while writing the thesis. Their valuable comments and help were very important and drove me to the ultimate goal. They have opened a big area of scientific research and this have already influenced my life.

I am thankful to my classmates and friends that appeared in these years and the friends that remained. You will be always welcomed in my heart. I explored many countries but still I have a place that I can call home. Finally, I would thank my family and especially Maria who was always close and supported me.

Abstract

Many declarative problems exploit hard computational nature that need to be evaluated effectively. Examples of such problems are task scheduling, optimization problems, database transaction management. Successful approaches appear when several programming paradigms are combined and state-of-the-art techniques are reused in different combinations. This thesis presents a hybrid constraint answer-set programming solver that provides a modeling language and solving capabilities of Answer-Set Programming (ASP) and special relations over variables called constraints from area of Constraint Programming (CP). This combination can be defined as Constraint Answer-Set Programming (CASP).

The proposed framework provides a custom syntax for supporting constraint and answer-set constructions and in addition it provides a support for a special type of global constraints. We then show how such programs can be transformed to HEX programs with external atoms and evaluated. Several formal properties of such transformation are provided.

The thesis provides several optimizations for the original approach. The first improvement relies on reducing the size of the transformed program. This is based on the observation of the redundancy in some of the translated rules. This translation reduction does not affect answer sets of the programs. Another optimization is related to the program evaluation. The key idea is to find the small irreducible inconsistent set of constraints and add it as a nogood to the program. This way we will introduce user defined learning for the program. We also show that the learning functions for HEX programs based on these algorithms are correct.

This thesis additionally provides the implementation of such approach as a plugin for DLVHEX system. We show the plugin architecture, ideas of the implementation and describe required algorithms. We describe how main components, namely CASP converter, rewriter, external atom evaluation and learning can be implemented.

Finally, we provide examples of how the CASP plugin can be used. We study particular instance of scheduling optimization and geometry problem. Additionally we provide comprehensive benchmarking for such problems using different configurations and improvements from the thesis. We also show an example of how the plugin can be used in combination with other source of knowledge in the DLVHEX system.

Kurzfassung

Viele Probleme sind von komplexer rechnerischer Natur und sollen auf effektive Art gelöst werden. Beispiele für solche Probleme sind Aufgabenplanung, Optimierungsaufgaben, oder Verwaltung von Datenbanktransaktionen. Erfolgreiche Lösungen entstehen dann, wenn sich mehrere Programmierparadigma vereinigen und state-of-the-art Techniken in diversen Kombinationen verwendet werden. Diese Diplomarbeit beschreibt Hybrid Constraint Answer-Set Programmierung, die die Modellierungssprache und Entscheidungsverfahren der Answer-Set Programmierung (ASP) sowie besondere Einschränkungen für Variablen anbietet, die als Constraints aus dem Gebiet der Constraint Programmierung (CP) bezeichnet werden. Diese Kombination wird als Constraint Answer-Set Programming (CASP) bezeichnet.

Das vorgeschlagene Framework bietet eine spezielle Syntax an, die Constraints und Answer-Set Programmierung unterstützt. Außerdem ermöglicht es, spezielle Constraints zu verwenden, die als Globalconstraints bezeichnet werden. Ferner werden wir näher beschreiben, wie man solche Programme in HEX Programme mit externen Atomen umwandelt und berechnet. Weiters werden formale Eigenschaften solcher Transformationen behandelt.

Diese Diplomarbeit beschreibt mehrere Optimierungsmöglichkeiten dieses Ansatzes. Die erste Verbesserung besteht in der Reduzierung der Größe des transformierten Programmes. Dies beruht auf der Beobachtung der Redundanz in einigen der transformierten Regeln. Diese Transformationsreduzierung beeinflusst nicht die Answer Sets der Programme. Eine weitere Optimierung betrifft die Programmberechnung. Die Hauptidee besteht darin, eine kleine unreduzierbar widersprüchliche Menge von Constraints zu finden, und diese als nogoods dem Programm hinzuzufügen. Wir nutzen hier also das benutzerdefinierbare Lernen für das Programm. Wir demonstrieren auch, dass die Lernfunktionen für HEX Programme, die auf solchen Algorithmen beruhen, korrekt sind.

Die Diplomarbeit zeigt ferner die Umsetzung dieses Ansatzes als Plugin für das DLVHEX System. Wir demonstrieren den Aufbau des Plugins sowie Ideen für die Umsetzung, und beschreiben die notwendigen Algorithmen. Wir erläutern, wie Hauptkomponenten, die als CASP Converter, Rewriter, Berechnung von externen Atomen, und Lernen bezeichnet werden, umgesetzt werden können. Zusätzlich führen wir Beispiele an, wie das CASP Plugin verwendet wird. Wir untersuchen bestimmte Instanzen der Optimierungsplanung und geometrische Aufgaben. Darüber hinaus bieten wir ein umfassendes Benchmarking für solche Probleme an, indem wir verschiedene Konfigurationen und Optimierungen der Diplomarbeit nutzen. Schließlich führen wir ein Beispiel an, wie der Plugin in Verbindung mit anderen Wissensquellen im System DLVHEX verwendet werden kann.

Contents

CHAPTER 1

Introduction

We will start the thesis with the motivation and problem definition for constraint answer-set programming. The typical problems and usage are also described in this section. We then provide the description of the thesis contribution to the overall research field and show the current research work in the related areas. Finally, this section provides an overview of how the thesis is structured.

1.1 Task description

Answer-set programming (ASP) is a widely used technique for knowledge representation and non-monotonic reasoning. As such, there exists a wide range of problems which are solved with ASP. The basic idea of ASP is to specify the problem by a non-monotonic logic program. The benefit over traditional, imperative solutions to the problems is that the problem is described *directly* with a declarative specification. Once it is specified, it can be automatically solved and there is not requirement to write a particular solver itself for the problem. Having this in mind, we can now shift attention from problem solving itself to the problem stating, which is an easier and less error-prone task in general.

There are many problems in answer-set programming that are connected with other programming paradigms: description logic reasoning, reasoning over Semantic Web and others. One of such natural integrations is constraint programming. This paradigm deals with such search problems, where a set of variables is required to satisfy restrictions that are named constraints. Traditional constraint satisfaction problems usually aim only at assignment of variables to the domain values. The representation in form of constraints is often much simpler and less error-prone while a general purpose constraint solvers provide reasonable performance. This field is under very active development and new, fast techniques are emerging. An overview of such techniques and solvers in general can be found in the results of the Constraint Solver Competition [46].

Having these two approaches defined, it is natural to provide a combination of them that would provide flexible and fast framework for solving problems. Historically, it started with constraint logic programming [20], which later developed into constraint answer-set programming. Constraint answer-set programming has a huge variety of application; an overview can be found in [21]. The most studied case is the usage of this paradigm for solving scheduling and assigning tasks, where an activity of an object is required to be spread out based on the constraints. There are many specific variations of the tasks, that be can solved by constraint answer-set programming [11], [12], [43].

Another usage of the paradigm is the application to database theory. Constraints are used to schedule the transactions performed in a database [37]. Constraint logic programming can also be applied to represent probabilistic databases [36].

A wide variety of problems is provided in the field of biology. These problems can be effectively solved by constraint answer-set programming. Protein structure prediction, one formalization of which can be seen as optimization problem that can be solved by logic programming with constraints [16]. Large area of research is the inconsistency detection and management in biological networks [25]. Most of the time the advantage of constraints in combination with logic programming is easy description of problem specification and quick program prototyping. Many of the problems in biology posses limitations on the structure of the objects, that can be easily encoded into constraint answer-set programs.

Constraint answer-set programming is also used in software verification. The main application of the paradigm is the specification of integer constraints over time in the answer-set programs. One particular problem is protocol verification [7], [15].

We will also give the motivation why constraint answer-set programming performs better than traditional approaches. Consider an example of packing problem, which is part of ASP competition 2011. [1] This problem requires set of rectangles to be placed on a plane such that no two of them intersect. The usual ASP approach would create a constraint based on intersection of two rectangles. This leads to iterating all possible intersections one after the other which is not very effective. More intelligent approach would be to give input data to an external constraint solver which would internally propagate constraints in a more sophisticated manner rather then checking all the possible pairs of the rectangle intersection.

Another important aspect is the domain of the variables involved in the problem. Constraint solvers can propagate variables with large domains in much more efficient way than traditional ASP solvers. As for the packing problem mentioned above, this happens when there's small number of rectangles to place on plane with big dimensions.

Recent success in accelerating performance of ASP solvers with conflict-driven nogood learning is another important step towards creating efficient solvers [17]. At the level of constraint solver we can find minimal reason of inconsistency, that is sets of constraints that bring inconsistency to the given program. This drastically speeds up the program execution. Such sets could be found at level of constraint solvers as well. As such, this technique could be reused for solving constraint answer-set programs.

Having defined the problem, it is desirable to provide a hybrid solver that integrates answer-set programming and constraint programming. Such a solver should be capable of solving logic programs with constraints and find solution in a robust and efficient manner. An implementation of such solver would solve big variety of problems in an elegant and fast way.

[1] https://www.mat.unical.it/aspcomp2011/OfficialProblemSuite

1.2 Thesis contribution

The first step of the thesis is to establish a theoretical framework of integrating constraints into the ASP context. We will define specific constraint atoms and an encoding algorithm to transform them to so called HEX programs with external atoms. There will be single external atom which takes only the interpretation related to constraint satisfaction problem and evaluates whether it is possible to satisfy a given set of constraints. We will also demonstrate formal properties of this transformation.

In addition to traditional constraints we provide support of global constraints. Global constraints are powerful and effective relations that restrict domain over non-fixed number of variables. These constraints also allow to solve the constraint optimization problem. Currently sum, maximization, minimization global constraints are supported.

We will also show several formal properties of both original encoding and encoding, that supports global constraints.

We then focus on optimization of the transformation and program evaluation. We first provide a mechanism to reduce the size of the transformed program by omitting several rules. We also show that this operation can be performed without losing potential answers to the problem. The next goal is to provide theoretical background for a user-defined nogood learning. At the level of the constraint solver, we can find a minimal constraint set that is inconsistent in the current interpretation. As such, we can immediately remove the candidate answer sets that contain such minimal inconsistent sets. We will evaluate existing approaches and show new method of finding inconsistent sets, also proving their formal properties. An important requirement for the algorithm is to reduce the number of search space reset as this is very costly operation. We will also show how the algorithms can be used for nogood learning functions for HEX programs and prove their correctness.

As for the practical part of the thesis we will implement the described above approach as a plugin for the open source reasoner DLVHEX. [2] which implements HEX programs. The advantage of HEX programs is a support of bidirectional information exchange with external sources of information, which in our case is the constraint solver GECODE [3]. This solver was picked up as the state-of-the-art solution in the field of constraint programming. In addition, at the moment of development GECODE was also used in older constraint answer-set system, Clingcon [23]; this would allow fair comparison with it. We provide an implementation of the plugin and describe necessary algorithms required for it. We also show how the plugin can be installed, configured and used.

We will finally evaluate the work based on several benchmark problems and specify the justification of obtained results. The problems selected for benchmarking are variations of assignment and geometry problems. We also provide an example of how the plugin can also be used in combination with other external sources of knowledge, in particular with RDF data.

Finally, we will provide conclusions and summarize the work. We will also define existing bottlenecks of the approach and define possible extensions and future work.

[2]http://www.kr.tuwien.ac.at/research/systems/dlvhex

[3]http://gecode.org

1.3 Related work

Related research work is done in the field of constraint programming. A proper selection of fast and sophisticated constraint solver is the key to build efficient hybrid constraint answer-set solver. An overview of recent general situation in constraint solvers is provided in [8]. Some of the most successful solvers are Gecode (described in [45], [44]), MINION [27], Choco [33], Comet [31]. However, they are usually providing own specific modelling language. A recent development in providing the standard modelling language for the constraint solver is associated with MiniZinc [40]

One important aspect of the work is identification of irreducible inconsistent sets of constraints, that leads to effective learning strategies. The roots of this problem are going to the research paper of van Loon [47]. The solution was later expanded can be found by Chinneck [13] with implementation of MINOS solver for finding such sets [14]. More sophisticated approaches are described by Gleeson and Ryan [29], Greenberg and Murphy [30]. A relation between finding irreducible inconsistent sets and hitting set problem was discovered and an approach based on it was proposed [9]. An additional technique that is based on relation between irreducible inconsistent set and the complement set of maximal satisfiable subset of constraint was proposed by Bailey [3], [4].

The other research field is generally answer-set programming. There is a big variety of implementations of this programming paradigm, notable members of which are CLASP [22], CModels [28], DLV [34] and SModels [41]. The systems are competitive and the evaluation is performed regularly, the latest report can be found in [10].

A general approach of using constraints in answer-set programming can be found in [5]. There are many existing answer-set solvers that support integration with the constraint programming, like Clingcon [23], EZCSP [5], ACSolver [38], Mingo [35] and IDP [48]. They provide support of various constraint types and consequently different language for the representation of the problem. The created systems also internally use different constraint solvers; e.g.Mingo uses CPLEX [4], Clingcon uses, as was already mentioned, Gecode solver. Such issue makes comparison of the system very hard. The latest version 4.1 [5] of the Gringo grounder [24] allows constraints over integer variables, where Clasp [22] used as a solver.

1.4 Organization of thesis

The thesis is organized in the following manner. The following chapter 2 gives an introduction to the necessary basic concepts of two big areas in declarative programming: constraint programming and logic programming. Necessary definitions and notions are described there. We will approach logic programming towards answer-set semantics and later on HEX programs. Finally, we will define constraint answer-set programming, that provides features of the above-mentioned areas of declarative programming.

Chapter 3 provides a theoretical approach of encoding constraints answer-set programs to HEX programs atoms. It also defines communication protocol to the constraint solver GECODE

[4]http://www-01.ibm.com/software/commerce/optimization/cplex-cp-optimizer/

[5]http://sourceforge.net/projects/potassco/files/gringo/4.1.0/

- the translation algorithm from constraints to answer-set atoms. This chapter contains introduction for syntax and semantics of constraint plugin. We will also show that the translation created for such programs is correct. Correctness is understood in the sense of the following statement: if there is a proper constraint answer set for CASP programs, then there is an answer set for translated HEX programs and vice versa.

This chapter also contains a solution for defining special types of constraint that are called global. Global constraints allow to define relations over non-fixed number of variables, like aggregation or minimization of particular number of variables. Global constraints define very powerful tool for compacting the problem representation.

Chapter 4 provides a theoretical part for improvements of plugin. We provide an optimization of omitting guessing part for the constraint atoms, that appear in the head of the rule. Such optimization still provides a correct translation for constraint answer-set programs.

We will also provide here a general overview of user-defined learning and learning functions and specifically learning for the external atom checking constraint consistency. We will show several algorithms for this purpose and show their termination and correctness.

Next chapter 5 concerns practical part of the plugin implementation for DLVHEX. Initially, we will show how to install and use the plugin. This chapter then provides an architectural overview of the plugin, main components and details of each part. We will also explain implementation details of the plugin itself and interaction with GECODE in this chapter.

Afterwards 6 chapter provides practical case studies of the plugin. We will first focus on the problem that demonstrates improvements from the thesis. Later on, we will explore one particular problem from the ASP competition 2011, which involves constraint parts.This chapter also has an example of using the plugin with other external sources of knowledge. Finally, each of the problems contains an analysis of obtained results.

Chapter 7 summarizes the obtained results and provides conclusions. This chapter also outlines possible future work and extensions for the existing approach.

CHAPTER 2

Preliminaries

The following chapter gives an introduction to the basics of constraint programming and general overview of logic programming and HEX programs under answer-set semantics.

Both constraint and logic programming are fields of declarative programming. We will first define original concepts and ideas of constraint programming and what types of problems it solves. Later we focus on logic programming from basic approaches to modern approaches like answer-set programming.

In the end we provide an introduction to so called HEX programs which are capable of interacting with external sources of information. This feature is required to build our integration plugin.

Finally, we will provide an introduction to the field of constraint answer-set programming. It adds a special type of constraint atom in additional to ordinary atoms in answer-set programs. We want to be able to solve such type of programs with our plugin.

In addition, this section defines necessary notation used in the thesis.

2.1 Constraint Programming

Constraint programming is an approach in declarative programming and as such is focused more on what is the answer to a specific problem rather than how to compute it. The basic idea is to represent the problem by variables that are connected with each other with so-called constraints. Informally, constraints are the restrictions on the domains of the variables that the problem posses on. Constraint problem solvers are simplifying such constraints in order to find a valid assignment for the variables. Provided below is the formal definition of constraint programming. In the following chapter we will use Apt's definitions [1].

Let $V = \{V_1, ..., V_m\}$ be the set of variables. Each variable is associated with a specific domain $D_1, ..., D_m$. The domain elements can be booleans, integers, rationals, finite field elements, multisets, intervals. For now on we will focus only on the finite integer domain. This domain will be the same for every variable.

A relation c between variable subset $\{V_1, ..., V_n\}$ which is represented as a subset of Cartesian product of their domains $c \subseteq D_1 \times ... \times D_n$ is called a *constraint*.

A set $\{V_1, ..., V_n\}$ is said to be the *scope* of a constraint. We will denote $S(c)$ as scope of the constraint c.

We will also denote C as the set of such constraints.

For example, constraint can be defined as $c : x + y > 5$ over domain $D_x = D_y = [0..10]$, meaning that the sum of constraint variables x and y should be greater than 5. The scope of c is $S(c) = \{x, y\}$.

A tuple $(d_1, ..., d_n)$ is called *solution* to c iff it belongs to relation c that is $(d_1, ..., d_n) \in c$. For the constraint $c : x + y > 5$ one possible solution is $(1, 5)$.

For an assignment $A : V \to D$ and a constraint c let $A(S(c)) = (A(v_1), ..., A(v_k))$ and $sat_C(A) = \{c \in C \mid A(S(c)) \in c)\}$ that is $sat_C(A)$ represents the subset of C that are satisfied under assignment A. Constraint c is called *satisfied* under assignment A iff $c \in sat_C(A)$.

If $c = \emptyset$ it is called *inconsistent*. If for any $c \in C : c = \emptyset$, then the set C is also called inconsistent.

Definition 2.1.1. Inconsistent set of constraints $C = \{c_1, ..., c_n\}$ is called *irreducible inconsistent set* or *irreducible infeasible set* (IIS) if any subset set $\forall i : I' = I \setminus c_i$ obtained by removal of any constraint from set C is consistent.

Example 2.1.1. Consider following set of constraints.

$$\begin{aligned} C = \{c_1 : x > 1, \qquad & c_2 : x < 1, \\ c_3 : x + y < 5, \qquad & c_4 : y > 4\} \end{aligned}$$

Suppose the same domain $D_i = [0..10]$. This set is inconsistent, as there is no solution that satisfies all constraints. It is also not an irreducible inconsistent set, however the following subsets of C are possible IISs:

$$\begin{aligned} C' = \{c_1 : x > 1, \qquad & c_2 : x < 1\} \\ C'' = \{c_3 : x + y < 5, \qquad & c_4 : y > 4\} \end{aligned}$$

Definition 2.1.2. A *constraint satisfaction problem* (CSP) represented by triple $\langle V, D, C\rangle$ is the problem to find possible assignment, $V_i = d_i \in D_I, i \in \{1..n\}$, such that all constraints $c_j \in C$ are satisfied.

In order to solve a constraint satisfaction problem, each of the constraints is propagated whenever possible until no more action is applicable. In such case we obtain so called *tightened problem*.

Constraint satisfaction problem $P' = \langle V', D', C'\rangle$ is called *tightening* of the problem $P = \langle V, D, C\rangle$ iff $V' = V$, $D' \subseteq D, \forall c \in C, \exists c' \in C' : c' \subseteq c$

As such, tightening transforms CSP P to P' such that solutions to them are the same. There are many such transformations that satisfy notion of *consistency*, among them node, arc, hyper-arc, bound, range, domain, path and strong consistency. A comprehensive overview can be found in [1] as well. The state-of-the-art constraint solvers use various algorithms to maintain such consistencies and find solutions to constraint satisfaction problems or declare that the problem is inconsistent.

Example 2.1.2. Consider the following CSP:

$$
\begin{aligned}
&V = \{x, y\} \\
&C = \{c_1 : 3 * x + y \leq 4, && D = \{x \in [0..10], \\
&\quad c_2 : y \neq 0, && \quad y \in [0..10]\} \\
&\quad c_3 : y \leq 3, \\
&\quad c_4 : y < 7\}
\end{aligned}
$$

First, we can observe that the constraint c_3 is stronger than c_4 and we can simply eliminate the latter one. Because of the constraints c_2 and c_3 the domain of y can be restricted to $y \in [1..3]$. Looking at the constraint c_1 we observe that the only possible values for x, such that expression $3 * x + y$ is less or equals to 4, are $x \in [0..1]$. As such, we obtain another CSP P':

$$
\begin{aligned}
&V' = \{x, y\} \\
&C' = \{c_1' : 3 * x + y \leq 4, && D' = \{x \in [0..1], \\
&\quad c_2' : y \neq 0, && \quad y \in [1..3]\} \\
&\quad c_3' : y \leq 3\}
\end{aligned}
$$

The problem P' is a tightening of problem P as $V = V'$, $D' \subseteq D$, and the constraint $c_4 \in C$ has a respective stronger constraint $c_4 \subseteq c_3', c_3' \in C'$.

A variant of constraint satisfaction problem is a *constraint optimization problem* (COP). It is represented by a tuple $\langle V, D, C, \gamma \rangle$, where $\gamma : C \to Z$ is a *cost function* whose value is required to be maximized or minimized. In such case, we are not only interested in solutions that satisfy the set of constraints, but also order them based on the cost function and pick up best of them.

This concludes necessary introduction of a constraint programming. In what is following we will present basics of a logic programming.

2.2 Logic Programming

Logic programming is another field of declarative programming where the focus is again put on what is computed rather than the actual procedure of computation. Throughout the development of the field many semantics were established. Logic programming is highly connected with an artificial intelligence, where problem nature is established in the form of general rules [42]. Initial approaches started with resolution method, yet they had many limitations. An example of the problem is that having self recursing rules could lead to non terminating programs. We will start first with least fix-point semantics and consequently move on to answer-set semantics. We will finally introduce HEX programs and explain the motivation for selecting them as necessary formalism to implements the plugin.

The following section presents necessary concepts, definitions and further used notions in the thesis.

Least Fixed Point Semantics

In what is below we are going to present an overview of the least fix-point semantics. We will start with basic concepts of logic programs.

A *first-order signature* $\Sigma = \langle \Sigma_v, \Sigma_p, \Sigma_c \rangle$ consists of a set of variables Σ_v, a set of predicate symbols Σ_p and a set of constant symbols Σ_c from a first-order vocabulary ϕ.

A *logic program* P *rule* consists of finite set of the *rules* of the following form:

$$H_1, ..., H_m \leftarrow B_1, ..., B_n.$$

where each $H_i, 1 \leq i \leq m$ and $B_j, 1 \leq j \leq n$ is an atomic formula.

In case $m = 1$, rule is called *Horn*, if $n = 0$, then the rule is called a *fact*, and finally in case H is empty that is $m = 0$, then the rule is called a goal.

We will call a *literal* L either an atom A or its strong negation $\neg A$.

The result of the logic program P is the question whether goal statement holds in this program.

The idea of the fixed point semantics is to incrementally increase the knowledge that is true in program. Informally, it starts with the facts which are always true, as they have no premises. If we pick up each rule $H \leftarrow B_1, ..., B_n$ and check whether all of $B_1, ..., B_n$ are true in the current knowledge, then this knowledge can be increased with H. This procedure is repeated until no more information can be added and a fix-point is reached.

In what is below we provide a formal definition of the approach. We will follow notion used in Fitting [39]

We will call the set of elements $B_1, ..., B_n$ as the body of rule and denote by $B(r)$ and the element H as the head of rule and denote it by $H(r)$.

We will also provide a procedure of grounding that allow to transform programs to variable-free analogs. Informally, this procedure replaces all variable occurrences by possible domain elements, which number in case of nested function symbols can be infinite. This is, however, impossible in the implementation DLVHEX as it explicitly forbids function symbols.

A term t is called *ground* iff it does not contain any variable.

Definition 2.2.1. A *grounding* of program P denoted as $ground(P)$ is a program P' where all ground terms are replaced by themselves, variables in non-ground terms are replaced by all possible ground terms.

Example 2.2.1. We illustrate application of grounding for the following program P.

$$\begin{aligned}&male(joe).\\&person(george).\\&person(X) \leftarrow male(X).\end{aligned}$$

In this program we assume there are two possible domain elements: joe and $george$. As such rule $person(X) \leftarrow male(X)$. should be replaced by their ground instances in order to obtain grounding of program P.

$$\begin{aligned}ground(P) = \{&male(joe).\\&person(george).\\&person(joe) \leftarrow male(joe).\\&person(george) \leftarrow male(george).\}\end{aligned}$$

We will explain now the idea of least fixed point semantics. Informally, this semantics picks up each rule $H \leftarrow B_1, ..., B_n$ and whenever all the body elements B_i are set to true, H is set to true as well. Therefore, the number of variables set to *true* is only increasing until no more atoms can be set to *true*.

We will formally define an operator $\Gamma(P, A)$ for a logic program P and a A - a set of atoms.

Definition 2.2.2. Operator $\Gamma(P, A)$ produces another set of atoms A', s.t. $H \in A'$ iff there exists a ground instance of a rule with having H in the head and the body being true in A.

$$\Gamma(P, A) = \{H \mid \exists H \leftarrow B_1, ..., B_n \in ground(P), s.t. \forall i(1 \leq i \leq n) B_i \in A\}$$

The semantics of the program P in such case is the least fix-point of this operator $lfp(\Gamma(P, \emptyset))$.

Example 2.2.2. Consider program P.

$$\begin{aligned}&p.\\&q \leftarrow p.\\&r \leftarrow p.\\&s \leftarrow q, r.\end{aligned}$$

Let $\Gamma_i(P, A)$ be the set obtained at each step *i* of applying least fix point operator.

$$\begin{aligned}\Gamma_0(P, \emptyset) &= \{p\}\\\Gamma_1(P, \{p\}) &= \{p, q, r\}\\\Gamma_2(P, \{p, q, r\}) &= \{p, q, r, s\}\\\Gamma_3(P, \{p, q, r, s\}) &= \{p, q, r, s\} = lfp(\Gamma(P, \emptyset))\end{aligned}$$

Least fix point approach always guarantees that the knowledge induced in program is always increasing. We will need more sophisticated techniques for deriving negative information though, but before moving further, we will have a look on the classical model approach.

Model-theoretical approach

Another point of view on the logic programs is a classical model approach, where we are interested in any model that satisfies the program. Because of this decision, there may be infinitely many models for a single program. For building a new model from existing one, we can simply add any literal not occurring in program to the current interpretation. We will now define formally classical models.

Definition 2.2.3. A model is a first-order interpretation I of program P iff for each rule $r = H_1 \leftarrow B_1, ..., B_n$ of grounded program $ground(P)$ following statement holds: if $B(r) \subseteq I$, then $H(r) \cap I \neq \emptyset$.

As such, the only requirement for the interpretation is to satisfy the head whenever the body is satisfied.

Example 2.2.3. Let P be the following logic program.

$$\begin{aligned}&p(a).\\&q(X) \leftarrow p(X).\end{aligned}$$

Assume the following constants can be used in the program:

$$\{a, a_2, ..., a_i, ..., b, b_2, ..., b_j, ..., c, c_2, ..., c_k, ...\}$$

The following interpretations, including infinite ones, are the models of the program:

$$\begin{aligned}I_1 &= \{p(a), q(a)\}\\I_2 &= \{p(a), q(a), q(a_2)\}\\I_3 &= \{p(a), q(a), p(a_2), q(a_2), ..., p(a_i), q(a_i), \forall i \leq \infty\}\end{aligned}$$

Whereas the following interpretations are not models of the program:

$$\begin{aligned}I'_1 &= \{p(a)\}\\I'_2 &= \{p(a), p(a_2), q(a_2), ..., p(a_i), q(a_i), \forall i \leq \infty\}\end{aligned}$$

Even though formally all interpretations I_1, I_2, I_3 are the models as per definition, the model I_1 looks more appropriate than others. This model contains only atoms relevant to the program. By relevant atoms we mean those atoms that are built from the constants occurring in program. Interpretation I_2, I_3 contain constants a_i, which do not add any value to the interpretation. To overcome such problem we will define a notion of Herbrand models in next section.

Answer-Set Semantics

In what is following we will present stable model semantics as defined by Gelfond and Lifschitz [26]. This formalism will have necessary features for our purposes.

In what is following we will define a notion of Herbrand models that are used in answer-set programming. With such models we will overcome the problem defined in previous section of models that are more natural than others.

The *Herbrand universe* $HU(P)$ of a logic program P is the set of all ground terms occurring in P.

The *Herbrand base* $HB(P)$ of a logic program P is the set of all ground atoms over predicates in P and terms over Herbrand universe $HU(P)$.

The *Herband interpretation* I of a logic program P is subset of Herbrand Base $I \subseteq HB(P)$.

Having Herbrand interpretations defined, it is natural to take Herbrand models over others. The reason is that they will contain only such constants that occur in program. This approach seems reasonable, however, there might be problems.

Example 2.2.4. One of such problems is illustrated in the following program.

$$p(a).$$
$$q(X) : -r(X)$$

There are two Herbrand models for this program, $M_1 = \{p(a)\}$, $M_2 = \{p(a), q(a), r(a)\}$. The model M_2, however, contains facts that are unfounded. That is, in particular $r(a)$ does not appear anywhere in program as a fact or in any head of the rules. To overcome such problem we should select *minimal* models with respect to set inclusion among other ones.

As per [26], the logic programs without negation will have only one minimal Herbrand model that is computed by fix-point procedure defined above. In what is following we will define additional types of programs that are dealing with both classical negation and negation by default.

Definition 2.2.4. *Extended logic program* P is a logic program with each rule having form of:

$$H_1 \vee ... \vee H_k \leftarrow B_1, ..., B_n, not\ B_{n+1}, ..., not\ B_m$$

Each of the $H_i, 1 \leq i \leq k$ is an atom, $B_j, 1 \leq j \leq m$ is a literal. We will denote $B^+(r)$ as part of rule body $B(r)$ that contains only literal without default negation and $B^-(r)$ with default negation respectively.

Definition 2.2.5. Let P be an extended logic program defined as above and $M \subseteq HB(P)$ be an arbitrary Herbrand interpretation. Then the $reduct$ P^M of program P with respect to M is the set of rules obtained from program $ground(P)$ by

- removing all rules which have at least one default-negated atom in $M : B^-(r) \cap M \neq \emptyset$;
- in the remaining rules removing all default-negated atoms. If the minimal Herbrand model of reduct P^M is the same as M, then M is called answer set.

Example 2.2.5. Let P be the following program.

$$p \leftarrow not\ q,$$
$$q \leftarrow not\ p$$

This program has two answer sets $M_1 = \{p\}$ and $M_2 = \{q\}$. Consider reducts of program P under M_1 and M_2, $P^{M_1} = \{p\} = M_1$, $P^{M_2} = \{q\} = M_2$

We will reuse answer-set semantics in HEX programs, which are defined in the following section. We need the following formalism in order to be able to solve the constraint answer-set programs.

2.3 HEX Programs

In order to implement necessary constraint answer-set programming, we need to define a mechanism to interact with the constraint solver. We need to pass as an input the set of constraints and answers the question, whether they are consistent or not. The external constraint solver should also receive the data in the constraint form that compromises to its own language.

For this reason, we will now use very convenient formalism, which allows to solve this problem. Particularly, we define logic programs under answer-set semantics as so-called generalized HEX programs, which is extension of extended logic program [18]. The DLVHEX system is an example of implementation of such system.

Provided below is a brief introduction to it with focusing mainly on the new concept of the *external atom.*

External atoms

An important extension of extended logic program is the notion of the external atom. Such atoms provide bidirectional exchange of information between the HEX program and external reasoner.

The syntax difference from ordinary atoms is that each external atom is required to be prefixed with symbol $\&$.

Definition 2.3.1. An *external atom* $\&g$ is an expression of a form $\&g[X_1, ..., X_m](Y_1, ..., Y_n)$, where g is a fixed predicate name, each of $X_1, ..., X_m$ is an input predicate, $Y_1, ..., Y_n$ is an output predicate.

Each of $X_i, 1 \leq i \leq m$ and $Y_j, 1 \leq j \leq n$ can be a predicate name, a constant or a variable. When the external reasoner gets called, it receives all atoms with predicate parameter which are true in current interpretation. These predicates are passed as a restricted interpretation. Based on the input parameters external reasoner produces a set of output tuples of arity n.

We will define the semantics of external atoms with respect to the assignment A as following. Let $f_{\&g}$ be a Boolean *oracle function* that is defined over all possible values of A, **p** and **c**, where **p** is the set of constant input parameters and **c** is the set of output terms. For this function it holds that $\&g[\mathbf{p}](\mathbf{c})$ is true iff $f_{\&g}(A, \mathbf{p}, \mathbf{c}) = 1$.

Definition 2.3.2. A HEX program P is a set of rules of form

$$H_1 \vee \cdots \vee H_k \leftarrow B_1, \cdots, B_n, not\ B_{n+1}, \cdots, not\ B_m$$

where each H_i is a higher-order atom, B_i is a higher-order atom or external atom.

The semantics of HEX programs is defined in [18]. Instead of the *Gelfond-Lifschitz reduct*, the *FLP-reduct* [19] is used that ensures the minimality of the models.

Definition 2.3.3. Let program P be defined as above and I be an interpretation. The *FLP reduct* fP^I is defined as $fP^I = \{r \in ground(P) \mid \text{if } B^+(r) \subseteq I, B^-(r) \cap I = \emptyset \text{ then } H(r) \cap I \neq \emptyset\}$

Informally, the FLP reduct is the set of all ground rules that are satisfied by I. A ground rule is satisfied whenever all positive body atoms are in the interpretation I and all negative are not in the interpretation I, at least one atom from head should be in the interpretation. The FLP reduct is named after Faber, Leone and Pfeifer and was introduced in [19]. It was also shown in [19] that the definition of FLP reduct is equivalent to the definition of the traditional Gelfond - Lifschitz reduct.

An interpretation I is an *answer set* of HEX program P iff I is minimal model of fP^I with respect to subset inclusion.

Example 2.3.1. Let the program P be as following

$$\begin{aligned} p &\leftarrow not\ q, & r &\leftarrow p, \\ q &\leftarrow not\ p, & r &\leftarrow q \end{aligned}$$

The FLP reduct of program relative to $I = \{p, r\}$ is

$$fP^I = \{p \leftarrow not\ q, r \leftarrow p\}$$

An example of system implementation with HEX program support is DLVHEX. Support for various reasoners was provided in this system, like Description Logics reasoner and others. [1] We want to reuse this system to provide the support for constraint logic programming, which is defined below.

[1]http://www.kr.tuwien.ac.at/research/systems/dlvhex

2.4 Constraint logic programming

As we introduced both constraint and answer-set programming, we will now consider an extension of logic programming that can handle constraints. The very first ideas were described in [6] and then [38], where a hybrid language *AC* was proposed. In the following subsection, we will follow the more recent approach in [23] .

We will split ordinary and constraint atoms and denote them as $\mathcal{A}$ and $\mathcal{C}$ respectively. For the following, we will always prefix constraint operators with the \$ sign. Each constraint c has the form of $c = X \odot Y$, where X and Y are the constraint expressions and $\odot \in \{=, \neq, <, >, \leq, \geq\}$. Constraint expressions might contain ASP variables, constraint variables, numerical literals and arithmetic operators $\cdot \in \{+, -, *, /\}$.

Constraints and constraint atoms are associated with the function $\gamma : \mathcal{C} \to C$. We will also extend the usage of the function over the set of constraint atoms : $\gamma(Y) = \{\gamma(c) \mid c \in Y\}$ for a subset $Y \subseteq \mathcal{C}$. As such, we can denote the set of used constraints in the program by $C(P) = \gamma(atom(P) \cap \mathcal{C})$, where $atom(P)$ returns set of all atoms in program P.

Each constraint variable has a domain, which is defined by special atom $\$dom(X..Y)$ with X and Y are lower and upper integer bound of domain. We will use notion $D(P)$ when representing domain of any variable in a program P.

Definition 2.4.1. A *constraint logic program* P is defined as a logic program over alphabet $\mathcal{A} \cup \mathcal{C}$.

Definition 2.4.2. An assignment $I : V(P) \to D(P)$ over $\mathcal{A} \cup \mathcal{C}$ is called *interpretation* of a constraint logic program.

Definition 2.4.3. A *constraint reduct* P^I of a constraint logic program P over $\mathcal{A} \cup \mathcal{C}$ and an assignment $I : V(P) \to D(P)$ is called following set:

$$\begin{aligned} P^I = \{&head(r) \leftarrow body(r)_{|\mathcal{A}} \mid r \in P, \\ &\gamma(body(r)_{\mathcal{C}^+}) \subseteq sat_{\mathcal{C}(P)}(I),\ \gamma(body(r)_{\mathcal{C}^-}) \cap sat_{\mathcal{C}(P)}(I) = \emptyset\}. \end{aligned}$$

Definition 2.4.4. A set $X \subseteq \mathcal{A}$ is a *constraint answer set* of the constraint program P with respect to the assignment I iff X is the answer set of the constraint reduct P^I. We will denote constraint answer set as a tuple $\langle \mathcal{A}' \subseteq \mathcal{A}, \mathcal{C}' \subseteq \mathcal{C} \rangle$, representing set of ordinary and constraint atoms that are true in the answer set.

Example 2.4.1. In order to illustrate the use of constraint logic programming, consider the following puzzle that is taken from [5]: *There are either 2 or 3 brothers in the family. There is a 3 year difference between one brother and the next (in order of age). The age of the eldest brother is twice the age of the youngest. The youngest is at least 6 years old. How many brothers are there in family?*

The following program P represents this puzzle:

% Necessary facts
$\$dom(1..20).$
$index(1).$
$index(2).$
$index(3).$

% There can either 2 or 3 brothers
$num\_brothers(2) \leftarrow \textit{not } num\_brothers(3).$
$num\_brothers(3) \leftarrow \textit{not } num\_brothers(2).$

% The first brother is the eldest
$eldest\_brother(1).$

% The second or third(as in num_brothers) is the youngest one
$youngest\_brother(B) \leftarrow index(B), num\_brothers(B).$
$is\_brother(B) \leftarrow index(B), index(N), num\_brothers(N), B \; \$\leq N.$

% There is a 3 year difference between brothers
$age(B1) \; \$- \; age(B2) \; \$= 3 \leftarrow is\_brother(B1), is\_brother(B2), B2 \; \$= B1 \; \$+ 1.$

% The age of the eldest brother is twice the age of the youngest
$age(BE) \; \$= age(BY) \; \$ * 2 \leftarrow eldest\_brother(BE), youngest\_brother(BY).$

% The youngest is at least 6 years old
$age(BY) \; \$\geq 6 \leftarrow youngest\_brother(BY).$

One of the constraint answer-sets for this program is the following set

$$\langle\{num\_brothers(3), index(1), index(2), index(3), is\_brother(1),$$
$$is\_brother(2), is\_brother(3), eldest\_brother(1), youngest\_brother(3)\},$$
$$\{age(1) = 12, age(2) = 9, age(3) = 6\}\rangle$$

We will reuse this example in the following chapters.

This concludes the necessary preliminaries for the constraint and logic programming. We have shown the motivation for the original concepts for these paradigms and how they effectively can be combined together. We will need these concepts further for developing our own integration of constraint and answer-set programming in the following sections.

CHAPTER 3

Encoding constraint logic programs into answer-set programs

In this section we are going to present the theoretical background for a custom constraint answer-set programming plugin for DLVHEX. We provide a procedure to transform constraint logic programs to answer-set programs with external atoms for calling the constraint solver. As such, this translation will provide an ability to run constraint answer set programs as HEX programs and compute their answer sets. In the following realization, we will be focusing on computing consistency of the given set of constraints when evaluating it with the external constraint solver.

3.1 Constraint consistency check

Let set C be the set of constraints possibly satisfied or not with respect to some interpretation. Consider an ordinary constraint logic program, which contains a set $\mathcal{C}$ of constraint atoms and set $\mathcal{A}$ of ordinary atoms. A particular constraint atom either belongs to the potential constraint answer set or does not belong to it. If the atom belongs to the constraint answer set, then the respective constraint should be satisfied. Consequently, if the constraint atom is not present in the answer set, the constraint should not be satisfied. We want to be able to answer the question: given the set $\mathcal{C}$ of such constraint atoms and a particular interpretation I, is it possible to find necessary assignments to constraints defined by set $\mathcal{C}$. In order to do that, we first need to convert constraint atoms to the set C of constraints.

We will denote $\mathcal{T}$ as the set of terms occurring in program P. In what is following, we will use simple primitive function for transforming the constraint atoms.

Let function $Str : \mathcal{C} \rightarrow \mathcal{T}$ be a primitive function that transforms any constraint atom to a string term that contains a string representation of this atom. For example :

$$A = age(B1)\ \$\text{–}\ age(B2)\ \$\!=\ 3$$
$$Str(A) = \texttt{age(B1)}\ \$\text{–}\ \texttt{age(B2)}\ \$\!=\ \texttt{3}$$

In order to distinguish constraint atom from its respective string literal, we will use typewriter font for describing the string literal, as was just shown.

We will now split these two atoms in two parts. Let G be arbitrary constraint atom and $F = Str(G)$ be simply string representation of this atom. Let $V_1, ..., V_n$ be the set of answer-set variables occurring in this constraint atom. Let in addition ordinary atom $A = expr(F, V_1, ..., V_n)$ be the atom, for which respective constraint atom holds in the constraint answer set or let it be $A = nexpr(F, V_1, ..., V_n)$, if it does not hold. We will define them formally later on. Given this set of atoms, we want to check whether there is a satisfying assignment for them.

We will firstly replace the occurrences of answer-set variables in such constraint atoms.

Let $A = expr(F, V_1, ..., V_n)$ or $A = nexpr(F, V_1, ..., V_n)$ be an ordinary atom and let also $A' = expr(F, x_1, ..., x_n)$ or $A' = nexpr(F, x_1, ..., x_n)$ be a ground atom in the interpretation I which has the same predicate as A. Having two if them, we can define a trivial variable assignment V that assigns each of the variable $V_1, ..., V_n$ to the actual value $x_1, ..., x_n$ from program domain.

A function *substitute* which replaces all occurrences of variable V_i in A' with respective grounded value is defined inductively.

$$substitute(A) = \begin{cases} F, & \text{if } F \text{ literal or constraint variable} \\ x_i, & \text{if } F = V_i \text{ is a variable} \\ substitute(F_1) \odot substitute(F_2), & \text{if } F = F_1 \odot F_2, \\ & \odot \in \{+, -, \cdot, /, =, \neq, <, >, \leq, \geq\} \end{cases}$$

Once we have a usual constraint, we need to distinguish $expr$ and $nexpr$ atoms. Let us fix a particular constraint $c \in C$. A constraint *complement* $\bar{c}$ is defined by inverting the main constraint operator $\odot$.

$$\bar{c} = \begin{cases} X \neq Y, & \text{if } c \text{ is } X = Y \\ X = Y, & \text{if } c \text{ is } X \neq Y \\ X \leq Y, & \text{if } c \text{ is } X > Y \\ X \geq Y, & \text{if } c \text{ is } X < Y \\ X < Y, & \text{if } c \text{ is } X \geq Y \\ X > Y, & \text{if } c \text{ is } X \leq Y \end{cases}$$

We are now ready to provide main definition of transforming ordinary ground atom to constraint.

Definition 3.1.1. Let $F \in \mathcal{A}$, $A = expr(F, V_1, V_2)$ or $A = nexpr(F, V_1, V_2)$ be an ordinary ground atom. Let also $A' = substitute(A)$. A function $\theta : \mathcal{A} \to C$ with respect to interpretation I over logic program P that transforms an ordinary atom to a constraint is defined as following:

$$\theta(A) = \begin{cases} A' & \text{if } A = expr(F, V_1, ..., V_n) \\ \bar{A}' & \text{if } A = nexpr(F, V_1, ..., V_n) \end{cases}$$

Informally, we firstly obtain F' by replacing all occurrences of variables $V_1, ..., V_n$ with $x_1, ..., x_n$ values and then convert them to the constraint or its complement.

In the same manner, $\theta^{-1} : C \to \mathcal{A}$ performs the inverse operation:

$$\theta^{-1}(A') = \begin{cases} expr(F, x_1, ..., x_n) & \text{if } A' = F, V_i = x_i, 1 \leq i \leq n \\ nexpr(F, x_1, ..., x_n) & \text{if } \bar{A}' = F, V_i = x_i, 1 \leq i \leq n \end{cases}$$

Having the definition of function θ we can now transform ordinary ground atoms to constraints such that later the constraint solver can solve the problem of their consistency.

Example 3.1.1 (ctd.). Let us fix a particular constraint atom from the previous brother's example 2.4 that is $age(BY)$ \$$\geq$ 6. Suppose $age(1)$ \$$\geq$ 6 atom is not satisfied in some interpretation I. In such case respective ordinary ground atom is $expr(\texttt{age(BY)}$ \$$\geq$ $6, 1)$ and finally the constraint obtained by function θ is simply $c : age(1) \geq 6$

We will now define the external atom for the constraint solver which will perform the consistency check of the constraints.

Definition 3.1.2. Let $\&casp[dom, expr, nexpr]()$ be an external atom of input arity 3 and output arity of 0. This atoms will check the consistency of the constraints that are passed as parameters. We will define the following semantics to it with respect to the interpretation I over logic program P.

Let the set of ordinary atoms passed in extension of predicates $expr$ and $nexpr$ is denoted by K. Then the set of respective constraints is defined as $C = \{\theta(F) \mid F \in K\}$.

Let V be the set of all constraint variables occurring in the constraint set C defined above.

Let D be the domain, as defined by the *dom* predicate.

Then evaluation of the external atom $\&casp$ in such case is the following:

$$f_{\&casp}(I, \{dom, expr, nexpr\}, \emptyset) = \begin{cases} 1, & \text{if there exists a solution to CSP} = \langle V, D, C \rangle \\ 0, & \text{otherwise} \end{cases}$$

Example 3.1.2 (ctd.). Consider the same constraint $c : age(1) \geq 6$. In case the solution to the CSP assigns value of constraint variable $age(1)$ to the value greater or equal to 6, then $\&casp[dom, expr, nexpr]^I = \{()\}$, otherwise $\&casp[dom, expr, nexpr]^I = \emptyset$

Having the external atom defined, we are now able to solve the problem in case the constraint atom is represented as a special ordinary atom with $expr$ or $nexpr$ predicate. The only problem now is that we need to be able to transform the traditional constraint atoms to such atoms. We will explain this in the following section.

3.2 Constraint logic programs translation

As a start of translation procedure we will try to replace constraint atoms with ordinary atoms in answer-set programming that can represent constraints. In order to do that, we will translate constraint atoms to string literals. Such literals will be an argument to special auxiliary atoms, each of which after grounding will represent a single constraint. A predicate $\$dom$ for the constraint variable domain will be handled in a special way. It is not considered to be a constraint, but only as another parameter to the external solver. Furthermore, we need to add additional rules for checking consistency and solving the constraint part of the problem. Formal definition of such approach is provided below. We will reuse the syntax of constraint answer-set programming as defined previously.

In what is following, we will replace constraint atoms with ordinary HEX atoms. Let the set Dom be the set of all atoms with $\$dom$ predicate.

Definition 3.2.1. Let $convert : |Dom \cup \mathcal{C} \cup \mathcal{A}| \to \mathcal{A}$ be the function that maps constraint answer-set constructs to traditional answer-set atoms with special auxiliary predicates.

$$convert(A) = \begin{cases} dom(Str(X)), & \text{if } A = \$dom(X) \\ expr(Str(A), V_1, ..., V_n), & \text{if } A \in \mathcal{C} \\ A, & \text{otherwise} \end{cases}$$

where $\{V_1, ..., V_n\}$ is the set of answer-set variables occurring in A.

This translation means that if the constraint domain is specified, the domain bounds should be translated to an ordinary atom with the string representation of the domain. Whenever atom A in the rule is a constraint atom that is $A \in \mathcal{C}$, then it should be replaced with the atom $expr(Str(A), V_1, ..., V_n)$. If the atom is not a constraint atom, then this ordinary atom should be left as it is. After this step we can represent every constraint construct using ordinary answer-set atoms.

We can extend this conversion to whole rule $r = H \leftarrow B_1, ..., B_n, not\ B_{n+1}, ..., not\ B_m$:

$$\begin{aligned} convert(r) = convert(H) \leftarrow & convert(B_1), ..., convert(B_n), \\ & not\ convert(B_{n+1}), ..., not\ convert(B_m) \end{aligned}$$

Example 3.2.1 (ctd.)**.** Consider the following rule from brother puzzle which requires the age of youngest brother to be greater or equal to 6:

$$age(BY)\ \$\geq 6 \leftarrow youngest\_brother(BY).$$

In this rule there is a constraint atom with a single ASP variable in the head of the rule and an ordinary atom in the body of the rule. After application of the function *convert* the constraint atom $age(BY)$ $\$ \geq$ 6 is replaced by an ordinary atom having *expr* predicate, whereas the ordinary atom $youngest\_brother(BY)$ is left as it is.

$$expr(\texttt{age(BY) \$}\geq \texttt{6}, BY) \leftarrow youngest\_brother(BY).$$

Each of the converted ordinary atoms might or might not be satisfied. In order to allow such behavior, we will add a guessing rule for the obtained translation of constraint atom. An additional atom with predicate *nexpr* is defined as part of the guessing rule. It takes the same parameters and *expr* atom and state that the constraint expression should not hold when the *nexpr* atom is evaluated to true.

Definition 3.2.2. Guessing function $guess(r)$ for the rule r is the mapping from a rule to a set of rules defined as:

$$\begin{aligned} guess(r) = \{&expr(Str(A), V_1, ..., V_n) \vee nexpr(Str(A), V_1, ..., V_n) \leftarrow B(r) \\ &\mid A \in (B(r) \cup H(r) \cap \mathcal{C})\} \end{aligned}$$

Example 3.2.2 (ctd.)**.** Consider again the rule defined above that the age of the youngest brother is greater or equal to 6. In this rule, there is a constraint atom in the head of the rule, for which a guessing rule will be added:

$$\begin{aligned} expr(\texttt{age(BY) \$}\geq \texttt{6}, BY) \vee nexpr(\texttt{age(BY) \$}\geq \texttt{6}, BY) \leftarrow \\ youngest\_brother(BY). \end{aligned}$$

Consequently, we will define the translation of the rule as a combination of $convert$ and $guess$ functions.

The *translation* of the rule $r = H \leftarrow B_1, ..., B_n, not\ B_{n+1}, not\ B_m$ is the set of rules

$$tr(r) = convert(r) \cup guess(r)$$

We are now ready to define the translation of the whole program. Once we transformed all constraint atoms to ordinary atoms the only missing thing is the addition of the constraint consistency check.

A *translation* of program P is the set of rules

$$\begin{aligned} tr(P) = &\{tr(r) \mid r \in P\} \cup \\ &\{\leftarrow not\ \&casp[dom, expr, nexpr]().\} \end{aligned}$$

Consequently, we will also define the translation of the answer set X, which is simply conversion from the constraint atoms to the auxiliary atoms *expr* and *nexpr* as defined above.

The *translation* function for the answer set X is defined as

$$tr(X) = \{convert(A) \mid A \in X\}.$$

Example 3.2.3 (ctd.)**.** Consider the program that encodes the brother puzzle. We will break it into logical pieces that reflect the transformation to the answer-set program with external atoms. Firs, the translated program contains original rules that do not have constraint atoms:

$$\begin{aligned}
&num\_brothers(2) \leftarrow not\ num\_brothers(3).\\
&num\_brothers(3) \leftarrow not\ num\_brothers(2).\\
&index(1).\\
&index(2).\\
&index(3).\\
&eldest\_brother(1).\\
&youngest\_brother(B) \leftarrow index(B), num\_brothers(B).
\end{aligned}$$

A special case is defined for conversion of domain atom:

$$dom(\texttt{1..20}).$$

All of the following rules contain constraint atoms. As such, after application of procedure $convert(r)$ they are replaced with the auxiliary *expr* predicates:

$$\begin{aligned}
&is\_brother(B) \leftarrow index(B), index(N), num\_brothers(N), expr(\texttt{B}\ \$\leq \texttt{N}, B, N).\\
&expr(\texttt{age(B1)}\ \$-\ \texttt{age(B2)}\ \$= \texttt{3}, B1, B2) \leftarrow is\_brother(B1), is\_brother(B2),\\
&\qquad\qquad expr(\texttt{B2}\ \$= \texttt{B1}\ \$+ \texttt{1}, B1, B2).\\
&expr(\texttt{age(BE)}\ \$= \texttt{age(BY)}\ \$\cdot \texttt{2}, BE, BY) \leftarrow\\
&\qquad\qquad eldest\_brother(BE), youngest\_brother(BY).\\
&expr(\texttt{age(BY)}\ \$\geq \texttt{6}, BY) \leftarrow youngest\_brother(BY).
\end{aligned}$$

For each of the converted atoms, necessary guessing rules are added. For example, for the first one in above-mentioned rules, the following new guessing rule is added:

$$\begin{aligned}
&expr(\texttt{B}\ \$\leq \texttt{N}, B, N)\ \vee\ nexpr(\texttt{B}\ \$\leq \texttt{N}, B, N) \leftarrow\\
&\qquad\qquad index(B), index(N), num\_brothers(N).
\end{aligned}$$

Finally, constraint for checking the constraint consistency is added.

$$\leftarrow not\ \&casp[dom, expr, nexpr]().$$

The answer set for this program is the following set

$$\begin{aligned}
&\{num\_brothers(3), index(1), index(2), index(3), is\_brother(1), is\_brother(2), is\_brother(3),\\
&\ eldest\_brother(1), youngest\_brother(3), expr(\texttt{age(B1)}\ \$-\ \texttt{age(B2)}\ \$= \texttt{3}, 1, 2),\\
&\ expr(\texttt{age(BE)}\ \$= \texttt{age(BY)}\ \$\cdot \texttt{2}, 3, 1), expr(\texttt{age(BY)}\ \$\geq \texttt{6}, 3)\}
\end{aligned}$$

As such, it contains the answer to the puzzle.

We will now show several formal properties of the procedure which was just shown.

3.3 Formal properties

In the remainder of this section, we will prove that constraint logic program translation always terminates and that it is correct.

Proposition 3.3.1. Translation of the constraint answer-set program P defined by function $tr(P)$ terminates and produces a program with finite number of rules.

Proof. The original program P contains a finite set of rules and each rule contains a finite set of atoms. Function $convert$ is applied only finite number of times for each atom in the rule. Function $guess$ also produces only finite number of rules, as for each constraint atom only a single rule is added. So, the application of function $tr(P)$ is terminating. □

We will now show the correctness of the translation. By this statement it is meant that is whenever we there exists an answer set for constraint answer-set programming, then it also exists in usual answer-set context given the translation procedure we have defined and vice versa. We will also need with additional proposition.

Proposition 3.3.2 (Guessing rule satisfaction)**.** If the rule

$$\leftarrow \mathit{not}\ \&\mathit{casp}[\mathit{dom}, \mathit{expr}, \mathit{nexpr}]().$$

is satisfied under assignment X, then the guessing rules defined by the procedure $guess(r)$ in program $tr(P)$ are always satisfied under this assignment.

Proof. If the last rule is satisfied under assignment X, this means that there is a solution to CSP $= \langle V, D, C\rangle$ that is the assignment to constraint variables to the elements of the domain. Let us fix a particular constraint atom A, for which a guessing rule was added. If this atom under the solution to the CSP is satisfied, then the whole rule is satisfied, as the atom with *expr* predicate is satisfied. If this atom is not satisfied, then complement of the respective constraint is satisfied, and so the atom with the *nexpr* predicate is satisfied. □

Finally, we will now show the correctness of the translation.

Proposition 3.3.3. The translation function for program P is correct. Constraint answer program P has at least one answer set X iff program $tr(P)$ has at least one answer set X'.

Proof. ($\Rightarrow$) Suppose X is a constraint answer set of the program P and let $A \in \mathcal{C}$ be an arbitrary constraint atom belonging to the constraint answer set X. Then there exists an assignment I over $\mathcal{A} \cup \mathcal{C}$ such that X is the answer set over the constraint reduct P^I. We need to show that $X' = tr(X)$ is the answer set of program $tr(P)$ that is:

- If a constraint atom A is satisfied under X, then the translated constraint atom $convert(A)$ is also satisfied under X'. This follows directly from definition of conversion of the answer set.

- Constraint for checking consistency $\leftarrow not\ \&casp[dom, expr, nexpr]().$ is satisfied. Recall the definition of reduct with respect to assignment I:

$$P^I = \{H(r) \leftarrow B(r)_{|\mathcal{A}} \mid r \in P, \\ \gamma(B(r)_{\mathcal{C}^+}) \subseteq sat_{\mathcal{C}[P]}(I),\ \gamma(B(r)_{\mathcal{C}^-}) \cap sat_{\mathcal{C}[P]}(I) = \emptyset\}.$$

  If X is a constraint answer set of the program, then there exist a solution to the $CSP = \langle V, D, C\rangle$, where C is the set of constraints $sat_{\mathcal{C}[P]}(X)$ that are satisfied under current assignment X. Let $c \in C$ be an arbitrary such constraint. Then $convert(c)$ will be also satisfied in $tr(P)$. As c was arbitrary, all of such atoms are satisfied. According to semantics of $\&casp$ external atom, it will be satisfied in such case.

- Added guessing rule by the procedure $guess(r)$ for constraint atom $A \in r$ is satisfied. Because of the previous constraint satisfaction and proposition defined above 3.3.2, this statement holds.

Note that we can provide a direct correspondence for the answer set of the converted program $X' = tr(X)$ when proving ($\Rightarrow$) part of the theorem. The satisfied constraint atoms are converted to the $expr$ atoms and added to the resulting answer set.

($\Leftarrow$). If X' is the answer set of program $tr(P)$, then the following rule for the encoded program is satisfied:

$$\leftarrow not\ \&casp[dom, expr, nexpr].$$

According to the semantics of external atom $\&casp$, there is a solution to CSP with constraint set $C = \{\theta(A) \mid A \in X'\}$.

Note that we can not provide a direct correspondence from the answer set of the translated program to the original constraint answer set. The reason is that the answer set of the translated program does not contain encoded solutions to the CSP, it only has the atoms that define the satisfied constraint expressions. This information is not enough to construct the constraint answer set, but we can still prove its existence.

Let X be a constraint answer set X obtained from X' by removing all *expr* and *nexpr* atoms and an adding assignment of constraint variables to the elements of the domain. Obtained in such way set X is indeed a constraint answer set. Consider again reduct P^I and fix an arbitrary rule r in P.

- If all constraint atoms A are satisfied in the rule, that is:

$$\gamma(B(r)_{\mathcal{C}^+}) \subseteq sat_{\mathcal{C}[P]}(I),\ \gamma(B(r)_{\mathcal{C}^-}) \cap sat_{\mathcal{C}[P]}(I) = \emptyset;$$

  then all respective conversions $convert(A)$ are also satisfied in the rule. According to the definition of reduct, this rule is also presented in the reduct with removed constraint atoms. In such case, this rule in reduct is the same as its translation $tr(r)$ where all $expr$ atoms are satisfied.

- If some of constraint atom A is not satisfied in the rule, then this atom A does not belong to any rule in the reduct of the program. According to definition of $convert(A)$, the converted ordinary atom will not be satisfied and so the body of this rule will not be satisfied. In such case this rule can be removed from translated program and the answer set will not be changed.

So, the obtained set X is indeed constraint answer set of the program P. $\square$

3.4 Global constraints support

Global constraints are the type of constraints that defines relations between an arbitrary number of variables. A particular example is a global constraint $alldifferent(x_1, ..., x_n)$ where the arguments is the list of variables $x_1, ..., x_n$ that can have arbitrary length. Such constraints are maintained and comprehensive list of them is provided in the following page. [1] As of now, we support following 3 global constraints: sum, $maximize$, $minimize$ in CASP plugin.

Syntax and semantics of global constraints

We will start with defining syntax and semantics for supported global constraints.

The sum global constraint is used to represent the sum over particular values in predicate extensions. For example, if $salary(john, 5)$ and $salary(joe, 10)$ provide information about salary information about single worker then $sum(salary, 2)$ would provide information about total salary of all workers and will be equal to 15. Let $\$sum(p, i)$ be a special constraint atom that takes as input two parameters: a predicate name p and an index i (1-based). When evaluating the value constraint atom over a particular interpretation I, the value of $\$sum(p, i)$ is replaced with $\sum a_i$, where $p(a_1, ..., a_n) \in I$.

A $maximize$ global constraint is defined by a special constraint atom $\$maximize(F)$ that takes as an input another constraint atom F that is required to be maximized. The $maximize(F)$ atom is satisfied whenever the assignment for the constraint atom F is maximum among any other possible satisfying assignments.

A similar definition is used for $minimize$ global constraint syntax and semantics.

We will now provide an example of an assignment problem, where global constraints can be used.

Example 3.4.1. *There are number of tasks to complete by a single worker. Each task requires t hours to finish. A worker can not work more than 40 hours per week and is willing to complete maximum number of tasks. What tasks can he pick up for this purpose?*

Let fact that a task X requires T hours to finish is represented by fact $task(X, T)$. In order to solve the problem, each of the task can be either assigned or not assigned. We will denote by

[1] http://www.emn.fr/z-info/sdemasse/gccat/

$task\_assignment(X, 1)$ the case that task X was assigned to worker and by $task\_assignment(X, 0)$ that this task was not assigned to him:

$$task\_assignment(X, 1) \leftarrow not\ task\_assignment(X, 0).$$
$$task\_assignment(X, 0) \leftarrow not\ task\_assignment(X, 1).$$
$$task\_assigned\_time(X, T) \leftarrow task\_assignment(X, 1), task(X, T).$$

Once the tasks are assigned, we need to satisfy the constraint that sum of times does not exceed 40. Let a special atom $sum(p, i)$ defines a constraint expression built as a sum of i-th (1-based) argument of the atoms having predicate p. Then the above-mentioned requirement is defined as the following rule:

$$\leftarrow \$sum(task\_asssigned\_time, 2)\ \$> 40.$$

Last, we want to maximize the aggregated number of tasks to perform.

$$\$maximize(\$sum(task\_assigned, 2)).$$

This concludes an example using these constraints. We will now define the formal approach for global constraints in the HEX programs.

Global constraints translation

We need to modify the translation algorithm for the constraint answer-set programs provided previously. Let P be the following set of predicates: $P = \{\$dom, \$sum, \$minimize, \$maximize\}$.

The modified conversion procedure $g\_convert(A)$ is defined as

$$g\_convert(A) = \begin{cases} p(Str(X)), & \text{if } A = p(X), p \in P \\ expr(Str(A), V_1, ..., V_n), & \text{if } A \in \mathcal{C} \\ A, & \text{otherwise.} \end{cases}$$

We will use a similar definition of conversion extension to the rule, as it was before and denote it as $g\_convert(r)$

Definition 3.4.1. The translation of rule r that supports global constraints is then

$$g\_tr(r) = g\_convert(r) \cup guess(r)$$

Let S be the set of all predicates occurring as the first argument in the sum predicate in program P: $S = \{p \mid \$sum(p, i) \in P\}$. We will now modify the external atom $\&casp$ to receive all such predicate extensions when evaluated:

Let $S' = S \cup \{dom, expr, nexpr, sum, maximize, minimize\}$. The external constraint $\&g\_casp$ for evaluating constraint consistency with global constraints is now defined as $\&g\_casp[S']()$

We need to modify its semantics accordingly.

Definition 3.4.2. Let $\&g\_casp[S']()$ be the external atom for checking the consistency of the constraints that are passed as parameters.

Let $\theta'(A) : \mathcal{A} \to A$ be function that whenever accepts $A = sum(p, i)$ as input with respect to the interpretation I produces a constraint $\sum a_i$, where $p(a_1, ..., a_n) \in I$.

Consequently, as was defined before let C be the set of constraints occurring in the program P and that are holding in interpretation I: $C = \{\theta'(A) \mid A \in I\}$.

Let V be the set of all constraint variables occurring in the constraint set C defined above.

Let D be the domain, as defined by the *dom* predicate.

Let γ be the cost function whose value is defined as a value of an expression F, if $maximize(F)$ is present in the interpretation I, or the value of an expression $-F$, if $minimize(F)$ is present in the interpretation I.

The semantics of the external atom is defined now as:

$$f_{\&g\_casp}(I, S', \emptyset) = \begin{cases} 1, & \text{if there exists a solution to COP = } \langle V, D, C, \gamma \rangle \\ 0, & \text{otherwise.} \end{cases}$$

Definition 3.4.3. The *translation* of program P with global constraints is the set of rules

$$\begin{aligned} g\_tr(P) = &\{g\_tr(r) \mid r \in P\} \cup \\ &\{\leftarrow \textit{not}\ \&g\_casp[S']().\} \end{aligned}$$

This formalism defines support for the global constraints in our plugin. We will provide more concrete details on how global constraints are implemented with use of external constraint solver further.

Formal properties

In this section we will show formal properties of the modified global constraint translation. Just as before, the translation for the program terminates and produces a finite set of the rules.

Proposition 3.4.1. Global constraint translation is correct. Constraint answer-set program P has a constraint answer set iff translation $g\_tr(P)$ will also have answer set.

Proof. The case for the non-global constraints was proven before 3.3. We now need to check only the case for the global constraints.

($\Rightarrow$) Let X be the constraint answer set of program P and let also $A \in C$ be an arbitrary constraint atom. Consider new cases for atom A:

- $A = maximize(F)$ or $A = minimize(F)$. If the atom is satisfied, then the expression X is the maximal(minimal) among any other possible for constraint answer sets. In such case the cost function $\gamma(F)$ receives maximal(minimal) value for any possible assignment. According to the definition of external atom $\&g\_casp$ it will be satisfied in such case and the constraint rule $\leftarrow \textit{not}\ \&g\_casp[S']()$. will also be satisfied.

- A contains expression $sum(p, i)$. If A was satisfied under X, then $convert(A)$ is also satisfied. This directly follows from the definition of modified function $\theta(A)$.

($\Leftarrow$) If X' is the answer set of program $g\_tr(P)$, then the following constraint rule is satisfied:

$$\leftarrow not\ \&g\_casp[S']().$$

In this case, the cost function γ that was defined from $maximize$ or $minimize$ predicate extension takes maximum or minimum value respectively. Consequently, the constraint atom $\$maximize$ or $\$minimize$ will be satisfied.

For the proof of the satisfaction of constraint atom with sum global constraint the same arguments, as in original correctness proof 3.3 can be reused. The modified θ' function translates the ordinary atoms to constraints just in the same manner as before.

So, the translation for constraint answer-set programs with global constraints is correct. $\square$

As was shown in this section, using the above-mentioned encoding we can define programs with custom syntax for constraints that have the same characteristics as the constraint answer-set programs. Internally, this program is transformed to a HEX program with external atom that calls constraint solver. The semantics of this external atom is checking the consistency of the given set of constraints. The implementation of such semantics is up to the concrete plugin and can reuse any available constraint solver. We have also shown how special global constraint can also be handled.

In the following section we are going to show extensions and improvements of this approach. The main idea is to provide more compact representation of translated programs and speed up the execution time for finding answer sets with more advanced techniques.

CHAPTER 4

Translation and evaluation optimization

This chapter contains improvements for the integration of constraint programming into answer-set programming defined in the previous section. First, we will provide an optimization of the translation algorithm defined previously that allows avoiding guessing rules for the constraint atoms in the head of the rule. Second, we will focus on conflict-driven learning algorithms that drastically speed up searching answer-sets of the programs. We will describe several existing algorithms, show their correctness for HEX programs and propose a new algorithm.

In this section we will focus only on the optimizations for ordinary constraint atoms omitting the global constraints. As such, we will use the $\&casp$ atom as the external atom for our purposes.

4.1 Improved translation

Recall the two-step translation procedure defined for the logic program P. One may note that there is redundancy in the number of rules that were added.

Example 4.1.1. We will start with an example, where a logic program P shows the existence of such redundancy. Let P be the following program:

$$\begin{aligned}
&\$dom(1..10).\\
&p(a).\\
&c(X)\$> 5 \leftarrow p(X).
\end{aligned}$$

A translation of such program as was defined in previous chapter is

$$
\begin{aligned}
tr(P) = \{&dom(\texttt{1..10}).\\
&p(a).\\
&expr(\texttt{c(X)\$> 5}, X) \leftarrow p(X).\\
&expr(\texttt{c(X)\$> 5}, X) \ \vee \ nexpr(\texttt{c(X)\$> 5}, X) \leftarrow p(X).\\
&\leftarrow \ not \ \&casp[dom, expr, nexpr]().\}
\end{aligned}
$$

Observe that adding the last rule that is the guessing rule for the constraint atom from rule head, is actually redundant. In case body of the rule $p(X)$ is satisfied, we can never derive $nexpr(\texttt{c(X)\$> 5}, X)$ appearing in the head of the guessing rule. The reason is that this is representation of the complementary constraint for which the guessing rule was added. In such rule

$$not \ \&casp[dom, expr, nexpr]()$$

will not be satisfied for such constraints. We will now prove that we can omit adding guessing rules for constraint atoms in the head of the rule and that such modified translation is still correct.

Definition 4.1.1. Let $guess'(r)$ be the modified version of guess(r) that adds guessing rules for the constraint atoms appearing in the body of the rules.

$$
\begin{aligned}
guess'(r) = \{&r' = expr(Str(A), V_1, ..., V_n) \vee nexpr(Str(A), V_1, ..., V_n) \leftarrow B(r)\\
&\mid A \in (B(r) \cap \mathcal{C})\}
\end{aligned}
$$

Definition 4.1.2. The modified translation procedure $tr'(P)$ is defined as:

$$tr'(r) = convert(r) \cup guess'(r)$$

Proposition 4.1.1. Modified translation procedure $tr'(P)$ for the constraint logic program P is correct: $tr(P)$ has an answer set iff $tr'(P)$ has answer set

Proof We will denote the answer set of $tr(P)$ as X and $tr'(P)$ as X' respectively.

($\Rightarrow$) Let us fix an arbitrary constraint atom A that appears in the head of the rule r and let $F_1 = expr(Str(A), V_1, ..., V_n)$ and $F_2 = nexpr(Str(A), V_1, ..., V_n)$. The guessing rule in such case is defined as $guess(r) = F_1 \ \vee \ F_2 \leftarrow B(r)$ and $tr(P) = tr'(P) \cup guess(r)$. Furthermore, rule $guess(r)$ is the only place in program $tr(P)$ where the atom F_2 occur. We are interested only in the cases where one of F_1 or F_2 atoms belong to the answer set X as all other atoms will not changed in the answer set X'. Consider the following cases:

- $F_1 \in X$. Let $X' = X$ in such case. We will show that X' is the answer set of $tr'(P)$.

  Consider answer-set reduct $tr(P)^X$ in this program. Specifically, the reduct of the guessing rule $guess(r)$ under the condition that $F_1 \in X$ is equal to r that is $guess(r) = r$. In such case, $tr(P)^X = tr'(P)^{X'}$. Consequently, if X is the answer set of $tr(P)$, $X = X'$ and the reducts are the same, then X' is the answer set of $tr'(P)$.

- $F_2 \in X$. We will show that $X' = X \setminus F_2$ is the answer-set of program $tr'(P)$.

  Consider answer-set reduct $tr(P)^X$ in this program. As F_2 does not occur anywhere except this guessing rule, then $tr(P)^X = tr'(P)^{X'} \cup \{F_2 \leftarrow B(r).\}$. Consider removing this rule from $tr(P)^X$. In such case F_2 can not be deduced anymore and this is the only atom that was possible to deduce. So, removing it will make X' be the answer set of $tr'(P)$

($\Leftarrow$) Let $X = X'$. We will show that X is then indeed the answer set of $tr(P)$.

The translated program $tr'(P)$ differs from the original translated program $tr(P)$ by adding guessing rules for the atoms in the head of the rule. Let r be rule with the constraint atom in the head and $guess(r)$ one of such added guessing rules. Let again $F_1 = expr(Str(A), V_1, ..., V_n)$ and $F_2 = nexpr(Str(A), V_1, ..., V_n)$. As $tr'(P)$ has an answer set X', then the last rule is satisfied

$$\leftarrow not\ \&casp[dom, expr, nexpr]().$$

If that is the case, then according to the proposition 3.3, the guessing rule $guess(r)$ is always satisfied. We will also show that X is still minimal model for grounding of the program $tr(P)$.

Suppose the opposite that is $\exists Y, Y \subseteq X$ and Y - is the answer set of $tr(P)$. As atom F_2 can occur only in the guessing rule and $tr'(P)$ does not contain such rule, then $F_2 \notin X$. Then the only possible case is that $Y = X \setminus F_1$. But then the original rule r with constraint atom in the head is not satisfied, as F_1 is not the answer set now. So Y is not the answer set, contradiction.

Consequently, $X = X'$ is the answer set of the translated program $tr(P)$.

As was defined, we may reduce the size of the translated program and as such decrease the computation time for finding answer sets. At the same time we are still able to find solutions, if they exist. We will provide another method for speeding up the computation below. □

4.2 Evaluation optimization

State-of-the-art ASP solving techniques are based on efficient conflict-driven learning algorithms. These methods originated from SAT, where the goal was to enlarge the set of the clauses that can never be satisfied. Enlarging number of such clauses would faster produce empty clause in the search tree and therefore reduce search pace. Later on, same ideas were reused in the answer-set programming. In what is following, we will provide general theoretical background on it based on [17] and show existing and new learning techniques for constraint answer-set programs.

Conflict-driven backtracking learning overview

We will start with general overview of the conflict-driven learning algorithms for HEX programs. We will firstly provide necessary concepts and definitions below.

We will call a *literal* σ either a positive or a negated ground atom $\mathbf{T}a$ or $\mathbf{F}a$, where ground atom is of the form $p(c_1, .., c_l)$, where p is predicate name and each of the $c_1, ..., c_l$ are function-free ground terms.

A *negation* of literal σ denoted as $\overline{\sigma}$ is a function such that $\overline{\mathbf{T}a} = \mathbf{F}a$, $\overline{\mathbf{F}a} = \mathbf{T}a$

We will then define a consistent set of literals $\mathbf{T}a$ and $\mathbf{F}a$ as an *assignment*.

A *nogood* $\delta = \{L_1, ..., L_n\}$ is a set of literals L_i.

Definition 4.2.1. An assignment A is a *solution* to nogood δ resp. to a set of nogoods Δ, iff $\delta \not\subseteq A$ resp. $\delta \not\subseteq A$, for all $\delta \in \Delta$.

Example 4.2.1. Let Δ be the following set of nogoods:

$$\begin{aligned}\Delta = \{\delta_1 &= \{\mathbf{T}a, \mathbf{T}b, \mathbf{F}c\},\\ \delta_2 &= \{\mathbf{F}a, \mathbf{T}b\}\}.\end{aligned}$$

and let A be the following assignment:

$$A = \{\mathbf{T}a, \mathbf{T}b, \mathbf{T}c\}$$

Then A is a solution to set of nogoods Δ, because $\delta_1 \not\subseteq A$ and $\delta_2 \not\subseteq A$.

In general, we want to find the nogoods for the programs, such that any other potential model will be not eliminated. That is whenever we add a nogood to the program any other possible assignment should still be a solution for it.

In order to do so, we need to understand the internals of HEX programs. We will briefly start with an introduction on how the external atoms are evaluated in HEX programs as was defined in 2.3

Consider now an external function $\&g[p](c)$ with respect to some assignment A that takes an predicate p as input parameter and has output parameter c.

As such, answer sets of the program P are defined by the transformation of the program with external atoms to a special intermediate state. Each external atom $\&g[p](c)$ is replaced with the so-called *replacement atom* $e_{\&g[p]}(c)$ and a guessing rule $e_{\&g[p]}(c) \ \vee \ ne_{\&g[p]}(c)$. Obtained in such manner program is called *guessing program* $\hat{P}$. The answer sets of such programs are determined in ordinary way and then converted back to non-replacement atoms. As such, we obtain *model candidate* that needs to be verified within the external atoms. Such model might not be satisfied. In case the model is still satisfied by evaluation of external atom, such set is called *compatible set*. We will provide a formal definition of this concept.

An assignment A is called a *compatible set* of program P iff

- It is an answer of the guessing program $\hat{P}$
- $f_{\&g}(A, p, c) = 1$ iff $\mathbf{T}e_{\&g[p]}(c) \in A$ for all external atoms $\&g[p](c)$ in P that is guessed values are the same as the actual output under the input from A.

A nogood δ is *correct* with respect to the program P if all compatible sets of P are solutions to δ

Whenever any assignment is not an answer-set of the program, we might add up some nogoods that are inconsistent. Later on, when the program gets evaluated with the assignment containing such nogoods, we can immediately say that such assignment can not be an answer set. This is the basic idea *conflict-driven learning* in answer set programming. In other words, we also want to add only those nogoods that do not eliminate any answer sets of our program. In other words, we are interested in *correct* learning. We will define this approach formally below.

We will use the following notion for important concepts of program. Let $A(P)$ be the set of all atoms occurring in P and let $BA(P) = \{B(r) \mid r \in P\}$ be the set of all rule bodies of P. Let $\mathcal{E}$ be the set of all external predicates with input list that occur in P. Let $\mathcal{D}$ be the set of all signed literals over atoms in $A(P) \cup A(\hat{P}) \cup BA(\hat{P})$

Definition 4.2.2. A *learning function* is a mapping $\Lambda : \mathcal{E} \times 2^{\mathcal{D}} \to 2^{2^{\mathcal{D}}}$.

Definition 4.2.3. A learning function Λ is *correct* for the program P iff every $d \in \Lambda(\&g[p], A)$ is correct for P, for all $\&g[p] \in \mathcal{E}$ and $A \in 2^{\mathcal{D}}$.

The point of the learning function is to provide a set of nogoods that can be added to program based on the current input and output predicates to the external atom. For algorithms of finding such sets refer to [17]. In this thesis we will focus on *user-defined learning* that is the learning algorithm that depends on the internal procedure of finding constraint consistency.

Example 4.2.2. Consider the following example. Let $\&bag$ be an external atom that takes a single predicate input q. The output of the $\&bag$ external atom is $overflow$ if the sum of the elements in extension of q is greater then 10, otherwise the output is ok. Consider then the program consisting of the rules:

$$\begin{array}{l}
p(5).\\
p(8).\\
p(15).\\
p(20).\\
q(X)\ \vee\ nq(X) \leftarrow p(X).\\
result(X) \leftarrow \&bag[q](X).\\
\leftarrow not\ result(ok).
\end{array}$$

Let then Λ be the following learning function. In case one element is assigned to true that is belonging to the extension of p is already bigger then 10, then we can add this element as a nogood. Naturally, such Λ is indeed correct.

A general procedure would generate 2^4 candidate answer-sets and evaluate them over the external atom. Let the following assignment was passed to external atom $\{\mathbf{T}p(5), \mathbf{F}p(8), \mathbf{T}p(15), \mathbf{T}p(20).\}$.

In such case we can immediately add $p(15)$ as the first nogood and $p(20)$ as the second nogood respectively. Then all candidate answer-sets having $p(15)$ or $p(20)$ in their extensions can be removed from the evaluation.

We want to implement similar algorithms in the case of constraints consistency. In case the set of constraints is inconsistent, only a small subset may actually lead to inconsistency itself. If we are able to provide an effective procedure for finding such sets once the answer-set is evaluated over external atoms, we will significantly reduce the search space. We will provide several user-defined learning algorithms for such purpose, prove their correctness and explain their advantages.

Algorithms for finding irreducible inconsistent sets

As was defined above, we now want to find correct learning functions for the external plugin checking for the constraint consistency. If the external atom is not satisfied, then the constraint set contained in extension of predicate *expr* is not satisfied. We want to find a minimal cause for this that is an irreducible inconsistent set of constraints.

The basis for that is defined at the level of constraints programming. In what is presented below we will provide algorithms for finding such irreducible inconsistent sets. We will start with a naive approach, then follow techniques defined in [23] and reuse their correctness and provide new algorithm for finding IIS.

Naive approach

Let $C = \{c_1, c_2, ..., c_n\}$ be the set of constraints that is inconsistent. We want to be able to find out effectively an irreducible inconsistent set $C' \subseteq C$. Once we find a set of such constraints we need to transform the constraint set to the set of ordinary atoms defined by the $expr$ predicate. These ordinary atoms can then be added as nogoods to the program.

One straightforward algorithm is simply to check all possible subsets of C, check their consistency and pick up minimal sets among them with respect to set inclusion $\subseteq$. The running time of this approach, however, is $O(2^{|C|})$, makes it impractical.

Moreover, this approach will check consistency of all subsets of given sets of constraints, even though some of them might not be in candidate models. This leads to adding nogoods that are never going to be used and drastically increases the size of the program. As such, naive learning might be even worse than direct evaluation without learning in some cases.

Hence, we need a more sophisticated approach to this problem.

Deletion filtering

The idea of deletion filtering, presented in [23], is to iteratively remove each possible constraint. We will start with the initial set of constraints C, and each constraint c_i, test whether $C \setminus \{c_i\}$ is still inconsistent. In case it is consistent, this constraint adds up to IIS and it can not be removed from the set. In case the set is inconsistent after removal, this literal can be safely removed and algorithm restarts with the set $C \setminus \{c_i\}$. Once all constraints c_i were checked, algorithm terminates.

The algorithm is provided below. 4.2

Algorithm 4.1: Deletion filtering

```
input  : Inconsistent set of constraints $I$
output: Irreducible inconsistent set of constraints $I'$
1 $I' \leftarrow I$;
2 for $i \leftarrow 1$ to $n$ do
3     if $I' \setminus \{c_i\}$ is inconsistent then
4         $I' \leftarrow I \setminus \{c_i\}$
5     end
6 end
```

We will now explain its application on a simple example.

Example 4.2.3. Consider the following set of constraints:

$$\begin{aligned} C = \{c_1 &: x \neq 4; \\ c_2 &: x > 5; \\ c_3 &: x + y < 10; \\ c_4 &: z \neq 1; \\ c_5 &: y \geq 7; \} \end{aligned}$$

over domain $D = [1, 20]$ for all the variables $V = \{x, y, z\}$. It is inconsistent, as there is no solution to the CSP $\langle V, D, C \rangle$. We will apply deletion filtering to find irreducible inconsistent subset C' of C. Initially $C' = C$.

Let first $C' = C \setminus \{c_1\}$. Set C' is still inconsistent, which means we can remove constraint c_1 safely.

Suppose then $C' = \{C \setminus \{c_1, c_2\}\} = \{c_3 : x + y < 10; c_4 : z \neq 1; c_5 : y \geq 7; \}$. This set of constraint is consistent, and assignment $x = 1; y = 7; z = 0;$ is a solution to $\langle V, D, \{C \setminus \{c_1, c_2\}\} \rangle$. Then the constraint c_2 can not be removed from initial assignment C.

Following this way, after applying algorithm to all remaining constraints we may only remove c_4 as well. The resulting irreducible inconsistent set is

$$\begin{aligned} C' = \{c_2 &: x > 5; \\ c_3 &: x + y < 10; \\ c_5 &: y \geq 7; \}. \end{aligned}$$

As per paper [23], this algorithm is correct. That is, given the input set of constraints C, in a finite time it produces set C' that is irreducible and inconsistent.

Note that in general irreducibility is not required for algorithm correctness. However, we are interested in finding minimal set of constraints, as in this case the search space is reduced more.

The bottleneck of deletion filtering algorithm is that the removal of the constraint requires constraint solver to reset the search space. This might be a costly operation and we want to avoid it with other learning strategies.

Forward filtering

The improvement of the forward filtering algorithm, also [23], is to add iteratively only those constraints, which add to the inconsistency. Let T be a initially empty testing set. At each step a constraint is added to T until T becomes inconsistent. In such case, the last added constraint belongs to IIS and is added to resulting set C'. The procedure continues until resulting set C' becomes inconsistent.

The main benefit of the algorithm is that adding new constraints to the set only restricts already existing domains of constraint variables. In such case we are avoiding the problem of resetting the search space, as described above. The detailed algorithm is provided below.

Example 4.2.4. Reconsult the previous set of constraints:

$$\begin{aligned} C = \{&c_1 : x \neq 4; \\ &c_2 : x > 5; \\ &c_3 : x + y < 10; \\ &c_4 : z \neq 1; \\ &c_5 : y \geq 7; \}. \end{aligned}$$

At first iteration the constraints are added in the order c_1, c_2, c_3, c_4, c_5 until adding c_5 leads to inconsistency. This constraint c_5 is added to currently found result list $C' = \{c_5\}$. At second iteration C' already contains c_5, so the minimal inconsistent set is c_5, c_1, c_2, c_3, adding the last one to $C' = \{c_5, c_3\}$. At last, c_2 will be added to $C' = \{c_5, c_3, c_2\}$ which is finally IIS. The search space was reset 3 times in comparison to 5 times in deletion filtering, however.

In this regard, it was also shown [23] that the algorithm is correct and terminates. We will need the correctness to prove the correctness of learning function for HEX programs later.

In what is following, we will provide different heuristics to the forward filtering,. The forward filtering picks up them blindly, not taking into account the inner structure of constraint, which might be a reason for improvement.

As they are in core following forward filtering, all the correctness proofs also holding for them as well.

Backward filtering

Backward filtering ([23]) is a simple heuristic over a forward filtering on the order of adding the constraints.

Algorithm 4.2: Forward filtering

```
input  : Inconsistent set of constraints $C$
output: Irreducible inconsistent set of constraints $C'$
1  $C' \leftarrow \emptyset$;
2  while $C'$ is consistent do
3      $T \leftarrow C'$;
4      for $i \leftarrow 1$ to $n$ do
5          $T \leftarrow T \cup c_i$;
6          if $T$ is inconsistent then
7              $C' \leftarrow C' \cup c_i$;
8              break
9          end
10     end
11 end
```

Instead of adding constraints from the beginning of the inconsistent set, last elements are added. Following this approach a constraint is first tested if it was added last. In such case it is more likely that it adds to a inconsistency.

The algorithm is also still correct as only the order of picking the constraint was changed.

Connected components filtering

Connected components filtering ([23]) is an improvement over the forward filtering algorithm that takes into account the inner structure of constraints. This algorithm tries to pick up the ones which are connected to it. The connection in this case meaning sharing the same constraint variables.

A constraint c_1 is *connected* to a constraint c_2 iff their scopes intersect: $S(c_1) \cap S(c_2) \neq \emptyset$.

A modified algorithm for the connected components filtering is provided. The idea is to add those constraints that are connected to the currently found IIS and after them disconnected.

In the main loop starting at line 3, we will split the set of constraints which are connected and disconnected to the current assignment. We will first add connected constraint in the inner loop in lines 5-15, then trying to add disconnected in lines 16-24 in case there was no addition already.

As this algorithm defines only a heuristics on selecting the constraint order it is still correct.

Weighted connected components filtering

We will now provide an additional learning strategy that was developed by us and extends forward filtering. We will coin more precise extension of connected components filtering. Instead of picking up first connected constraints and then disconnected, we will do this in decreasing order of so-called weight of constraint.

Algorithm 4.3: Connected components filtering

```
input  : Inconsistent set of constraints $C$
output: Irreducible inconsistent set of constraints $C'$
1  $C' \leftarrow \emptyset$;
2  $V \leftarrow \emptyset$;
3  while $C'$ is consistent do
4      $T \leftarrow C'$;
5      for $i \leftarrow 1$ to $n$ do
6          Trying to add constraints that are connected with current assignment
7          if $S(c_i) \cap V \neq \emptyset$ then
8              $T \leftarrow T \cup \{c_i\}$;
9              if $T$ is inconsistent then
10                 $C' \leftarrow C' \cup \{c_i\}$;
11                 $V \leftarrow V \cup S(c_i)$;
12                 break
13             end
14         end
15     end
16     for $i \leftarrow 1$ to $n$ do
17         Trying to add other constraints
18         $T \leftarrow T \cup \{c_i\}$;
19         if $T$ is inconsistent then
20             $I' \leftarrow I' \cup \{c_i\}$;
21             $V \leftarrow V \cup S(c_i)$;
22         end
23     end
24 end
```

Let the *connection weight* of constraint c with respect to a set of currently assigned constraints I be a function $weight(c) : C \times I \to \mathbb{Z}^+$ defined as $weight(c, I) = |S(c) \cap S(I)|$.

Now we will modify the connected components algorithm by adding constraint c with biggest $weight(c)$ first in the inner loop of checking testing set inconsistency. The modified algorithm is provided. Let also function $sort{-}weight : C \times I \to C$ produces set C', where C' is the sorted list of the C based on the weight of constraints with respect to the set of currently assigned constraints I.

Algorithm 4.4: Weighted connected components filtering

```
input  : Inconsistent set of constraints $C$
output: Irreducible inconsistent set of constraints $C'$
$C' \leftarrow \emptyset$;
$V \leftarrow \emptyset$;
while $C'$ is consistent do
    $T \leftarrow C'$;
    $SortC \leftarrow sort{-}weight(C, T)$;
    for $i \leftarrow 1$ to $n$ do
        $c \leftarrow SortC[i]$;
        if $S(c) \cap V \neq \emptyset$ then
            $T \leftarrow T \cup \{c\}$;
            if $T$ is inconsistent then
                $C' \leftarrow I' \cup \{c\}$;
                $V \leftarrow V \cup S(c)$;
                break
            end
        end
    end
end
```

Example 4.2.5. Consider the following set of constraints:

$$\begin{aligned} C = \{&c_1 : x > 0; \\ &c_2 : y > 0; \\ &c_3 : z > 0; \\ &c_4 : x + y + z + t = 4; \\ &c_5 : x + y + z + t = 5; \} \end{aligned}$$

over the domain $D = [0..10]$. We will apply weighted connected components filtering to find IIS.

Initially, set $C' = T' = \emptyset$. At the first iteration testing set T' becomes inconsistent with addition of constraint c_5, so then $C' = \{c_5 : x + y + z + t = 5\}$.

In the following step, as all constraints are connected to c_4 then the ordinary connected components filtering will try to add in sequence c_1, c_2, c_3 and c_4 to the set $T' = \{c_5\}$ until it becomes inconsistent with addition of c_4 finally, so IIS is $C' = \{c_5, c_4\}$. However, we can save 3 propagations if we follow weighted approach. Constraint c_5 is connected with assignment $\{c_4\}$ with weight of 4, as they share all variables x, y, z and t. As such, it would be added immediately, testing set T' becomes inconsistent and so final the IIS is the same $C' = \{c_5, c_4\}$. The improved algorithm reduced the time of the propagation by more than two times in this example.

User-defined learning for constraint plugin

We can define now a learning function for the HEX programs with respect to the assignment A. We will denote $expr^A$ as the set of atoms with predicate *expr* that hold in the assignment A. Let also function $iis\_algorithm(C)$ be the result of the application of any of the algorithm(deletion, forward, backward, connected components and weighted connected components filtering) to the set of constraints C.

Recall also the functions $\theta(A)$ and $\theta(A)^{-1}$ that converts ordinary atom to a constraint and vice versa. Having this in mind the learning function Λ is defined accordingly as per its definition.

$$\Lambda_{IIS}(expr, A) = \{\theta^{-1}(iis\_algorithm(\theta(expr^A)))\}$$

We will now show the correctness of this function. Recall that learning function is correct iff all compatible sets of the logic program are solutions to the nogood that was added to the program.

Proposition 4.2.1. Learning function Λ_{IIS} is correct.

Proof. Let P be the program and A an assignment for it. Learnt nogood in such case is $\delta = \Lambda_{IIS}(\&casp[expr], A)$. We want to show that this nogood is correct; each set from compatible sets of P is an solution to δ.

Let A' be an arbitrary compatible set of P. Then A' is an answer set of the guessing program $\hat{P}$. Program $\hat{P}$ will contain a rule

$$e_{\&casp[expr]}() \lor ne_{\&casp[expr]}().$$

The first atom $e_{\&casp[expr]}()$ is always satisfied, as the original program contains the following rule:

$$\leftarrow not\ \&casp[expr, nexpr, dom]().$$

If this is the case, then the set of atoms with *expr* and *nexpr* define a CSP that is consistent. On the other hand, nogood δ defines a set of inconsistent constraints as was proven above, all the algorithms for finding IISs are correct. As a set of inconsistent constraints can not be a subset of consistent constraints, then assignment A' is a solution to δ. As A' was picked arbitrarily, all compatible sets are solution to δ and the learning function is correct. □

We have provided theoretical background on possible improvements of the constraint answer-set programming plugin: improved constraint atom translation and nogood learning strategies for the inconsistent set of constraints. The DLVHEX system provides convenient support for implementing the plugin including the above-mentioned improvements which we will explain in the following chapter.

CHAPTER 5

Implementation

In the previous chapters we provided a theoretical approach for constraint answer set programming. To summarize, we need to implement two main tasks of the plugin.

The first task is to provide a translation procedure given the program with constraint atoms to answer-set program with external atoms. The single external atom

$$\&casp[dom, expr, nexpr]().$$

is responsible for checking the consistency of constraints with an external constraint solver. The constraints are passed over $expr, nexpr$ predicate extension, the domain passed in dom predicate extension. We want to be able to interact with source of knowledge that is pluggable to our system.

The second part is implementing user-defined learning for external atoms. When the constraint solver evaluates a set of constraints to be inconsistent, a smaller subset of inconsistent constraints might contribute to adding a nogood. An answer-set solver must be capable of adding such nogoods to its evaluation procedure.

In this chapter we will explain how this task can be implemented in the framework of DLVHEX. We will use an open-source constraint solver GECODE [1]. We will then later explain the implementation of concrete parts of the plugin.

5.1 Installation and usage

A CASP plugin needs to be installed to DLVHEX system for supporting constraint answer-set programs. The plugin needs to be installed from source [2]. After obtaining the source, necessary dependencies are installed by calling *bootstrap.sh* and finally for installing the plugin itself with *make install*.

[1]http://www.gecode.org/

[2]https://github.com/hexhex/caspplugin

The plugin is installed as a shared *C++* library for the DLVHEX system. Once installed, the following command line options are available when calling **dlvhex2**.

- *--cspenable* Enables the CASP plugin.
- *--headguess* Enables guessing in head rules(as in original translation procedure $tr(P)$.
- *--csplearning=[option]* Defines learning strategy. The values for the *[option]* can be the following values.

– none	- No learning.
– deletion	- Deletion filtering learning.
– forward	- Forward filtering learning.
– backward	- Backward filtering learning.
– cc	- Connected component filtering learning.
– wcc	- Weighted connected component filtering learning

- *--help* Displays the help message.

The default command line options of the plugin is disabled CASP plugin, improved translation algorithm $tr'(P)$ and no used-defined learning.

For example, consider constraint logic program that is present in file *casp.tex*. The following command runs it with improved translation and backward learning

dlvhex2 --cspenable --csplearning=backward

5.2 Architecture overview

We are going to present an architecture of the implementation in this section. The DLVHEX system provides convenient support of writing custom plugins to the system through the dynamically loaded shared C++ libraries that contain plugin.

The following figure shows main parts of the system for the CASP plugin.

In general, the workflow of our program will pass various steps during processing. First, before actual processing with DLVHEX system we will translate program with specific layers, called CASP converter and CASP rewriter. These layers implement the translation procedure: CASP converter translates the constraint atoms to the auxiliary ordinary atoms, whereas CASP rewriter adds necessary guessing rules.

The semantics of the external $\&casp$ atoms that interacts with constraint solver GECODE are also implemented. Two use cases are realized for the constraint solver: checking consistency of the given set of constraints and finding irreducible inconsistent set in case the original set was inconsistent.

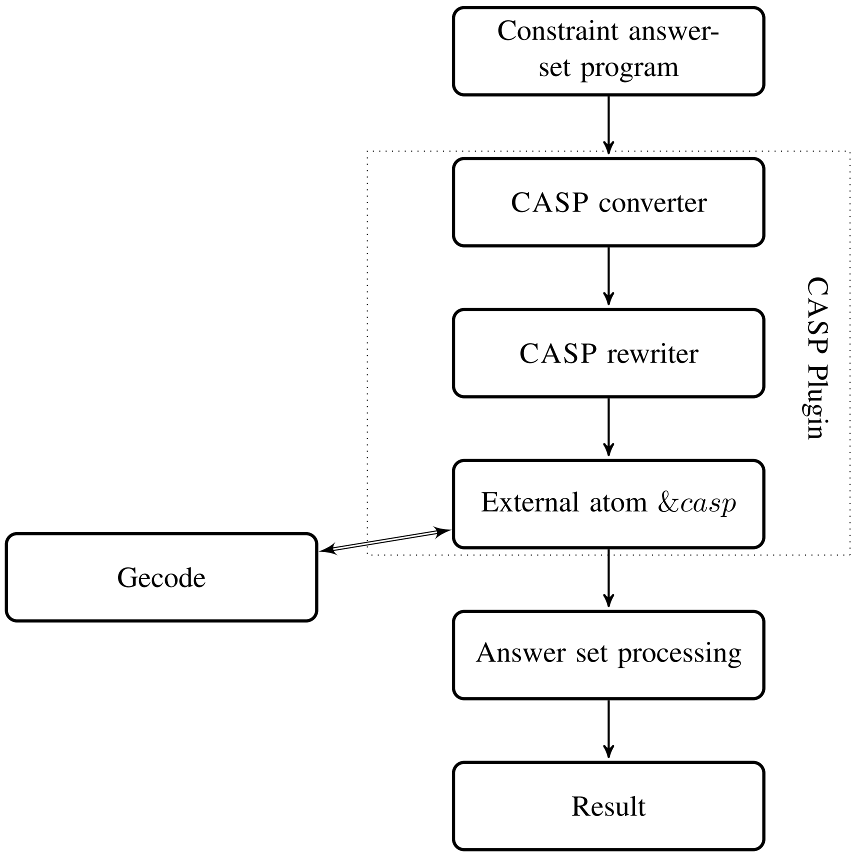


Figure 5.1: Implementation architecture

The three parts of converter, rewriter and external atom definition provide the core of the CASP plugin implementation. They are embedded in the DLVHEX system, along with the learning procedures.

Finally, the answer set is computed by the DLVHEX system in the usual manner is calling CASP plugin, when performing evaluation. An answer set is obtained as a result of the solver.

We also want to split parts of the translation and learning into smaller subtasks that are handled separately and independently. Therefore, it will be possible to change different strategies and techniques for them, as was defined in the previous sections. We now describe implementation parts in more details.

5.3 Translation implementation

We will break the translation task into the following three smaller subtasks.

- converting the constraint atoms into the representation of regular atoms by the procedure

$convert(A)$;

- adding guessing rules depending on original and modified translation. The first strategy is to add it to all occurrences of constraint atoms, the second strategy is to add them only for the atoms in the body of the rules;
- implementing the interaction with constraint solver for external atom.

The implementation of the conversion of the atom translation is done by CASP *converter*. Converter is a preprocessor for the program that is getting executed before the program is run. As such, it takes as input program text as a string and returns converted program text. The implementation of casp plugin converter takes the program text and searches for all inclusions of the constraint atoms. Note that constraint atoms and only them will have \$ symbol in their notion. As such, we might define the following procedure for converting them to ordinary atoms.

First, we need to find the constraint atom with main operator $\odot \in \{=, \neq, <, >, \leq, \geq\}$. After that we want to find out the beginning and the end of the atom in string representation by moving left and right from the main operator as long as we don't reach the delimiter for the atom. We will also keep track of ASP variables that are occurring during this step. Once the delimiters are reached, the constraint atom is quoted with " symbols and predicate *expr* is added before the constraint; tracked variables are added as parameters to the predicate.

Throughout all the conversion procedure, we need to handle special cases, such as ignoring comments, string literals, handling multi-lining and other ones are defined as per DLVHEX grammar.

Example 5.3.1. Consider following rule appearing in the text of the program:

$$r \leftarrow p(X), q(X), c(X) \ \$\!> \ 5 \ \$\!+ k, s(\text{“}\texttt{p \$< p}\text{”}), m \ \$\!< 2.$$

We have first tracked down the occurrence of $\$ >$ symbol of the constraint operator in expression $\$>$. We will start moving left until we reach the symbol , right after $q(X)$ expression and have found ASP variable X. Similarly, we have found , right before $s(“p \ \$ < p'')$ expression. As such, we will replace this substring with $expr(c(X) \ \$> \ 5 \ \$+ z, X)$.

Following similar way, another occurrence of \$ symbol is found in $\$ <$. Note that the one in the string literal “p \$< p” is ignored. As such, after finding the delimiter to the left and to the right of such expressions no variables are found and this is replaced with *expr* atom as well. Finally, the converted rule as follows:

$$\begin{aligned} r \leftarrow p(X), q(X), &expr(\texttt{c(X) \$> 5 \$+ k}, X), \\ &s(\text{“}\texttt{p \$< p}\text{”}), expr(\texttt{m \$< 2}). \end{aligned}$$

The second part of the translation is adding the guessing rules for the plugin. The DLVHEX plugin part called rewriter is a framework part gets executed after the converter. It receives the parsed set of rules based on the DLVHEX parser and produces another set of rules. This is the case for our plugin: we want to enlarge the set of the rules by adding new guessing rules.

As such, we simply iterate over all rules, looking for the rules that have atoms with *expr* predicate. Once it is found we need to add respective guessing rule.

Both converter and rewriter are added to the plugin, which is installed in the DLVHEX system. We also use different rewriter implementation depending on the command line option for the strategy of adding guessing parts for constraint atoms in the head of the program rule.

5.4 An external atom for constraint consistency

The DLVHEX system defines a mechanism for inserting the definitions of external atoms. As such, we want to define an interface of how the external atoms interact with the external solver. This interface should be independent of the actual realization of the solver. The external atom input is a simply string array representation of constraints and another string for the domain of such constraints.

Let us focus now on the GECODE specific implementation of a constraint solver. A GECODE solver requires to define constraint variables before posting the constraints for propagation.

We will first need to provide proper translation of ordinary ground atoms in the candidate model to constraints. Consider the following non-ground atom:

$$expr(\mathtt{X}\ \${>}\ (\mathtt{Y}\ \${+}\ \mathtt{X})/\mathtt{Y}, X, Y).$$

and its respective grounded version:

$$expr(\mathtt{X}\ \${>}\ (\mathtt{Y}\ \${+}\ \mathtt{X})/\mathtt{Y}, 2, 4).$$

In order to translate it to the constraint, we need to take the string representation of the constraint and replace all occurrences of answer-set variables with their actual ground values. In this example we will obtain:

$$2\ \${>}\ (4\ \${+}\ 2)/4$$

We will now must feed such constraints to the solver to check their consistency. In order to solve a constraint satisfaction problem, GECODE defines a class of search *Space*. We propagate the constraints in it with the form of linear expressions and solve the problem using *engine* : depth-first search, branch-and-bound engine. [3]

GECODE provides a very convenient way of propagation of the constraint types over a so-called *LinExpr* object defines a linear expression over our constraint variables. A part of linear expression is either constraint variable, number or another linear expression. We will then define a comparison of two linear expressions as a separate object called *Expr* contains exactly one of the main comparison constraint operators $\odot \in \{=, \neq, >, <, \geq, \leq\}$.

We will define the following syntax for a linear expression:

[3]http://www.gecode.org/doc-latest/reference/group__TaskModelSearch.html

$$\begin{aligned} LinExpr &:= c, && \text{where } c \text{ is constraint variable} \\ LinExpr &:= N, && \text{where } N \text{ is a numerical literal} \\ LinExpr &:= LinExpr \odot LinExpr, && \text{where } \odot \in \{+, -, *, /\} \\ Expr &:= LinExpr \odot LinExpr, && \text{where } \odot \in \{=, \neq, >, <, \geq, \leq\} \end{aligned}$$

We will define a separate recursive parser for such expressions. Once the input string is parsed, we can create respective *Expr* object for GECODE solver. For more information on creating relation in GECODE solver we refer to official documentation [4].

Constraint solver in general can define many different parameters for propagating and searching the solution. In the implementation of the CASP plugin, we will be reusing branch-and-bound (BAB) search engine. We will also give the plugin maximum available amount of memory and stop processing once some solution is found. This is possible as we are only interested in checking the constraint consistency, not actually all possible solutions to CSP.

5.5 Global constraints implementation

We want to provide support for global constraints in our CASP plugin. We will focus here on specific GECODE features that allow support of the given global constraint.

The case for global constraint *sum* is simpler. Whenever there is an occurrence of atom with $\$sum(p, i)$ construct, we simply want to replace it the *i-th* argument of the predicate extension p. Note that per definition of the external atom with global constraints, the predicate p would belong to the set S' and as such would be passed to the solver when the external atom would be evaluated.

For the *maximization* or *minimization* constraints we are interested in selecting only those answer sets that have the value of cost function equal to maximum or minimal value respectively. We will then implement the external atom $\&casp$ in such way that it will set the output to the empty tuple iff the set of constraints is consistent and the value of cost function is the same an maximum over all possible. The only requirement then is to find the maximum and minimum value of the expression given the constraints.

GECODE solver provides direct support *maximize* or *minimize* constraint. An internal solver engine is required to implement instead of *Space* class structure to use *MaximizableSpace*. Such space requires implementation of the virtual *cost function* of the following declaration:

virtual Gecode::IntVar cost() const;

The function returns GECODE *IntVar* objects returns numerical variable of the maximization of the expression under global constraint $\$maximize$. In case of $\$minimize$ constraint, the value of cost function should be taken with minus sign.

Such approach concludes the implementation of global constraints in our plugin.

[4]http://www.gecode.org/documentation.html

5.6 Learning implementation

We want to define learning strategies that are interchangeable between themselves. The DLVHEX system defines a simple interfacing of learning nogood by adding set of atoms compromise to nogood to a specific object called *NogoodContainer*.

As such, we will define an interface LearningProcessor that takes a pointer to a *Nogood-Container* and a set of constraints. A specific implementation will provide the learning strategy. We will provide following implementations: DeletionLearningProcessor, ForwardLearningProcessor, BackwardLearningProcessor, RangeLearningProcessor, CCLearningProcessor, WeightedCCLearningProcessor. An additional empty implementation called NoLearningProcessor is provided that simply ignores provided set of constraints and does not add anything to Nogood-Container.

One specific implementation point for learning is that search spaces should be *cloned* instead of propagating constraints once a step is performed(for example, in case of forward learning the resetting of testing set T from already assigned set S.

Once the plugin is initialized based on the command line arguments we will inject a specific implementation to the plugin.

One important aspect is that we have implemented learning for full model candidates in CASP plugin. A potential area of improvement is to add learning for the partial interpretation which can already have inconsistent subset of constraints. This set can then be immediately added as a learning nogood.

This section concludes the implementation details of the plugin. We will focus now on how the implementation performs.

CHAPTER 6

Application Scenarios and Benchmarking

In this chapter we are going to present usage of the algorithm on problems that involve both constraint and answer-set nature. We will provide couple of examples of the problems that exploit a specific behavior of CASP plugin. We also show how the plugin can be effectively integrated with other external sources of knowledge.

Benchmark Settings. We run our benchmarks single-threaded on a cluster with 24 cores with 2.3GHz each within total of 128 GB RAM. DLVHEX core version 2.1.0 was compiled using and installed with GCC version 4.6.3 using GECODE 3.7.0. Operating system is Linux 3.2.0-39-generic, distribution is Ubuntu 12.04. We used the standard configuration of DLVHEX with enabled CASP plugin with option *--csp-enable*.

6.1 Worker skill

We want to test first the translation procedure and its improvement in comparison to state-of-the-art implementation. We will also now omit learning techniques.

Consider following variation of scheduling problem that illustrates our needs. The problem and the test set is created by ourselves.

A number of tasks is required to be completed by workers. Each worker has a skill. In order to solve the task his skill should be equal or greater than some number. This number is different for each task. Each worker can solve at most 1 task. What are the possible assignments of workers to the tasks?

Note that in general whenever each worker among M workers is capable of solving each task among N, then the answer is what is known in combinatorics M-permutations of N: $\frac{N!}{(N-M)!}$.

A constraint answer-set program representing this program is provided below. Fact *task(X,Y)* represents that task X requires at least skill Y to be solved, whereas *worker(X,Y)* states that worker X has a skill Y.

% Task and worker data
$task(1,4).$
$task(2,6).$

$worker(1,7).$
$worker(2,5).$
$worker(3,4).$

% A task X can be assigned or not assigned to worker Y
$task\_assigned(X,Y) \leftarrow not\ task\_unassigned(X,Y), task(X,Z), worker(Y,T).$
$task\_unassigned(X,Y) \leftarrow not\ task\_assigned(X,Y), task(X,Z), worker(Y,T).$

% Each task should be assigned
$task\_assigned\_any(X) \leftarrow task\_assigned(X,Y).$
$\leftarrow not\ task\_assigned\_any(X), task(X,Y).$

% Task can not be assigned to more than 1 worker
$\leftarrow task\_assigned(X,Y1), task\_assigned(X,Y2), Y1\ \ != Y2.$
% Each worker can not perform more that 1 task
$\leftarrow task\_assigned(X1,Y), task\_assigned(X2,Y), X1\ \ != X2.$

% If task is assigned to worker, skill of worker should be greater or equal than skill of task
$Y \ \$>=\ \ Z \leftarrow task\_assigned(X,Y), task(X,Z), worker(Y,T).$

This program contains a single constraint atom that is presented in the head of the last rule. Based on two translation algorithms $tr(P)$ and $tr'(P)$ defined previously, the generated programs will contain or will not contain guessing rule for the head atom. As was shown before the second rule in the original translation $tr(P)$ is actually not necessary.

% If task is assigned to worker, skill of worker should be greater or equal than skill of task

$expr(\texttt{Y \$>= Z}, Y, Z) \leftarrow task\_assigned(X, Y), task(X, Z), worker(Y, T).$

$expr(\texttt{Y \$>= Z}, Y, Z) \lor nexpr(\texttt{Y \$>= Z}, Y, Z) \leftarrow$
$\quad task\_assigned(X, Y), task(X, Z), worker(Y, T).$

Whenever body of the rule is satisfied, atom $nexpr(\texttt{Y \$>= Z}, Y, Z)$ can not be satisfied.

We will split our instances into two classes based on number of tasks and workers: small($\leq$ 12) and big($\leq$ 24). As the number of answer sets might be big, as stated above, we also want to make another split in classes. In first category a small number of workers capable of solving each task, leading to small number of answer sets. The second category has big number of workers capable of solving a task and the number of answer sets is expected to be bigger. The guessing rule for assigning task is executed more frequently.

The results are provided below. We have performed a benchmark run on a set of tests with state-of-the art solver Clingcon 2.0.3 [1], CASP plugin with modified translation procedure omitting guessing in heads and denoted as $tr'(P)$, CASP plugin with original translation procedure denoted as $tr(P)$. Test sets are divided from small denoted as (1) to big (2) and can be found in [2].

As we are interested in all possible assignments of worker to tasks, we want to compute all possible answer sets. For both Clingcon and CASP plugin we need to specify command line option *--number=0*

The timeout was set to 600 seconds that is expression *(1)* denotes that timeout has occurred, whereas *(0)* denotes the program terminated without being timed out.

In all the configurations, learning was disabled.

We are seeing that CASP plugin implementation is not performing as state-of-the art, yet is capable to solve one hard instance of the problem. One reason of this problem is the theory propagation. The external constraint consistency atom is evaluated over full interpretations, not partial and therefore the inconsistent set is found much later.

Depending on the size of the problem the removal of guessing in the head of the program speeds up the execution time. This fact is consistent with the theoretical background provided in the previous section. This improvement is visible more in bigger instances, for example *7,9,10,13*. It also shows that guessing improvement can be viable in some cases.

[1]http://sourceforge.net/projects/potassco/files/clingcon/2.0.3/

[2]https://github.com/hexhex/caspplugin/tree/master/benchmarks/workerskill

Table 6.1: Worker skill benchmarking results

N	*Clingcon*	$tr'(P)$	$tr(P)$
1 (1)	0.00 (0)	0.08 (0)	0.09 (0)
2 (1)	0.00 (0)	0.08 (0)	0.09 (0)
3 (1)	0.00 (0)	0.09 (0)	0.11 (0)
4 (1)	0.01 (0)	0.40 (0)	0.42 (0)
5 (1)	0.01 (0)	0.22 (0)	0.27 (0)
6 (1)	0.05 (0)	3.85 (0)	3.96 (0)
7 (1)	0.16 (0)	13.60 (0)	15.08 (0)
8 (1)	0.05 (0)	4.01 (0)	4.24 (0)
9 (1)	3.06 (0)	120.45 (0)	133.69 (0)
10 (1)	0.48 (0)	35.97 (0)	40.20 (0)
11 (2)	18.26 (0)	535.00 (0)	553.20 (0)
12 (2)	33.02 (0)	600.00 (1)	600.00 (1)
13 (2)	48.27 (0)	600.00 (1)	600.00 (1)
14 (2)	360.50 (0)	600.00 (1)	600.00 (1)
15 (2)	91.18 (0)	600.00 (1)	600.00 (1)
16 (2)	600.00 (1)	600.00 (1)	600.00 (1)
17 (2)	600.00 (1)	600.00 (1)	600.00 (1)
18 (2)	600.00 (1)	600.00 (1)	600.00 (1)
19 (2)	600.00 (1)	600.00 (1)	600.00 (1)

6.2 Packing

The following problem is part of ASP solver competition 2011 [3]. We will modify the official encoding for the constraint answer-set programs. The problem name is `Packing` and defined as following:

Given a rectangular area of a known dimension and a set of squares, each of which has a known dimension, the problem is to pack all the squares into the rectangular area such that no two squares overlap each other. There can be wasted spaces in the rectangular area.

The rectangular area forms a coordinate plane where the left upper corner of the area is the origin, the top edge is the x-axis, and the left edge is the y-axis.

We will encode the problem in the following manner. We will try to position square with left upper corner in (X, Y) coordinate defined atoms with $posx$ and $posy$ predicates respectively. Whenever a square with size S is placed, its edge is now aligned along edges $X + S$, $Y + S$. If in both axes the edges intersect, the alignment is improper.

[3] https://www.mat.unical.it/aspcomp2011/FinalProblemDescriptions/Packing

$posx(X)\$ <= W\$ - S \leftarrow square(X,S), area(W,H).$
$posy(X)\$ <= H\$ - S \leftarrow square(X,S), area(W,H).$

$intersectx(X1,X2) \leftarrow square(X1,S1), square(X2,S2), posx(X1)\$ <= posx(X2),$
$\quad posx(X1)\$ + S1\$ > posx(X2), X1! = X2.$
$intersectx(X1,X2) \leftarrow square(X1,S1), square(X2,S2), posx(X1)\$ < posx(X2)\$ + S2,$
$\quad posx(X1)\$ + S1\$ > posx(X2)\$ + S2, X1! = X2.$

$intersecty(X1,X2) \leftarrow square(X1,S1), square(X2,S2), posy(X1)\$ <= posy(X2),$
$\quad posy(X1)\$ + S1\$ > posy(X2), X1! = X2.$
$intersecty(X1,X2) \leftarrow square(X1,S1), square(X2,S2), posy(X1)\$ < posy(X2)\$ + S2,$
$\quad posy(X1)\$ + S1\$ > posy(X2)\$ + S2, X1! = X2.$

$intersectx(X1,X2) \leftarrow intersectx(X2,X1), square(X1,S1), square(X2,S2).$
$intersecty(X1,X2) \leftarrow intersecty(X2,X1), square(X1,S1), square(X2,S2).$

$\leftarrow intersectx(X1,X2), intersecty(X1,X2).$

We will split problem instances to small($\leq$ 15) and big($\leq$ 40) based on the number of squares in the input data.

The results are provided below. We will use the following shortcuts provided in the following table.

Cn - Clingcon with no learning.

Cf - Clingcon with forward learning.

Cb - Clingcon with backward learning.

Ccc - Clingcon with with connected components learning.

Dn - CASP plugin DLVHEX with no learning.

Dd - CASP plugin DLVHEX with deletion learning.

Df - CASP plugin DLVHEX with forward learning.

Db - CASP plugin DLVHEX with backward learning.

Dcc - CASP plugin DLVHEX with with connected components learning.

Dwcc - CASP plugin DLVHEX with with weighted connected components learning.

We also always used improved translation that omits guessing in constraint heads of the rules.

Table 6.2: Packing benchmarking results

N	*Cn*	*Cf*	*Cb*	*Ccc*	*Dn*	*Dd*	*Df*	*Db*	*Dcc*	*Dwcc*
1 (1)	0.01 (0)	0.01 (0)	0.01 (0)	0.01 (0)	250.13 (0)	7.39 (0)	1.41 (0)	0.68 (0)	0.71 (0)	0.70 (0)
2 (1)	0.01 (0)	0.01 (0)	0.01 (0)	0.01 (0)	600.00 (1)	22.79 (0)	1.44 (0)	1.96 (0)	2.08 (0)	2.02 (0)
3 (1)	0.02 (0)	0.02 (0)	0.02 (0)	0.02 (0)	600.00 (1)	47.02 (0)	3.71 (0)	2.12 (0)	2.43 (0)	2.47 (0)
4 (1)	0.01 (0)	0.03 (0)	0.03 (0)	0.02 (0)	600.00 (1)	600.00 (1)	22.88 (0)	56.89 (0)	44.43 (0)	42.12 (0)
5 (1)	0.02 (0)	0.04 (0)	0.08 (0)	0.09 (0)	600.00 (1)	600.00 (1)	19.52 (0)	40.94 (0)	45.15 (0)	47.14 (0)
6 (1)	0.04 (0)	0.10 (0)	0.05 (0)	0.07 (0)	600.00 (1)	600.00 (1)	94.68 (0)	111.13 (0)	108.43 (0)	109.53 (0)
7 (1)	0.14 (0)	0.45 (0)	0.68 (0)	0.98 (0)	600.00 (1)	600.00 (1)	242.76 (0)	600.00 (1)	512.63 (0)	600.00 (1)
8 (1)	0.12 (0)	0.28 (0)	0.18 (0)	0.23 (0)	600.00 (1)	600.00 (1)	223.99 (0)	480.65 (0)	470.23 (0)	439.13 (0)
9 (1)	600.00 (1)	0.73 (0)	0.84 (0)	0.77 (0)	600.00 (1)	600.00 (1)	245.08 (0)	600.00 (1)	600.00 (1)	600.00 (1)
10 (1)	0.04 (0)	0.09 (0)	0.22 (0)	0.29 (0)	600.00 (1)	600.00 (1)	92.84 (0)	120.43 (0)	104.23 (0)	107.66 (0)
11 (2)	600.00 (1)	0.79 (0)	0.92 (0)	0.72 (0)	600.00 (1)	600.00 (1)	543.53 (0)	600.00 (1)	600.00 (1)	600.00 (1)
12 (2)	600.00 (1)	1.12 (0)	1.89 (0)	1.77 (0)	600.00 (1)	600.00 (1)	600.00 (1)	600.00 (1)	600.00 (1)	600.00 (1)
13 (2)	600.00 (1)	4.53 (0)	4.10 (0)	8.94 (0)	600.00 (1)	600.00 (1)	600.00 (1)	600.00 (1)	600.00 (1)	600.00 (1)
14 (2)	600.00 (1)	6.17 (0)	5.59 (0)	13.53 (0)	600.00 (1)	600.00 (1)	600.00 (1)	600.00 (1)	600.00 (1)	600.00 (1)
15 (2)	600.00 (1)	44.65 (0)	40.17 (0)	67.80 (0)	600.00 (1)	600.00 (1)	600.00 (1)	600.00 (1)	600.00 (1)	600.00 (1)
16 (2)	600.00 (1)	134.46 (0)	139.60 (0)	144.39 (0)	600.00 (1)	600.00 (1)	600.00 (1)	600.00 (1)	600.00 (1)	600.00 (1)
17 (2)	600.00 (1)	341.01 (0)	323.53 (0)	600.00 (1)	600.00 (1)	600.00 (1)	600.00 (1)	600.00 (1)	600.00 (1)	600.00 (1)
18 (2)	600.00 (1)	600.00 (1)	600.00 (1)	600.00 (1)	600.00 (1)	600.00 (1)	600.00 (1)	600.00 (1)	600.00 (1)	600.00 (1)

As was seeing before, in general CASP plugin in general performs worse than Clingcon system having the same set of parameters, as the theory propagation is currently not implemented the plugin. As such, we will focus on comparison of learning techniques for both systems.

In general, learning drastically improves speed execution for both Clingcon and DLVHEX with CASP plugin. The latter system without learning was able to solve only a single first case. Deletion learning performs worst among others (it is also not possible to run Clingcon with this type of learning) and is possible to solve only cases *1-8,10*. An instance 9 was also a hard instance, even though the number of rectangles was low. The solutions timed out because of the big area and many possibilities of rectangle alignment. Based on the running time forward learning is the next option along all the cases with backward, connected components and weighted connected components performing best. Particularly, in some cases *(1,2,4,8)* weighted improvement was better than straight connected components learning, whereas in others it performed worse, like in test case 7 where improvement timed out. The better performance of weighted approach can be explained by more clever pick-up of the constraints. The bad performance of this approach is associated with high cost of the operation for sorting the constraints based on their weight.

To summarize, in general learning is an important optimization technique that reduces running time significantly. Several learning algorithms show clear advantage over the other ones and are preferable for this problem.

6.3 Integration with different sources of knowledge

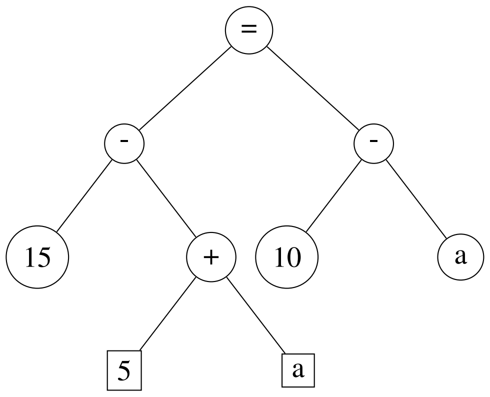
The DLVHEX system provides a various support of integration with different sources of knowledge. Consequently, constraint answer-set plugin can be used in a powerful way with combination of some of them. The following problem is an extension of the ontology consistency problem as was defined in [2].

Consider an arithmetic expression given in a RDF *form. An expression can be either a number, variable or an operation between exactly two expressions(let only sum and difference be supported). Finally, any two expressions can be equal.*

We want to check whether the expression defines a proper consistent expression. By this we will first mean that the expression is well-defined: whenever we have a sum, there are exactly two distinct summands that add up to the sum(similar case for difference, equality). Also, we want to check that expression is consistent that is we can find some assignment to the variables that all expressions are arithmetically correct.

The solution to the problem should contain an answer set having fact consistent in case the consistency is provided or no answer sets in other case.

Example 6.3.1. Suppose the following data is present in RDF format. We will represent it graphically for better understanding.



This RDF file defines an expression $15 - (5 + a) = 10 - a$. This expression is well defined that is, it has proper syntax and defines a correct arithmetic equality regardless of a concrete variable assignment.

In order to solve this problem, we will reuse two plugins of DLVHEX: the RDF plugin and the constraint answer-set plugin. The first plugin queries the input data from RDF source. We will then check whether this data is correct with answer-set constraints. Finally, expression consistency is checked with CASP plugin.

RDF plugin[4] of DLVHEX defines a single external atom

$$\&rdf[I](X, Y, Z).$$

where I - is the input path(file or web address), $\langle X, Y, Z\rangle$ represents RDF triple of subject-predicate-object, where X is a subject, Y is a predicate and Z is an object.

We will encode the program as following:

% Initial data
$\$dom(0..100).$
$file(\text{“}data.rdf''').$

% Expression correctness
% The equality should contain exactly two elements
$\leftarrow filename(F), \&rdf[F](X, \texttt{equality}, Y1), \&rdf[F](X, \texttt{equality}, Y2),$
$\quad \&rdf[F](X, \texttt{equality}, Y3), Y1 \mathrel{!=} Y2, Y1 \mathrel{!=} Y3, Y2 \mathrel{!=} Y3.$
$equality\_at\_least\_two(X) \leftarrow filename(F), \&rdf[F](X, \texttt{equality}, Y1),$
$\quad \&rdf[F](X, \texttt{equality}, Y2), Y1!! = Y2$
$\leftarrow filename(F), \&rdf[F](X, \texttt{equality}, Y), not\ equality\_at\_least\_two(X).$

% The sum should contain exactly two elements
$\leftarrow filename(F), \&rdf[F](X, \texttt{sum}, Y1), \&rdf[F](X, \texttt{sum}, Y2),$
$\quad \&rdf[F](X, \texttt{sum}, Y3), Y1 \mathrel{!=} Y2, Y1 \mathrel{!=} Y3, Y2 \mathrel{!=} Y3.$
$sum\_at\_least\_two(X) \leftarrow filename(F), \&rdf[F](X, \texttt{sum}, Y1),$
$\quad \&rdf[F](X, \texttt{sum}, Y2), Y1!! = Y2$
$\leftarrow filename(F), \&rdf[F](X, \texttt{sum}, Y), not\ sum\_at\_least\_two(X).$

% The difference should contain exactly two elements
$\leftarrow filename(F), \&rdf[F](X, \texttt{difference}, Y1), \&rdf[F](X, \texttt{difference}, Y2),$
$\quad \&rdf[F](X, \texttt{difference}, Y3), Y1 \mathrel{!=} Y2, Y1 \mathrel{!=} Y3, Y2 \mathrel{!=} Y3.$
$difference\_at\_least\_two(X) \leftarrow filename(F), \&rdf[F](X, \texttt{difference}, Y1),$
$\quad \&rdf[F](X, \texttt{difference}, Y2), Y1 \mathrel{!=} Y2$
$\leftarrow filename(F), \&rdf[F](X, \texttt{difference}, Y), not\ difference\_at\_least\_two(X).$

[4]http://www.kr.tuwien.ac.at/research/systems/dlvhex/rdfplugin.html

% Expression consistency
$X \; \$= Y \; \$+ \; Z \leftarrow filename(F), \&rdf[F](X, \texttt{sum}, Y),$
$\quad \&rdf[F](X, \texttt{sum}, Z).$
$X \; \$= Y \; \$- \; Z \leftarrow filename(F), \&rdf[F](X, \texttt{difference}, Y),$
$\quad \&rdf[F](X, \texttt{difference}, Z).$
$Y \; \$= Z \leftarrow filename(F), \&rdf[F](X, \texttt{equality}, Y),$
$\quad \&rdf[F](X, \texttt{equality}, Z).$

% We will add consistency fact to answer set, if it will be found $consistent.$

Given the expression definition in RDF we will first define whether it is correct given first 7 rules. Notably, we first need to assure that there is exactly on equality in the expression. Among all the later subexpressions, we need to ensure that binary operations are defined properly. That is, we add a constraint forbidding an expression containing 3 subexpressions. We then require to prove that each expression contains at least two subexpressions.

Later, we will check its correctness with CASP plugin with last 3 rules. Once an expression holds, we will define a constraint atom appearing in the head of the rule. The translated set of constraint will be evaluated with the external constraint solver.

Finally, if the expression is not well formed or not consistent there will be no answer set, as either correctness is not satisfied, or rule with external atom $\&casp[dom, expr, nexpr]().$ is not satisfied.

We will now show the obtained results based on auto-generated test set. As for the learning option, we only used backward filtering with improved translation(denoted as *Cb*) for the reason explained below.

Table 6.3: Expression correctness benchmarking results

N	$tr(P)$	$tr'(P)$	Cb
1 (1)	0.00 (0)	0.00 (0)	0.00 (0)
2 (1)	0.00 (0)	0.01 (0)	0.01 (0)
3 (1)	0.02 (0)	0.02 (0)	0.03 (0)
4 (1)	0.01 (0)	0.02 (0)	0.03 (0)
5 (1)	4.23 (0)	3.87 (0)	4.01 (0)
6 (1)	32.51 (0)	27.66 (0)	28.93 (0)
7 (1)	153.24 (0)	144.78 (0)	146.90 (0)
8 (1)	155.76 (0)	144.23 (0)	147.55 (0)

In general we are seeing that improved translation performs slightly better as it was in the worker skill problem. The running time is mostly used in ordinary answer set solving, as the constraint part for this problem is very small - only checking consistency of the expressions once.

The learning also does not benefit at all for this particular problem. This is the case because a check for verifying constraint consistency is performed only once. It is also the reason why only a single learning strategy was evaluated. In case the set is consistent, no nogood can be added. In this case the set is consistent, one possible nogood is added but it is not reused anymore. A slight time edge without learning is explained by unnecessary computation of finding single irreducible inconsistent set.

This chapter concludes the practical implementation and evaluation of the thesis work. We have picked up several problems, demonstrated encoding application of the solver and optimizations, compared it with other state-of-the-art solver and provide results description. We will now finally summarize the thesis results and state possible future work in the following section.

CHAPTER 7

Conclusions

Task definition

Many declarative problems require both constraint and answer-set programming techniques in order to be solved effectively. Variations of scheduling, assigning, graph problems, consistency checking are examples of such problems. The user should be able to encode the problem in a natural manner that describe the essential of the problem. This way the user would be focused on the description of the problem rather than actually solving, which forms the basis of the declarative programming. Based on such requirements a field of constraint answer-set programming has emerged. There are already existing systems that support this paradigm.

It is highly desirable to provide a framework such that the problems can be solved effectively. Additionally, an application of global constraints is also highly effective when encoding problems. The global constraints have a list of constraints of an arbitrary length as an argument.

Finally, it is necessary to reuse best techniques in both answer-set and constraint programming in order to achieve necessary performance requirements and solve the problems efficiently. We want to be able to solve hard problems that are already known and hard to tackle using ordinary approaches.

Solution

The developed in the thesis constraint answer-set plugin provided an instrument to solve abovementioned type of problems. We started with a *translation* procedure, that converts the constraint answer-set programs into HEX programs with external atom that interacts with external constraint solver. We also proved the correctness of such translation procedure in chapter 3. We also showed how several global constraints can be implemented for such case.

We then defined improvement the over direct translation, that reduces the size of the resulting program in chapter 4. We firstly showed that we can omit adding guessing rules when head contains head with constraint atom and show the equivalence to the standard translation.

We also provided a theoretical background on *conflict-driven backtracking learning* in answer-set programming, specifically *user-defined learning*. An external atom for checking constraint consistency is evaluated to false if the set of constraints in model candidate is not satisfied. We are interested in finding irreducible inconsistent set of constraints in this case as we can add nogood to the program. We presented several algorithms for finding such sets. We also showed that learning functions based on such algorithms are correct. This work is provided in chapter 4.

We explained the implementation details for DLVHEX system in chapter 5. A guide to installation, configuration and usage is provided. We split the translation and learning task into separate smaller subtasks with interchangeable implementation. We defined the main components of the application, namely CASP converter, rewriter and external atom. An implementation of semantics of external atom that checks constraint consistency is reusing external GECODE constraint solver was also explained. We provided explanation of such interaction and necessary algorithms and structures reused in the implementation. Last, we developed CASP plugin having required features.

We finally defined a set of problems, for which a CASP plugin can be applied in chapter 6. We started with the illustration of benefit of omitting guessing in heads with constraint atoms as per modified translation procedure. An example problem is a variation of scheduling problem. We then provided another benchmarking problem that was taken from ASP competition 2011, that is a geometry problem. The results included the explanation how learning techniques can reduce search space when evaluating the program. It also provides comparison with state-of-the-art constraint answer set solver.

This chapter was concluded with an example of how the DLVHEX system provides support of integrating the problem with various external sources of knowledge, that contains RDF data. An additional application may arise with the support of the new plugins for the external sources. Finally, we provided an analysis of each of the obtained results and the reasoning behind them as well.

Extensions and future work

The constraint answer-set plugin has a room for improvement. The plugin currently provides support of the constraint answer-set programming by checking the constraint consistency with the usage of external atoms and provides various techniques for finding answer efficiently.

A natural extension to this will be the addition of the constraint variables assignment in the answer-sets. This would require changing the definition of external atom 3.1 to add the external parameter that would define the extension of constraint atom predicates, that are the solution to the CSP. Consequently, this would require revision of the improvements techniques defined in chapter 4.

Additionally, used-defined learning for constraint atoms in chapter 4 can be improved. As of now, the learning is performed over full model candidates, however inconsistency may be found even when partial interpretation is available. Fixing this issue of theory propagation should significant improvement in the benchmark instances.

Currently only a very few types of global constraints are supported. A direct way would be to provide a way for supporting more of such constraints, especially taking into consideration that they are mostly implemented in the constraint solvers. The only requirement in such case is the integration in the constraint answer-set language and the plugin itself.

Finally, learning algorithm that reuses the search of irreducible inconsistent set is limited as of now. One way would be implementing more recent techniques, that are described in RRelated Workßection 1.2. A very recent approach can be found in [32]. Another way of solving this task can be integrating already existing libraries for finding such sets, like MINOS.

Bibliography

[1] Krzysztof R. Apt. *Principles of constraint programming*. Cambridge University Press, 2003.

[2] Kenneth Baclawski, Mieczyslaw M Kokar, Richard Waldinger, and Paul A Kogut. Consistency checking of semantic web ontologies. In Ian Horrocks and James Hendler, editors, *First International Semantic Web Conference (ISWC'2002), Sardinia, Italy*, volume 2342 of LCNS, pages 454–459. Springer Berlin Heidelberg, June 2002.

[3] James Bailey, Thomas Manoukian, and Kotagiri Ramamohanarao. A fast algorithm for computing hypergraph transversals and its application in mining emerging patterns. In *Proceedings of the Third IEEE International Conference on Data Mining (ICDM'03), Melbourne, Florida, USA*, pages 485–488. IEEE Computer Society, November 2003.

[4] James Bailey and Peter J Stuckey. Discovery of minimal unsatisfiable subsets of constraints using hitting set dualization. In Manuel V. Hermenegildo and Daniel Cabeza, editors, *7th International Symposium, Practical Aspects of Declarative Languages (PADL'2005), Long Beach, California, USA*, volume 3350 of LCNS, pages 174–186. Springer, January 2005.

[5] Marcello Balduccini. Representing constraint satisfaction problems in answer set programming. In *Proceedings of Workshop on Answer Set Programming and Other Computing Paradigms (ASPOC'09), Pasadena, California,USA*, July 2009.

[6] Sabrina Baselice, Piero A Bonatti, and Michael Gelfond. Towards an integration of answer set and constraint solving. In Maurizio Gabbrielli and Gopal Gupta, editors, *21st International Conference on Logic Programming (ICLP'2005), Sitges, Spain*, volume 3668 of LCNS, pages 52–66. Springer, October 2005.

[7] Giampaolo Bella and Stefano Bistarelli. Soft constraints for security protocol analysis: Confidentiality. In Venkatraman Ramakrishnan, editor, *Third International Symposium, Practical Aspects of Declarative Languages (PADL'2001), Las Vegas, Nevada, USA*, volume 1990, pages 108–122. Springer, March 2001.

[8] Frédéric Benhamou, Narendra Jussien, and Barry A O'Sullivan. *Trends in Constraint Programming*. Wiley, May 2010.

[9] Renato Bruni and Antonio Sassano. Restoring satisfiability or maintaining unsatisfiability by finding small unsatisfiable subformulae. *Electronic Notes in Discrete Mathematics*, 9:162–173, 2001.

[10] Francesco Calimeri, Giovambattista Ianni, and Francesco Ricca. The third open answer set programming competition. *Theory and Practice of Logic Programming*, FirstView article:pp.1–19, September 2013.

[11] Alberto Caprara, Filippo Focacci, Evelina Lamma, Paola Mello, Michela Milano, Paolo Toth, and Daniele Vigo. Integrating constraint logic programming and operations research techniques for the crew rostering problem. *Software: Practice and Experience*, 28(1):49–76, May 1998.

[12] Paolo Carraresi, Giorgio Gallo, and Gabriella Rago. A hypergraph model for constraint logic programming and applications to bus drivers' scheduling. *Annals of Mathematics and Artificial Intelligence*, 8(3-4):247–270, 1993.

[13] John Chinneck. Locating minimal infeasible constraints sets in linear programs. *ORSA Journal On Computing*, 3:157–168, 1991.

[14] John Chinneck. MINOS(IIS): Infeasibility Analysis Using MINOS. *Computers and Operations Research*, 21(1):1–9, 1991.

[15] Ricardo Corin and Sandro Etalle. An improved constraint-based system for the verification of security protocols. In Manuel V. Hermenegildo and Germán Puebla, editors, *9th International Symposium in Static Analysis (SAS'2002) Madrid, Spain*, pages 326–341. Springer, September 2002.

[16] Alessandro Dal Palù, Agostino Dovier, and Federico Fogolari. Constraint logic programming approach to protein structure prediction. *BioMed Central Journal*, 5(186):1–12, 2004.

[17] Thomas Eiter, Michael Fink, Thomas Krennwallner, and Christoph Redl. Conflict-driven ASP Solving with External Sources. *Theory and Practice of Logic Programming*, 12(4):659–679, 2012.

[18] Thomas Eiter, Giovambattista Ianni, Roman Schindlauer, and Hans Tompits. A Uniform Integration of Higher-Order Reasoning and External Evaluations in Answer Set Programming. In *Proceedings of the 19th International Joint Conference on Artificial intelligence (IJCAI'2005), Edinburgh, Scotland*, pages 90–96, August 2005.

[19] Wolfgang Faber, Gerald Pfeifer, and Nicola Leone. Semantics and complexity of recursive aggregates in answer set programming. *Artificial Intelligence*, 175(1):278–298, 2011.

[20] Thom Frühwirth, Alexander Herold, Volker Küchenhoff, Thierry Le Provost, Pierre Lim, Eric Monfroy, and Mark Wallace. *Constraint Logic Programming. An Informal Introduction. Technical Report ECRC-93-5, European Computer-Industry Research Center (ECRC).* ECRC, 1993.

[21] Marco Gavanelli and Francesca Rossi. Constraint logic programming. In *A 25-year perspective on logic programming*, pages 64–86. Springer, 2010.

[22] Martin Gebser, Benjamin Kaufmann, Roland Kaminski, Max Ostrowski, Torsten Schaub, and Marius Schneider. Potassco: The potsdam answer set solving collection. *AI Communication*, 24(2):107–124, April 2011.

[23] Martin Gebser, Max Ostrowski, and Torsten Schaub. Constraint answer set solving. In Patricia M. Hill and David S. Warren, editors, *25th International Conference on Logic Programming (ICLP'2009), Pasadena, California, USA*, volume 5649 of LCNS, pages 235–249. Springer Berlin Heidelberg, July 2009.

[24] Martin Gebser, Torsten Schaub, and Sven Thiele. Gringo: A new grounder for answer set programming. In Chitta Baral, Gerhard Brewka, and John Schlipf, editors, *9th International Conference in Logic Programming and Nonmonotonic Reasoning (LPNMR'07), Tempe, Arizona, USA*, volume 4483, pages 266–271. Springer, May 2007.

[25] Martin Gebser, Torsten Schaub, Sven Thiele, Björn Usadel, and Philippe Veber. Detecting inconsistencies in large biological networks with answer set programming. In Maria Garcia de la Banda and Enrico Pontelli, editors, *24th International Conference in Logic Programming (ICLP'2008), Udine, Italy*, volume 5366 of LCNS, pages 130–144. Springer, December 2008.

[26] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Proceedings of the Fifth International Conference and Symposium (ICLP/SLP'88), Seattle, Washington*, volume 1, pages 1070–1080, August 1988.

[27] Ian P Gent, Christopher Jefferson, and Ian Miguel. Minion: A fast scalable constraint solver. In Loris Penserini, Pavlos Peppas, and Anna Perini, editors, *17th European Conference on Artificial Intelligence (ECAI'2006), Riva del Garda, Italy*, volume 141, pages 98–102, August 2006.

[28] Enrico Giunchiglia, Yuliya Lierler, and Marco Maratea. Answer set programming based on propositional satisfiability. *Journal of Automated Reasoning*, 36(4):345–377, April 2006.

[29] John Gleeson and Jennifer Ryan. Identifying minimally infeasible subsystems of inequalities. *ORSA Journal on Computing*, 2(1):61–63, 1990.

[30] Harvey J Greenberg and Frederic H Murphy. Approaches to diagnosing infeasible linear programs. *ORSA Journal on Computing*, 3(3):253–261, 1991.

[31] Pascal Van Hentenryck and Laurent Michel. *Constraint-based local search*. The MIT Press, August 2009.

[32] Jun Hu, Philippe Galinier, and Alexandre Caminada. Yet another breakout inspired infeasible subset detection in constraint satisfaction problem. In Hamid R. Arabnia, David de la

Fuente, Elena B. Kozerenko, and Jose A. Olivas, editors, *15th International Conference on Artificial Intelligence (ICAI'2011), Las Vegas, Nevada, USA*, volume 3213 of LCNS, pages 56–72. CSREA Press, July 2011.

[33] Francois Laburthe. Choco: implementing a CP kernel. In *Techniques for Implementing Constraint programming Systems, a post-conference workshop of CP'00, Singapoure*, volume 55, pages 71–85, 2000.

[34] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The DLV system for knowledge representation and reasoning. *ACM Transactions on Computational Logic*, 7(3):499–562, 2006.

[35] Guohua Liu, Tomi Janhunen, and Ilkka Niemelae. Answer set programming via mixed integer programming. In *Thirteenth International Conference on the Principles on Knowledge Representation and Reasoning (KR'2012), Rome, Italy*, pages 3232–3242. AAAI Press, June 2009.

[36] Zoran Majkic. Constraint logic programming and logic modality for event's valid-time approximation. In Bhanu Prasad, editor, *Second International Indian Conference on Artificial Intelligence (IICAI'2005), Puna, India*, volume 2, pages 1258–1273, December 2005.

[37] Viviana Mascardi and Emanuela Merelli. Agent-oriented and constraint technologies for distributed transaction management. In *Third International Symposium on Intelligent Industrial Automation and Soft Computing (IIA/SOCO'99), Rome, Italy*, pages 24–56. ICSC Academic Press, June 1999.

[38] Veena S. Mellarkod, Michael Gelfond, and Yuanlin Zhang. Integrating answer set programming and constraint logic programming. *Annals of Mathematics and Artificial Intelligence*, 53,1–4:251–287, 2008.

[39] Fitting Melvin. Fixpoint semantics for logic programming - a survey. *Theoretical Computer Science*, 278(1–2):25–51, 2006.

[40] Nicholas Nethercote, Peter J Stuckey, Ralph Becket, Sebastian Brand, Gregory J Duck, and Guido Tack. Minizinc: Towards a standard CP modelling language. In Christian Bessière, editor, *13th International Conference in Principles and Practice of Constraint Programming (CP'2007), Providence, Rhode Island, USA*, volume 4741 of LCNS, pages 529–543. Springer, September 2007.

[41] Ilkka Niemelä and Patrik Simons. Smodels - an implementation of the stable model and well-founded semantics for normal logic programs. In Jürgen Dix, Ulrich Furbach, and Anil Nerode, editors, *4th International Conference in Logic Programming and Nonmonotonic Reasoning (LPNMR'1997), Dagstuhl Castle, Germany*, volume 1265 of LCNS, pages 420–429. Springer, July 1997.

[42] John Alan Robinson. Logic and logic programming. *Communications of the ACM*, 35(3):40–65, 1992.

[43] Andrea Schaerf. Scheduling sport tournaments using constraint logic programming. *Constraints*, 4(1):43–65, 1999.

[44] Christian Schulte. *Programming Constraint Services*. Doctoral dissertation, Universität des Saarlandes, Naturwissenschaftlich-Technische Fakultät I, Fachrichtung Informatik, Saarbrücken, Germany, 2000.

[45] Christian Schulte and Guido Tack. Implementing efficient propagation control. In *Third workshop on techniques for implementing constraint programming systems (TRICS'2010), Saint Andrews, Scotland*, volume 2. Springer, September 2010.

[46] Marc van Dongen, Christophe Lecoutre, and Olivier Roussel. Second international csp solver competition. Technical report, Twelfth International Conference on Principles and Practice of Constraint Programming (CP'08), September 2006.

[47] Joyce Van Loon. Irreducibly inconsistent systems of linear inequalities. *European Journal of Operational Research*, 8(3):283–288, 1981.

[48] Johan Wittocx, Maarten Marien, and Marc Denecker. The IDP System: a model expansion system for an extension of classical logic. In Mark Denecker, editor, *Proceedings of the 2nd Workshop on Logic and Search (LASH'2008), Leuven, Belgium*, pages 153–165. ACCO, 2008.