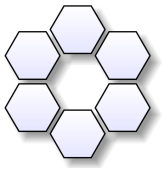

DLVHEX

VIENNA UNIVERSITY OF TECHNOLOGY

User Guide
DLVHEX 2.X

INFSYS RESEARCH REPORT 1843-15-05
May 18, 2020

Thomas Eiter[1]

Mustafa Mehuljic[2]

Christoph Redl[1]

Peter Schüller[2]

[1]Institut für Informationssysteme,
Abteilung Wissensbasierte Systems,
Technische Universität Wien, Austria
{eiter,redl}@kr.tuwien.ac.at

[2]Department of Computer Engineering,
Faculty of Engineering,
Marmara University, Turkey
mehuljic.mustafa@gmail.com
peter.schuller@marmara.edu.tr

Abstract

This document provides a user guide for the Answer Set Programming (ASP) system called DLVHEX. ASP is a declarative problem solving paradigm, rooted in logic programming and nonmonotonic reasoning, which has been gaining increasing attention during the last years. The DLVHEX system is a reasoner for computing the models of so-called HEX-programs, which are an extension of *answer-set programs* towards integration of *external computation sources*. This guide aims at explaining the syntax of HEX-programs and the usage of the DLVHEX solver to enable users to interoperate with a broad set of external computation sources. The guide refers to version 2.4 and higher.

Contents

1 Introduction

The DLVHEX system is a logic-programming reasoner for computing the models of so-called HEX-programs [3], which are an extension of *answer-set programs* towards integration of *external computation sources*. To enable access to external information, HEX-programs extend programs with external atoms, which allow for a bidirectional communication between the logic program and external sources of computation (e.g. description logic reasoners and Web resources) [4]. The system is motivated by the need to interoperate with a broad set of external computation sources and the observation, that for meta-reasoning in the context of the Semantic Web, no adequate support is available in ASP to date. To overcome this, HEX-programs support higher-order logic programs (which accommodate meta-reasoning through higher-order atoms) with external atoms for software interoperability.

This guide is intended to help beginners make use of the system and provide a reference for the features of the tool. The language of HEX-programs is an extension of disjunctive datalog, it largely realizes the ASP-Core-2 Standard [2] and extends it with external and high-order atoms.

1.1 Download and Installation

DLVHEX is written in the C++ programming language and published under the GNU Lesser General Public License [9]. In this section we provide an overview of the download and installation process. For a quick overview, some examples and the possibility to evaluate HEX-programs directly in the browser, an online demo is available at `http://www.kr.tuwien.ac.at/research/systems/dlvhex/demo.php`.

1.1.1 Building from source

There are two possibilities to install DLVHEX from source: install the latest stable release of the system or install the latest development version which may not be stable. Both ways are described in the following sections.

1.1.1.1 Latest release version (tarball) Packages (tarballs) of DLVHEX can be downloaded from the project page `http://www.kr.tuwien.ac.at/research/systems/dlvhex/`. The latest release of the software runs on Linux-based systems, Mac OS X and Microsoft Windows. Installation instructions are given in the `INSTALL` and `README` files of DLVHEX and plugin source directories. Changes between versions can be found in the `NEWS` files and details about changes in the `ChangeLog` file.

The system requires the following packages: git, gcc and g++ (version 4.8 or later), BZ2, Python (version 2.7 or later), bison, scons, cmake, automake, autoconf, Curl (version 4 or later), libtool, and Boost (version 1.55 or later).

After downloading Boost from `http://www.boost.org/`, the following steps should be followed in order to install it in a way usable for DLVHEX.

Here and in the following, commands prefixed with "`$`" sign are the commands executed from the system shell. The following commands need to be executed:

```
$ ./bootstrap.sh
```

```
$ ./b2 install --prefix=BPREFIX
```

In this command, `BPREFIX` is the directory where Boost should be installed.

After downloading the latest release version of DLVHEX, the following sequence of commands DLVHEX will configure DLVHEX:

```
$ ./configure
```

To enable the Python features of DLVHEX, `--enable-python` needs to be added as an additional parameter to `configure`. If Boost was installed in a non-standard location, the parameter `--with-boost=BPREFIX` also needs to be added.

After configuration, the following command builds the system:

```
$ make
```

To allow using of multiple cores one should specify the `-jN` option to make use of N cores. Finally,

```
$ make install
```

installs the package. The installation location can be set to `HPREFIX` by adding parameter `--prefix=HPREFIX` to `configure`.

1.1.1.2 Development version (git clone)

The source code of DLVHEX is hosted on github at `https://github.com/hexhex/`. To get the latest development version it is necessary to clone the repository as follows:

```
$ git clone https://github.com/hexhex/core --recursive
```

After cloning it is necessary to execute the script `bootstrap.sh`.

```
$ ./bootstrap.sh
```

After that, the steps from Section 1.1.1.1 (`configure`, `make`, and `make install`) should be followed to complete the installation.

We provide a script for installing DLVHEX automatically on Ubuntu systems: `https://github.com/hexhex/core/blob/master/scripts/setupdlvhex.sh`.

Once installation is completed, DLVHEX can be used as follows:

```
$ dlvhex2 program.hex
```

where `program.hex` refers to the input program. Various additional command line options are available and explained in Section 6.

1.1.2 Pre-built binaries

Pre-built binaries of DLVHEX are available for some systems. For details see our website `http://www.kr.tuwien.ac.at/research/systems/dlvhex/`.

1.2 Outline

This guide is organized as follows. Section 2 provides an introductory example including problem instance, encoding and its solution. Section 3 explains the input language of DLVHEX. In Section 4 we introduce three real life problems which can be solved using HEX. Section 5 is focused on the description of external interfaces which are written in C++ or Python. Input-related warnings and errors are described into more details in Section 7.

2 Quickstart

As an introductory example, we consider a *social graph* as used in social networks. Beginning from a simplified scenario, we stepwisely extend it to present various features of DLVHEX.

2.1 Problem Setting

A *social graph* is a graph that represents interconnections among people, groups and organizations in a social network. Services such as Facebook facilitate the exchange of information, news, photographs, literary works, music, art, software, opinions or even money among users. Individuals and organizations, called actors, are nodes of the graph. In this environment, the social graph for a particular actor consists of the set of nodes and edges which model other actors that are directly connected to that actor. Interdependencies, called ties, can be multiple and diverse, including characteristics or concepts such as age, gender, ethnical group, genealogy, chain of command, ideas, financial transactions, trade relationships, political affiliations, club memberships, occupation, education and economic status. Social graphs contain edges between one person and related people, places, and things they interact with online. For this particular example, we consider a simulation of social graphs as used, e.g., by Facebook.

Consider the situation where a birthday party should be organized and a specific number of friends will be invited. The *person* X who organizes the event wants to call his or her friends and friends of these friends up to some distance from the root node X. A *depth constraint* specifies how many edges we can go away from the root node X.

We make use of an external source which returns for a given person all direct friends, while a direct access to the full graph is not available due to privacy issues imposed by social networks. Also, due to the large amount of data, importing the whole graph would be infeasible (billions of users), while only a small fraction is relevant for the application. Given a person X, the external source retrieves all neighbour nodes (successor nodes). More details about the external source implementation are given in Section 5.

2.2 Encoding

The problem can be modeled as a HEX-program as follows:

Example 2.1 (download example_2_1.hex)**.**

$$\mathbf{r_1}\colon\ personOfInterest(john).$$

$$\mathbf{r_2}\colon\ friendOfDegree(P, 0, P) \text{ :- } personOfInterest(P).$$

$$\mathbf{r_3}\colon\ friendOfDegree(P, DegPlus, F2) \text{ :- } friendsOfDegree(P, Deg, F1),\ \&friendsOf[F1](F2),\ DegPlus = Deg + 1,\ DegPlus < 2,\ \#int(DegPlus), \#int(Deg).$$

$$\mathbf{r_4}\colon\ invite(P) \vee ninvite(P) \text{ :- } friendOfDegree(J, X, P), \#int(X).$$

$$\mathbf{r_5}\colon\ \text{ :- } not\ 4 = \#count\{P : invite(P)\}.$$

□

The complete source code for this example is available at `https://github.com/hexhex/manual/tree/master/example_2_1`.

Rule $\mathbf{r_1}$ specifies the person who organizes the event and initializes the search. Rule $\mathbf{r_2}$ defines that the initiating person has distance 0 from him- or herself.

The most interesting part of the program is rule $\mathbf{r_3}$. It cyclically defines friends of already known persons using an external atom and increments the distance with each definition. Variables used in these predicates are:

- *F1* to represent the person for which we are looking for the successors
- *F2* is the variable for successor nodes of $F1$
- P represents the person of interest
- *Deg* and *DegPlus* are variables used to compute the distance from the root node

The external atom $\&friendsOf[F1](F2)$ has one input and one output parameter. For input *F1*, it finds all successor nodes of *F1* and returns them in *F2*. In other words, $\&friendsOf[F1](F2)$ is true for all pairs $(F1, F2)$ such that *F2* is a direct friend of *F1* in the graph. The implementation of the external atom is discussed in Section 5. The atom

$$friendOfDegree(P, Deg, F1)$$

is true for all friends *F1* of P that we already know; it binds the variable *F1* to a person for which we want to discover more successor nodes. This value is used as input to the external source $\&friendsOf[F1](F2)$ which returns all friends $F2$ of $F1$. For each found friend $F2$ we define:

$$friendOfDegree(P, DegPlus, F2)$$

where *DegPlus* is *Deg* incremented by 1 to represent that the distance to $F2$ is by 1 greater than to $F1$. The condition

$$DegPlus < 2$$

ensures the distance is limited to 2.

We now move to the part where we handle invitations. Rule $\mathbf{r_4}$ guesses all possible persons to be invited or not. Since atom $friendOfDegree(J, X, P)$ is true for person P, that person will be either invited or not.

We limit the number of invited persons by using an *integrity constraint* from the $\mathbf{r_5}$ It ensures that exactly 4 persons are invited to the party.

The combination of $\mathbf{r_4}$ and $\mathbf{r_5}$ can be replaced by a construction called 'choice' as shown in $\mathbf{r_5\prime}$. This rule also allows to specify lower and upper bound on the number of persons independently.

$$\mathbf{r_5\prime}\colon 3 <= \{invite(P) : friendOfDegree(J, X, P)\} <= 3.$$

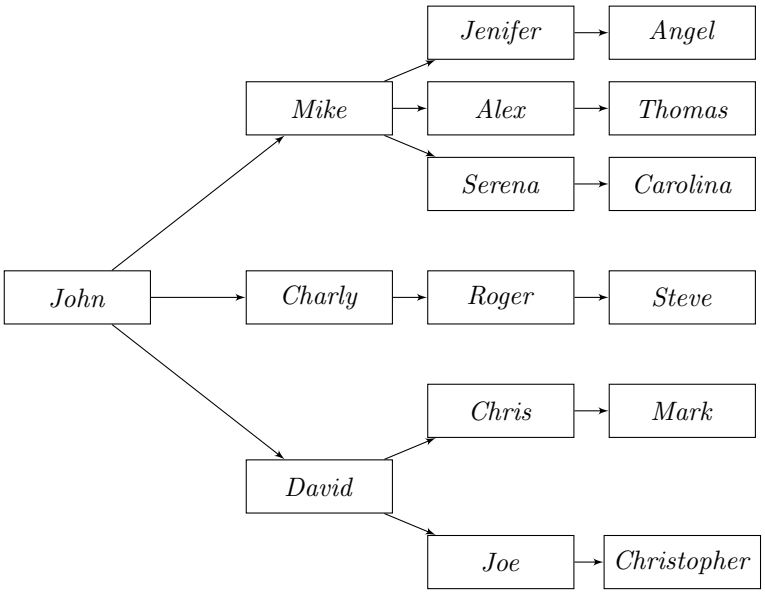

Figure 1: Social Network Graph

2.3 Problem Solution

Now we are ready to solve our *social graph* problem. Consider that we have the data as specified in the Figure 1. To compute the answer sets representing the solution, DLVHEX should be invoked as follows:

```
$ dlvhex2 --pythonplugin=example_2_1.py example_2_1.hex
```

where `example_2_1.hex` is HEX-program and `example_2_1.py` is the Python plugin which realizes the external source implementation. Details of the Python plugin interface are given in Section 5, the files can be downloaded from example_2_1.py (the plugin) and example_2_1.edgelist (the graph in Figure 1).

The output of DLVHEX is as follows:

$$\{personOfInterest(john),\ friendOfDegree(john, 0, john),\ invite(john),\ friendOfDegree(john, 1, mike),\ friendOfDegree(john, 1, david),\ friendOfDegree(john, 1, charly),\ invite(mike),\ invite(david),\ invite(charly)\}$$

Note that the order of the atoms and the order of answer sets does not bear any meaning. As we specified in the previous section, we can travel at most 1 edge far from the root node. Considering the graph given above only, *John*, *Mike*, *Charly* and *David* are found since they are at most one edge away from the root node. The next three atoms express who are the new friends discovered and at which depth level. For the invitations, it is specified by using aggregates that answer sets must have four distinct *invites* atoms. In the single answer set we have four *invites* atoms which are $invite(john), invite(mike), invite(david), invite(charly)$. Note that this is the only answer set possible from this program because there are only 4 distinct persons with depth level 1.

If we allow the depth level to be larger there may be more answer sets found due to the fact that more nodes will be discovered. If we decrease the minimum number of friends to be invited to the party there may also be more than one answer set. Consider the different example where instead of 4 persons we want to invite only 3 persons to the party. The integrity constraint at $\mathbf{r_5}$ will be modified to:

$$\mathbf{r_5}'\text{: } :\text{- } not\ \#count\{P : invite(P)\} = 3.$$

This time we have more than one answer set. Since the depth level is still 2 there will be 4 persons discovered again, however, out of these 4 persons we have to invite only three of them and one of them will not be invited. According to this we have 4 answer sets. Two of them are shown below:

$\{personOfInterest(john), friendOfDegree(john, 0, john),$
$invite(john), friendOfDegree(john, 1, mike), ninvite(charly),$
$friendOfDegree(john, 1, david), friendOfDegree(john, 1, charly),$
$invite(mike), invite(david)\}$
$\{personOfInterest(john), friendOfDegree(john, 0, john),$
$invite(john), friendOfDegree(john, 1, mike), ninvite(mike),$
$friendOfDegree(john, 1, david), friendOfDegree(john, 1, charly),$
$invite(david), invite(charly)\}$

(Again, note that the order of answer sets is arbitrary and does not bear any meaning.) This time in the answer set we have $ninvite(charly)$ and $ninvite(mike)$ since one friend must be discarded and only three will be invited. One can experiment with the *depth constraint* and *aggregate atom* to see how the output and answer sets will be affected.

3 Input Language

This section provides an overview of the input language of DLVHEX and some examples to illustrate the concepts.

3.1 Terms and Atoms

The vocabulary consists of terms, constants, variables and external predicates. Terms may be integers, constants, function terms, strings and variables as well as the "_" token. Constant names begin with lowercase letters or are strings enclosed in quotation marks. Variable names begin with uppercase letters.

Function terms (uninterpreted functions) are *complex terms* composed of a name (like a constant) and one or more terms as arguments. For instance, $at(john, t(10), X)$ is a function term with three arguments: constant *john*, another function term $t(12)$ with an integer argument, and variable X [8].

While a constant, function term, or string, always represents itself; a variable is a placeholder for all variable-free terms in the language of a logic program. An anonymous variable is a special variable denoted by "_" (the underscore): each occurrence of "_" represents a new and unique variable which does not

occur anywhere else in the same rule. This might be used to specify that an argument can be ignored or does not matter.

An *atom* has the form $p(t_1, \ldots, t_n)$ where p is a predicate name, $t_1, \ldots, t_n$ are terms and $n < 0$ is the arity of the predicate p; an atom $p()$ of arity 0 is usually written as p without parentheses. Atoms can be classically negated using "-", yielding q and - q.

Example 3.1. The following example illustrates the above concepts.

Constants: a, 1, *a1*, *9862*, *c1*, "hello"
Variables: X, Y, Z
Atoms: $parent(X, Y)$, $employee(name, salary, ID, location)$
Predicates: *parent*, *employee*

□

3.2 Normal Programs and Integrity Constraints

A HEX-program is constructed using *facts, rules and integrity constraints.*

Fact: A_0.
Rule: A_0 :- L_1,..., L_n.
Constraint: :- L_1,..., L_n.

The sign ":-" is meant to be an implication to the left ($\leftarrow$). The left side of a rule is called its head, and its right side is called body. The head A_0 of a rule or a fact is an atom. In the body of a rule or an integrity constraint, every L_j, $1 \leq j \leq n$, is a literal of the form A or *not* A, where A is an atom and the connective *not* denotes default negation. We say that literal L is positive if it is an atom and negative otherwise. While the head atom A_0 of a fact must unconditionally be true, the intuitive reading of a rule corresponds to an implication: if all positive atoms in the rule body are true and negated atoms are false, then the head A_0 must be true. On the other hand, an integrity constraint is a rule that filters solution candidates: the literals in its body must not jointly be satisfied. A result of a DLVHEX computation is called an *answer set* which is a consistent explanation (model) of the world given the knowledge about the world represented by rules as intuitively explained above.

We here give only informal semantics; for a formal description of semantics of normal ASP programs and HEX-programs we refer to [7] and [3], respectively.

Example 3.2. Consider the following logic program:

$\mathbf{r_1}$: *joke*.
$\mathbf{r_2}$: *laugh* :- *joke*.

The first line here represents an *atom* which is always true. The second line is a *rule* and reads as "If *joke* is true, *laugh* must also be true". Also we can read this as "from *joke* follows *laugh*". The single *answer set* of the program above is $\{joke, laugh\}$ since they are the atoms which are true in the program. □

Another important feature of DLVHEX is *default negation* which is also called "negation as failure". Intuitively, negation as failure means: if we cannot show truth of an atom, then we may safely assume that it is false.

Variables in default negated atoms must be safe which means that they must also occur in a positive atom in the body of the same rule.

Example 3.3. With default negation we can represent the complementary graph $comp\_edge(X, Y)$ of a graph given as atoms $edge(X, Y)$. The complementary graph has the same nodes as the original graph, but of all possible edges it has exactly those edges, which do not exist in the original graph.

$\mathbf{r_1}$: $node(X) :\text{-}\ edge(X, \_).$
$\mathbf{r_2}$: $node(Y) :\text{-}\ edge(\_, Y).$
$\mathbf{r_3}$: $comp\_edge(X, Y) :\text{-}\ node(X), node(Y), not\ edge(X, Y).$

Note that $node(X)$ and $node(Y)$ are included in the body to satisfy the safety requirement for variables X and Y. Also note the anonymous variables "_". □

Default negation allows for generating multiple answer sets that are equally correct as models of a single program. Integrity constraints eliminate those answer set candidates that make the body of the constraint true.

Example 3.4. Consider the following example for coloring nodes of a graph either red, green or black.

$\mathbf{r_1}$: $node(X) :\text{-}\ edge(X, Y).$
$\mathbf{r_2}$: $node(Y) :\text{-}\ edge(X, Y).$
$\mathbf{r_3}$: $colored(X, r) :\text{-}\ node(X), not\ \ colored(X, g), not\ \ colored(X, b).$
$\mathbf{r_4}$: $colored(X, g) :\text{-}\ node(X), not\ \ colored(X, r), not\ \ colored(X, b).$
$\mathbf{r_5}$: $colored(X, b) :\text{-}\ node(X), not\ \ colored(X, r), not\ \ colored(X, g).$
$\mathbf{r_6}$: $:\text{-}\ edge(X, Y), colored(X, C), colored(Y, C).$
$\mathbf{r_7}$: $edge(2, 4).\ edge(2, 3).\ edge(5, 5).$
$\mathbf{r_8}$: $edge(4, 6).\ edge(4, 5).\ edge(5, 7).$
$\mathbf{r_9}$: $edge(6, 7).$

In the first two rules we extract the nodes implicitly given by the edges of the graph. Rules $\mathbf{r_3}$-$\mathbf{r_5}$ describe a *guess* such that for each node X, either $colored(X, r)$, $colored(X, g)$, or $colored(X, b)$ will be true in answer sets. These three rules generate all possible node color combinations. Rule $\mathbf{r_6}$ is a constraint, sometimes called *check*, that deletes all color combinations which do not satisfy the requirement that there may be no edge between two nodes of equal color. □

3.3 Classical Negation

DLVHEX supports two kinds of negation. Here we will emphasize the difference between explicitly expressing falseness of an atom and having it done by negation as failure. The connective *not* expresses default negation, i.e. a literal *not A* is assumed to hold unless atom A is derived to be true. This is also called *Closed World Assumption*. In contrast, the classical (or strong) negation of an atom holds only if it can be derived. In other words if there is no evidence that an atom is true, it is considered to be false. Classical negation "-" is permitted in from of an atom. The semantic relationship between A and $\text{-}\ A$ is simply that they must not jointly hold.

Example 3.5. Imagine a situation where an agent has to cross a railroad. The agent should cross it if there is no train approaching. With this description, one might specify the following program:

$$\mathbf{r_1}\colon \mathit{cross\_railroad} \text{ :- } \mathit{not}\ \mathit{train\_approaches}.$$

This program has the answer set $\{\mathit{cross\_railroad}\}$ because *train_approaches* is assumed to be false (as it being true is not stated anywhere). This kind of negation is called *negation as failure*. □

Example 3.6. Imagine, instead, that we use the following program which uses classical negation (sometimes called strong or true negation).

$$\mathbf{r_1}\colon \mathit{cross\_railroad} \text{ :- } \text{ - } \mathit{train\_approaches}.$$

Since - *train_approaches* is not known to be true, the program has only an empty answer set. □

Note the difference between the two kinds of negation: in the first example, the railroad track is crossed if there is no information on any trains approaching; while in the second example, it is only crossed if we can prove that no train comes. In that sense classical negation is stronger than negation as failure.

3.4 Disjunctive Programs

Disjunctive logic programs permit the connective "$\vee$" between atoms in rule heads.

$$\begin{array}{l}\text{Fact: } A_0 \vee \ldots \vee A_m.\\ \text{Rule: } A_0 \vee \ldots \vee A_m \text{ :- } L_1, \ldots, L_n.\end{array}$$

A *disjunctive head* holds if at least one of its atoms is true. If all body literals $L_1, \ldots, L_n$ of the rule specified above are known to be true then the head also needs to hold, i.e. one of the atoms in $A_0 \vee \ldots \vee A_m$ needs to be true. If our program just contains the fact "$a \vee b$" we obtain two answer sets $\{a\}$ and $\{b\}$.

Disjunctive logic programs intuitively make the minimum of disjunctive heads true that are necessary to satisfy all constraints to obtain an answer set. Formally speaking semantics are not that simple: disjunctive programs have a higher expressive power than normal logic programs (it is possible to encode subset minimality using advanced techniques such as *saturation* that we will not discuss further in this manual).

Example 3.7. In the Example 3.4 we could replace the guess in $\mathbf{r_3}$-$\mathbf{r_5}$ by the following single disjunctive rule:

$$\mathit{colored}(X, r) \vee \mathit{colored}(X, g) \vee \mathit{colored}(X, b) \text{ :- } \mathit{node}(X).$$

This generates all possible node color combinations. □

Example 3.8. Suppose we met a friend recently and know that he had one of his arms broken, but do not know which one. Now suppose we did not receive a greeting card for your birthday and wonder if you should be angry on him or he

just could not write because his right hand is broken. Suppose that we encode this reasoning problem in HEX.

$\mathbf{r_1}$: $left\_arm\_broken \vee right\_arm\_broken.$

$\mathbf{r_2}$: $can\_write :\!\text{-}\ left\_arm\_broken.$

$\mathbf{r_3}$: $be\_angry :\!\text{-}\ can\_write.$

For this program, DLVHEX will generate two possible explanations. The first rule is called a disjunctive rule which is read as "For sure, either the left or the right arm is broken." Without being sure which arm is broken DLVHEX will evaluate the program and produce the two models $\{left\_arm\_broken, can\_write, be\_angry\}$ and $\{right\_arm\_broken\}$. □

3.5 Built-in Arithmetic Functions

DLVHEX supports integers and the following arithmetic operators: $+$ (addition), $-$ (subtraction), $*$ (multiplication), and $/$ (integer division). Atom $+(X, Y, Z)$ is true, iff Z is the sum of X and Y and likewise for other operators.

Alternatively to *prefix notation* one can also use *infix notation* to use built-in arithmetic functions in DLVHEX. For instance $+(X, Y, Z)$ alternatively can be written as $Z = X + Y$.

Example 3.9. Suppose the following program.

$\mathbf{r_1}$: $a(6).$

$\mathbf{r_2}$: $b(2).$

$\mathbf{r_3}$: $c(X, Y, Z) :\!\text{-}\ a(X), b(Y), +(X, Y, Z).$

$\mathbf{r_4}$: $d(X, Y, Z) :\!\text{-}\ a(X), b(Y), -(X, Y, Z).$

$\mathbf{r_5}$: $e(X, Y, Z) :\!\text{-}\ a(X), b(Y), *(X, Y, Z).$

$\mathbf{r_6}$: $f(X, Y, Z) :\!\text{-}\ a(X), b(Y), /(X, Y, Z).$

The single answer set for the example above is:

$$\{a(6), b(2), e(6, 2, 12), f(6, 2, 3), c(6, 2, 8), d(6, 2, 4)\}.$$

The infix alternatives of rules $\mathbf{r_3}$-$\mathbf{r_6}$ above are as follows.

$\mathbf{r_3'}$: $c(X, Y, Z) :\!\text{-}\ a(X), b(Y), Z = X + Y.$

$\mathbf{r_4'}$: $d(X, Y, Z) :\!\text{-}\ a(X), b(Y), Z = X - Y.$

$\mathbf{r_5'}$: $e(X, Y, Z) :\!\text{-}\ a(X), b(Y), Z = X * Y.$

$\mathbf{r_6'}$: $f(X, Y, Z) :\!\text{-}\ a(X), b(Y), Z = X/Y.$

□

3.6 Built-in Comparison Predicates

DLVHEX features a total order among variable-free terms using built-in predicates == (equal), ! = or <> (not equal), < (less than), <= (less than or equal), > (greater than) and >= (greater than or equal). All ground terms and integers can be compared this way. Integer comparison is according to numeric values. All other comparisons just guarantee a fixed ordering over all terms.

Example 3.10. Suppose we have the following program.

$$\begin{aligned}
&\mathbf{r_1}\text{: } a(1).\\
&\mathbf{r_2}\text{: } a(2).\\
&\mathbf{r_3}\text{: } b(1).\\
&\mathbf{r_4}\text{: } c(X,Y) \text{ :- } a(X), b(Y), X \,!= Y.\\
&\mathbf{r_5}\text{: } d(X,Y) \text{ :- } a(X), b(Y), X <> Y.\\
&\mathbf{r_6}\text{: } e(X,Y) \text{ :- } a(X), b(Y), X < Y.\\
&\mathbf{r_7}\text{: } f(X,Y) \text{ :- } a(X), b(Y), X > Y.\\
&\mathbf{r_8}\text{: } g(X,Y) \text{ :- } a(X), b(Y), X <= Y.\\
&\mathbf{r_9}\text{: } h(X,Y) \text{ :- } a(X), b(Y), X >= Y.\\
&\mathbf{r_{10}}\text{: } i(X,Y) \text{ :- } a(X), b(Y), Y == 1.
\end{aligned}$$

The single answer set this program is

$$\begin{aligned}
\{&a(1), a(2), b(1), i(1,1), i(2,1), c(2,1),\\
&d(2,1), f(2,1), g(1,1), h(1,1), h(2,1)\}.
\end{aligned}$$

□

3.7 Conditions and Conditional Literals

A *conditional literal* is of the form

$$L_0 : L_1, \ldots, L_n$$

where every L_j with $0 \leq j \leq n$ is a literal, the literals $L_1, \ldots, L_n$ are called *condition*, and ":" resembles the vertical bar (|) in mathematical set notation. The condition is expanded into all ground versions of L_0 where the condition is true. For example, the rule a :- $b : c$. yields a whenever either c is false (whether b holds or not) or both b and c are true.

Together with variables, conditions allow for specifying collections of expressions within a single rule or aggregate. This is particularly useful for encoding conjunctions (or disjunctions) over arbitrarily many ground atoms as well as for the compact representation of aggregates [8].

Example 3.11. Consider the following program for scheduling a meeting [8].

$$\begin{aligned}
&\mathbf{r_1}\text{: } person(jane).\ person(john).\\
&\mathbf{r_2}\text{: } day(mon).\ day(tue).\ day(wed).\ day(thu).\ day(fri).\\
&\mathbf{r_3}\text{: } available(jane) \text{ :- } not\ on(fri).\\
&\mathbf{r_4}\text{: } available(john) \text{ :- } not\ on(mon), not\ on(wed).\\
&\mathbf{r_5}\text{: } meet \text{ :- } available(X) : person(X).\\
&\mathbf{r_6}\text{: } on(X) : day(X) \text{ :- } meet.
\end{aligned}$$

Conditions are used in $\mathbf{r_5}$ and $\mathbf{r_6}$. The *conjunction* in $\mathbf{r_5}$ is instantiated by replacing X in $available(X)$ with all ground terms t such that $person(t)$ holds, namely $X = jane$ and $X = john$. The condition in $\mathbf{r_6}$ is contained in the head of the rule and turns into a *disjunction* over all ground instances of $on(X)$ such

that X is substituted by terms t for which $day(t)$ holds, i.e., whenever $meet$ holds, one of $on(t)$ for which $day(t)$ is true should hold.

This program has the following two answer sets.

$$\{meet, on(tue), day(mon), day(tue), day(wed), day(thu), day(fri), person(jane), person(john), available(jane), available(john)\}$$
$$\{meet, on(thu), day(mon), day(tue), day(wed), day(thu), day(fri), person(jane), person(john), available(jane), available(john)\}$$

□

3.8 Aggregates

Aggregates allow to express properties over sets of elements. HEX-programs with aggregates often allow clean and concise problem encodings by minimizing the use of auxiliary predicates and recursive programs, and help the programmers to depict problems in a more natural way. For instance, we may state that the sum of a semester's course credits must be at least 20, or that the sum of shopping items must not exceed 30 Euros. We can say that an aggregate is a function on a set of tuples that are normally subject to conditions. By comparing an aggregated value with given values, we can extract a truth value from an aggregate's evaluation, thus obtaining an aggregate atom. Aggregates can occur in the bodies of rules and constraints, possibly negated using negation-as-failure [8]. The form of an *aggregate atom* is as follows:

$$s_1 \prec_1 \alpha\{t_1, \ldots, t_n : L_1, \ldots, L_m\} \prec_2 s_2$$

where t_i are terms, L_i are literals, α is a function that evaluates the numerical value of the aggregate, and $\prec_1/\prec_2$ are comparison predicates that compare the resulting value with the terms s_1/s_2. An aggregate is true if the comparison is true with respect to evaluating α on those tuples $t_1, \ldots, t_n$ for which $L_1, \ldots, L_m$ in the aggregate body is true.

Supported aggregate functions α are $\#count$, $\#sum$, $\#times$, $\#min$, and $\#max$.

Example 3.12. Consider the following program where we want to count how many employees of the company earn more than 1000.

$\mathbf{r_1}$: $emp(1, goofie, 1250).$
$\mathbf{r_2}$: $emp(2, willy, 750).$
$\mathbf{r_3}$: $emp(3, woody, 750).$
$\mathbf{r_4}$: $emp(4, jerry, 900).$
$\mathbf{r_5}$: $emp(5, tom, 1050).$
$\mathbf{r_6}$: $over1000(I, S) \text{ :- } emp(I, N, S), S > 1000.$
$\mathbf{r_7}$: $over1000nr(X) \text{ :- } \#count\{I : over1000(I, W)\} = X, \#int(X).$

Intuitively the aggregate is expanded into

$$\#count\{over1000(1, 1250), over1000(5, 1050)\}$$

and the (symbolic) set of tuples I appearing in the aggregate looks as follows:

$$\{\langle 1\rangle, \langle 5\rangle\}.$$

The aggregate function $\#count$ returns the cardinality of the symbolic set to which it is applied, in this case it returns 2.

As a result this program has the following unique answer set.

$$\{emp(1, goofie, 1250), emp(2, willy, 750), emp(3, woody, 750),\\ emp(4, jerry, 900), emp(5, tom, 1050),\\ over1000(1, 1250), over1000(5, 1050), over1000nr(2)\}$$

□

Aggregate functions get as input the set of tuples from the aggregate and they operate on the first item of these tuples.

Example 3.13. Suppose we want to know how much the company spends on salaries, then we can use the following rule.

$$\mathbf{r_8}\colon\ salaryTotal(X) \text{ :- } \#sum\{S, I : emp(I, N, S)\} = X.$$

The symbolic set for the rule $\mathbf{r_8}$ consists of 5 elements:

$$\{\langle 1250, 1\rangle, \langle 750, 2\rangle, \langle 750, 3\rangle, \langle 900, 4\rangle, \langle 1050, 5\rangle\}.$$

The $\#sum$ function returns the sum over the first elements of all tuples in the set, therefore the answer set contains the fact $salaryTotal(4700)$.

Note that the first term in the tuple is the salary S, but we need also the second term I in order to get the correct result: consider the simplified rule

$$\mathbf{r'_8}\colon\ salaryTotal(X) \text{ :- } \#sum\{S : emp(I, N, S)\} = X.$$

without I: the symbolic set with $\mathbf{r'_8}$ looks as follows.

$$\{\langle 1250\rangle, \langle 750\rangle, \langle 900\rangle, \langle 1050\rangle\}$$

It contains only one element for both employees with salary 750, therefore we obtain the incorrect result $salaryTotal(3950)$. □

The aggregate function $\#times$ computes the product of the first values of tuples in the symbolic set. When applied over the empty set, $\#times$ returns 1. The aggregate function $\#min$ (resp., $\#max$) returns the minimum (resp., maximum) value of the first values of tuples in the symbolic set. Note that one aggregate body can contain multiple symbolic set constructors of form $t_1, \ldots, t_n\colon L_1,\ \ldots, L_m$ by separating them using ";".

3.9 Optimization

Introducing *weak constraints* into HEX-programs allows us to formulate several optimization problems in an easy and natural way. While standard constraints (integrity constraints, strong constraints) always have to be satisfied, weak constraints can be satisfied *at a cost* and the answer sets of a program P with a

set W of weak constraints are those answer sets of P which minimize the cost of violated weak constraints.

Weak constraints can be weighted according to their importance (the higher the weight, the more important the constraint). In the presence of weights, best models minimize the sum of the weights of the violated weak constraints. Weak constraints can also be prioritized. Under prioritization, the semantics minimizes the violation of the constraints of the highest priority level first; then the lower priority levels are considered one after the other in descending order.

Weak constraints are specified as follows.

$$:\sim b_1, \ldots, b_n. \quad [w@l, t_1, \ldots, t_m]$$

As in aggregates, $t_1, \ldots, t_m$, $m > 0$, are terms that specify a symbolic set over which the cost of constraint violations is computed. As in a normal rule, $b_1, \ldots, b_n$, $n > 0$, are literals, and w and l are terms standing for a *weight* and a *level*. If omitted, the level defaults to 0.

Example 3.14. Consider we want to compute the minimum spanning trees of a weighed directed graph.

$\mathbf{r_1}$: $root(a).$

$\mathbf{r_2}$: $node(a).\ node(b).\ node(c).\ node(d).\ node(e).$

$\mathbf{r_3}$: $edge(a,b,4).\ edge(a,c,3).\ edge(c,b,2).\ edge(c,d,3).$

$\mathbf{r_4}$: $edge(b,e,4).\ edge(d,e,5).$

$\mathbf{r_5}$: $in\_tree(X,Y,C) \vee out\_tree(X,Y) :\text{-}\ edge(X,Y,C), reached(X).$

$\mathbf{r_6}$: $:\text{-}\ root(X), in\_tree(\_,X,C).$

$\mathbf{r_7}$: $:\text{-}\ in\_tree(X,Y,C), in\_tree(Z,Y,C), X\,!=Z.$

$\mathbf{r_8}$: $reached(X) :\text{-}\ root(X).$

$\mathbf{r_9}$: $reached(Y) :\text{-}\ reached(X), in\_tree(X,Y,C).$

$\mathbf{r_{10}}$: $:\text{-}\ node(X), not\ reached(X).$

$\mathbf{r_{11}}$: $:\sim in\_tree(X,Y,C).[C@1, X, Y, C]$

The fact $\mathbf{r_1}$ of the example above defines the root node of a tree. Nodes and edges are defined in $\mathbf{r_2}$ and $\mathbf{r_3}$. Rule $\mathbf{r_5}$ guesses for each edge from node X to a node Y (node X is already reached) whether it is in the minimum spanning tree or out of it. The integrity constraint $\mathbf{r_6}$ ensures that there is no incoming edge to the root node. Rule $\mathbf{r_7}$ eliminates all answer sets where there are two outgoing edges going to the same node Y. In $\mathbf{r_8}$ and $\mathbf{r_9}$ we compute all reached nodes and $\mathbf{r_{10}}$ removes all answer sets where there is some node which is not reached.

Rule $\mathbf{r_{11}}$ is a weak constraint with weight C and level 1. Intuitively this constraint says that using an edge has a cost corresponding to the weight of the edge. As a consequence, minimizing cost of the answer set will find a tree starting at the root node reaching all nodes with minimal cost, i.e., a minimal spanning tree.

The best answer set has cost 12 at level 1 and is as follows (omitting facts).

$$\{reached(a), reached(b), reached(c), reached(d), reached(e),$$

$out\_tree(a, b), out\_tree(d, e),$
$in\_tree(a, c, 3), in\_tree(b, e, 4), in\_tree(c, b, 2), in\_tree(c, d, 3)\}$

The complete source code for this example is available at example_3_opti.hex.

To obtain all optimal answer sets execute the following command.

```
$ dlvhex2  example_3_opti.hex
```

To obtain all answer sets, even non-optimal ones execute the following.

```
$ dlvhex2  example_3_opti.hex  --weak-allmodels
```

□

More examples can be found in the DLV-User Manual [1].

DLVHEX supports two evaluation algorithms for weak constraints. The way they work has some implications on the runtime performance as well as memory characteristics. The default algorithm usually performs better, but may consume more memory. If the evaluation fails because of a memory shortage, one can try and rerun DLVHEX with the parameter `--heuristics=monolithic`. With this parameter, the second algorithm is used, which consumes less memory.

3.10 Higher-order Atoms

HEX-programs are non-monotonic logic programs admitting *high-order atoms*, which are atoms containing a variable predicate symbol (instead of a constant). A high-order atom allows to quantify values over predicate names, and to freely exchange predicate symbols with constant symbols, like in the rule

$$C(X) \leftarrow subclassOf(D,C), D(X)$$

where $C(X)$ and $D(X)$ are high-order atoms. An atom can be seen as a tuple $(Y_0, Y_1, \ldots, Y_n)$, where $Y_0, Y_1, \ldots, Y_n$ are terms and $n \geq 0$ is the *arity* of the atom. Intuitively, Y_0 is the predicate name, and we thus also use the more familiar notation $Y_0(Y_1, \ldots, Y_n)$. The atom is *ordinary*, if Y_0 is a constant, it is *high-order* if Y_0 is a variable.

For example, $(x, rdf : type, c)$, $node(X)$, and $D(a, b)$, are atoms; the first two are ordinary atoms.

3.11 External Atoms

External atoms are the central feature that distinguishes HEX-programs from normal ASP programs.

Through external atoms, HEX-programs can communicate with other sources of computation; this can be used to model part of a program outside of ASP (e.g., parts that cannot be modeled in ASP such as 3D simulations or Description Logic reasoning) or to import knowledge into ASP (e.g., from Semantic Web triplestores or from databases).

An *external atom* is of the form

$$\&g[Y_1, \ldots, Y_n](X_1, \ldots, X_m),$$

where $Y_1, \ldots, Y_n$ and $X_1, \ldots, X_m$ are the two lists of terms (called *input* and *output* lists, respectively), and $\&g$ is an external predicate name.

Intuitively, an external atom provides a way for deciding the truth value of an output tuple depending on the input tuple and depending on the truth

of input predicates in the answer set. An external atom can have three kinds of inputs: *predicate input* provides the truth values of all atoms of a predicate to the computation, while *constant input* only provides the constant symbol to the computation. The third input type is *tuple input* which allows an arbitrary amount of constant inputs as arguments. The type of an input must be specified when implementing the external computation in the DLVHEX API (see Section 5).

Example 3.15. For instance, the rule

$$reached(X) \leftarrow \&reach[edge, a](X).$$

defines the predicate *reached* and takes values from the external predicate $\&reach$, which computes in an external program all the reachable nodes X in the graph specified by predicates *edge* starting from node a. Here we assume $\&reach$ takes a predicate as first input and a constant as second input. Then *edge* is a predicate input: the computation of $\&reach$ depends on the truth values of predicate *edge* in the answer set. On the other hand, a is a constant input: the computation of $\&reach$ only uses the symbol a. □

Note that because we used a predicate input in the previous example, the above example can define reachability in a graph that is part of a guessed in an answer set. This is a feature unique to DLVHEX: it cannot be emulated by externals in Gringo (which have to be computed during grounding, while DLVHEX computes external atoms during both grounding and solving).

In the next example we give a full program with external atoms including their implementation.

Example 3.16. The following program concatenates strings specified by the *system* predicate, computes a set difference, and then associates items in the resulting set and the strings in a unique mapping.

$\mathbf{r_1}$: $system(dlvhex).\ system(clasp).$

$\mathbf{r_2}$: $sayhello(X) :- \&concat[hello, Y](X), system(Y).$

$\mathbf{r_3}$: $set1(a).\ set1(b).\ set1(c).$

$\mathbf{r_4}$: $set2(b).\ set2(c).\ set2(d).$

$\mathbf{r_5}$: $set3(X) :- \&setdiff[set1, set2](X).$

$\mathbf{r_6}$: $pairs(X, Y) :- \&sortandmap[sayhello, set3](X, Y).$

$\mathbf{r_2}$ concatenates strings to produce messages in $sayhello(X)$, $\mathbf{r_5}$ computes a set difference in $set3(X)$, and $\mathbf{r_6}$ produces a unique mapping between hello messages and the set difference.

There are three different external sources in this program, $\&concat$ has a tuple input and one output and computes as output the concatenation of all inputs, $\&setdiff$ uses two predicate inputs and also has one output and computes as output the set difference of constants in the first and the second input, finally $\&sortandmap$ has two predicate inputs and two outputs and computes a sorted one-to-one mapping between constants in first and second inputs.

These external sources are implemented in the following Python plugin:

```
import dlvhex

# concat has one input parameter of type tuple
# (=arbitrarily many constants),
# which specifies the terms to be concatenated
def concat(tup):
  # start with empty string
  ret = ""
  for x in tup:
    # append all input constants in sequence
    ret = ret + x.value()
  # output the final string
  dlvhex.output( (ret,) )

# computes the set of all elements in the
# extension of unary predicates p minus q
def setdiff(p, q):
  # go over all input atoms (p or q)
  for x in dlvhex.getTrueInputAtoms():
    # get predicate/argument of atom
    pred, arg = x.tuple() # pred(arg)
    # check if x is of form p(arg)
    if pred == p:
      # produce atom q(arg)
      qatom = dlvhex.storeAtom( (q, arg) )
      # check q(arg) is NOT in input
      if dlvhex.isFalse(qatom):
        # then put arg into the output
        dlvhex.output( (arg,) )

# computes the extension of unary predicates p and q
# returns unique pairing between sorted elements
def sortandmap(p, q):
  # get all tuples
  tuples = [ x.tuple() for x in dlvhex.getTrueInputAtoms() ]
  # p and q tuples
  ptuples = filter(lambda x: x[0] == p, tuples)
  qtuples = filter(lambda x: x[0] == q, tuples)
  # sorted p and q extensions
  pext = sorted(map(lambda x: x[1], ptuples))
  qext = sorted(map(lambda x: x[1], qtuples))
  # output all pairs
  for out in zip(pext, qext):
    dlvhex.output(out)

# register external atoms
def register():
  # setdiff has two predicate input parameters
  # and its output arity is 1
  dlvhex.addAtom("setdiff",
    (dlvhex.PREDICATE, dlvhex.PREDICATE), 1)
  # concat has arbitrarily many input parameters
  # of type constant (=TUPLE) and its output arity is 1
  dlvhex.addAtom("concat", (dlvhex.TUPLE,), 1)
  # sortandmap has two predicate input parameters
  # and its output arity is 2
  dlvhex.addAtom("sortandmap",
    (dlvhex.PREDICATE, dlvhex.PREDICATE), 2)
```

Answer sets of the above program are obtained by invoking

```
$ dlvhex2 --pythonplugin=example_3_stringset.py \
```

`example_3_stringset.hex`

We obtain the following answer set.

$$\{system(dlvhex), system(clasp), set1(a), set1(b), set1(c), set1(d),$$
$$set2(b), set2(d), set3(a), set3(c), sayhello(hellodlvhex), sayhello(helloclasp),$$
$$pairs(hellodlvhex, a), pairs(helloclasp, c)\}$$

Note that we use this example demonstration purposes, in practice set difference can easier be realized using rules. The complete source code for this example is available at `https://github.com/hexhex/manual/tree/master/example_3_stringset` (example_3_stringset.hex and example_3_stringset.py). □

4 Examples

We now present three real life examples which are encoded and solved by HEX-programs. In Section 4.1 we solve a basic problem about choosing a proper swimming location [3]. In Section 4.2 we solve the traveling salesperson problem. In Section 4.3 we show how we can use DLVHEX to plan routes of one or multiple agents in the dynamic environment controlled by the external atoms.

4.1 Swimming Example

In this example we present a program which selects the best swimming location among some available choices for a swim in Vienna. Selecting a location has to satisfy all constraints given from the user, and properties of swimming locations are imported from external using an external atom.

The complete source code for this example is available at `https://github.com/hexhex/manual/tree/master/example_4_swim` (example_4_swim.hex and example_4_swim.py).

4.1.1 Problem

Imagine Alice wants to go for a swim in Vienna. She knows two indoor pools called Margarethenbad and Amalienbad (represented by *margB* and *amalB*, respectively), and she knows that outdoor swimming is possible in the river Danube at two locations called Gänsehäufel and Alte Donau (denoted *gansD* and *altD*, respectively). She looks up on the Web whether she needs to pay an entrance fee, and what additional equipment she needs. Finally she has the contraint that she does not want to pay for swimming. Assume Alice finds out that indoor pools in general have an admission fee, and that one also has to pay at Gänsehäufel, but not at Alte Donau. Furthermore Alice reads some reviews about swimming locations and finds out that she will need her Yoga mat for Alte Donau because the ground is so hard, and she will need goggles for Amalienbad because there is so much chlorine in the water.

The problem we will solve with HEX is to find a suitable swimming location for Alice.

4.1.2 Encoding

A HEX-program used to select an appropriate swimming location is as follows.

$$
\begin{aligned}
&\mathbf{r_1}\colon location(ind, margB).\ location(ind, amalB).\\
&\quad location(outd, gansD).\ location(outd, altD).\\
&\mathbf{r_2}\colon swim(ind) \vee swim(outd).\\
&\mathbf{r_3}\colon need(inoutd, C) \text{ :- } \&rq[swim](C).\\
&\mathbf{r_4}\colon goto(X) \vee ngoto(X) \text{ :- } swim(P), location(P, X).\\
&\mathbf{r_5}\colon go \text{ :- } goto(X).\\
&\mathbf{r_6}\colon \text{ :- } not\ go.\\
&\mathbf{r_7}\colon \text{ :- } goto(X), goto(Y), X\,!=Y.\\
&\mathbf{r_8}\colon need(loc, C) \text{ :- } \&rq[goto](C).\\
&\mathbf{r_9}\colon \text{ :- } need(X, money).
\end{aligned}
$$

This program above represents Alice's reasoning problem. Rule $\mathbf{r_1}$ contains a set of facts about possible swimming locations (where *ind* and *outd* are short for indoor and outdoor, respectively). Rule $\mathbf{r_2}$ chooses indoor vs. outdoor swimming locations, and $\mathbf{r_3}$ collects requirements that are caused by this choice using the external atom. In $\mathbf{r_4}$ it is decided whether to visit indoor or outdoor locations. By $\mathbf{r_5}$ we define *go* if at least one location is selected, and $\mathbf{r_6}$ removes answer set candidates where no location is selected. Constraint $\mathbf{r_7}$ ensures that only a single location is selected. Rule $\mathbf{r_8}$ collects all requirements caused by the choice of *goto* location. Finally $\mathbf{r_9}$ states that all candidate solutions where Alice has to pay are removed.

Resources are obtained from the external atom of the form

$$\&rq[\textit{location-choice}](\textit{required-resources})$$

which intuitively evaluates to true if a given *location-choice* requires a certain *required-resource* and represents such resources and their origin (*inoutd* or *loc*) using the predicate *need*. The external atom $\&rq$ has input and output arity $in(\&rq) = out(\&rq) = 1$. Intuitively $\&rq[\alpha](\beta)$ is true if a resource β is required when swimming is a place in the extension of predicate α. For example, $\&rq[swim](money)$ is true if $swim(ind)$ is true because indoor swimming pools charge money for swimming.

The implementation of the external atom $\&rq$ in Python is discussed in Example 5.2 on page 31.

4.1.3 Problem Solution

To obtain all answer sets of the above program, execute the following command.

```
$ dlvhex2 --pythonplugin=example_4_swim.py example_4_swim.hex
```

The DLVHEX solver gives the following single answer set as output.

$$
\begin{aligned}
\{&location(ind, margB), location(ind, amalB), location(outd, altD),\\
&location(outd, gansD), swim(outd), go, ngoto(gansD), goto(altD),
\end{aligned}
$$

$need(loc, yogamat)\}$

Under this answer set, the external atom $\&rq[goto](yogamat)$ is true and all others (e.g., $\&rq[swim](money)$, $\&rq[goto](money)$) are false.

Intuitively, the answer set tells Alice to take her Yoga mat and go for a swim to Alte Donau (outside) which is free of charge. This is the only answer set which satisfies the given constraints.

4.1.4 Additional Remarks

In this example, the external atom $\&rq$ uses a predicate as an input parameter. If we want to use a constant input instead of a predicate then $\mathbf{r_3}$ and $\mathbf{r_8}$ should be replaced with

$$\mathbf{r'_3}\colon need(inoutd, C) \text{ :- } \&rq'[Swim](C), swim(Swim).$$
$$\mathbf{r'_8}\colon need(loc, C) \text{ :- } \&rq'[Goto](C), goto(Goto).$$

where $\&rq'$ is implemented differently from $\&rq$.

This will result in a larger grounding (the extensions of *swim* and *goto* are expanded into rules).

4.2 Traveling Salesperson Example

In this section we consider the well-known traveling salesperson problem, where the task is to decide whether there is a round trip that visits each node in a graph exactly once and whose accumulated edge costs must not exceed some budget B.

4.2.1 Problem

The traveling salesperson problem describes a salesperson who must travel between N cities. The order is not relevant as long as all cities are visited exactly once and the route is closed. Each of the links between the cities has a weight (or the costs) attached. The costs describe how "difficult" it is to traverse this edge on the graph, and may be given, for example, by the cost of an airplane ticket or train ticket, or perhaps by the length of the edge, or time required to complete the traversal [10].

This example is interesting for us because it is a typical optimization problem. Among all answer sets DLVHEX should select the best one according to the weak constraints concept explained in Section 3.9. In the classical solver we are able to load only small graphs because it is not feasible to load large graphs completely into memory. In this example we are using an external atom which is loading a graph from the external source edge by edge up to a specified depth level what makes it able to solve problems with extremely large graphs by considering only the part of the graph that is relevant for solving the problem.

The source code for this example is available at `https://github.com/hexhex/manual/tree/master/example_4_travel` (example_4_travel.hex, example_4_travel.py, and example_4_travel.graph).

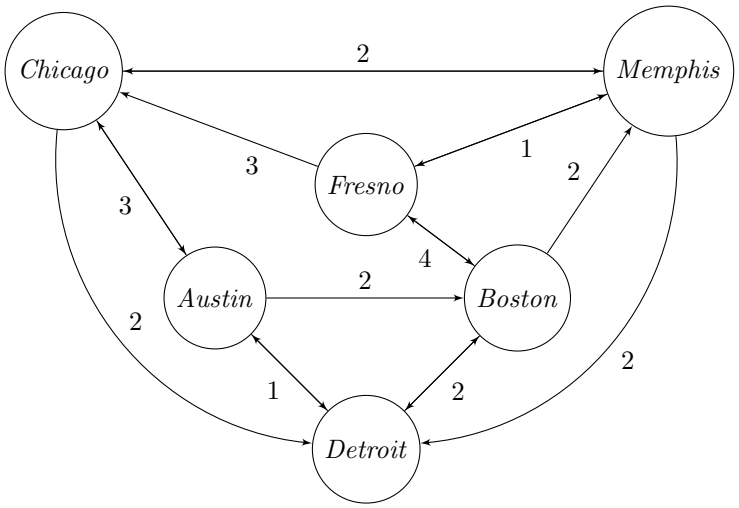

Figure 2: Graph of cities for the Traveling salesperson example.

4.2.2 Encoding

The HEX-program for this problem is as follows.

$\mathbf{r_1}$: $startingCity(austin)$.

$\mathbf{r_2}$: $budgetB(11)$.

$\mathbf{r_3}$: $cityOfDegree(P, 0, P, 0) \text{ :- } startingCity(P)$.

$\mathbf{r_4}$: $cityOfDegree(F1, DegPlus, F2, Cost) \text{ :- } cityOfDegree(\_, Deg, F1, \_),$
$\&edges[F1](F2, Cost), DegPlus = DegPlus + 1, DegPlus < 4,$
$\#int(DegPlus), \#int(Deg)), \#int(Cost)$.

$\mathbf{r_5}$: $node(Y) \text{ :- } cityOfDegree(X, V, Y, C)$.

$\mathbf{r_6}$: $edge(X, Y) \text{ :- } cityOfDegree(X, V, Y, C)$.

$\mathbf{r_7}$: $cost(X, Y, C) \text{ :- } cityOfDegree(X, V, Y, C)$.

$\mathbf{r_8}$: $\{cycle(X, Y) : edge(X, Y)\} = 1 \text{ :- } node(X)$.

$\mathbf{r_9}$: $\{cycle(X, Y) : edge(X, Y)\} = 1 \text{ :- } node(Y)$.

$\mathbf{r_{10}}$: $costCalculated(X) \text{ :- } \#sum\{C, X, Y : cycle(X, Y), cost(X, Y, C)\} = X$.

$\mathbf{r_{11}}$: $withinBudget(B, C) \text{ :- } budget(B), costCalculated(C), B >= C$.

$\mathbf{r_{12}}$: $\text{ :- } budgetB(B), costCalculated(C), not\ withinBudget(B, C)$.

$\mathbf{r_{13}}$: $reached(Y) \text{ :- } cycle(Start, Y), startingCity(Start)$.

$\mathbf{r_{14}}$: $reached(Y) \text{ :- } cycle(X, Y), reached(X)$.

$\mathbf{r_{15}}$: $\text{ :- } node(Y), not\ reached(Y)$.

$\mathbf{r_{16}}$: $:\sim cycle(X, Y), cost(X, Y, C).\ [C@1, X, Y, C]$

Fact $\mathbf{r_1}$ specifies the starting city, i.e., the city where the salesperson starts and finishes the trip. Fact $\mathbf{r_2}$ defines the available budget of the salesperson.

The predicate *cityOfDegree*(R, D, S, C) keeps track of the cities newly discovered defined as successor nodes S for the root node R, their distances from the root node as D and the weight of the edge between R and S denoted as C. Rule $\mathbf{r_3}$ defines the starting city for discovered cities, rule $\mathbf{r_4}$ defines a graph of the discovered cities using the external atom &*edges* to load new cities from the external file and use them in the program.

The external atom is of the form &*edges*$[File, F_1](F_2, Cost)$ where *File* is the file to load the graph from, and F_1 represents the node for which we want to know all successor nodes. The external atom semantics function returns all pairs $(F_2, Cost)$ such that F_2 is successor node of F_1 and the edge has weight *Cost*. *DegPlus* is limited to be less than 4, which means we can go at most three edges far from the root node. There are several potential advantages of using an external atom of this type:(i) the graph may be very large and it is not possible to load it at once as a set of edges specified manually, (ii) we do not know the graph completely (e.g., due to limited capabilities of a web service), or (iii) we want to analyze only a subgraph of the graph that is reachable from the specified node.

In $\mathbf{r_5}$, $\mathbf{r_6}$ and $\mathbf{r_7}$ the program is extracting *nodes*, *edges* and *costs* of the edges from the *cityOfDegree* atoms. Rules $\mathbf{r_8}$ and $\mathbf{r_9}$ assert that every node must have exactly one outgoing and exactly one incoming edge, respectively, belonging to the cycle. The syntax used in these rules is described in Section 3.7.

In $\mathbf{r_{10}}$ we use aggregates (cf. Section 3.8) to find the sum X of the costs over the *cycle*(X, Y). In rule $\mathbf{r_{11}}$, an atom *withinBudget*(B, C) is true if the term of the *costCalculated*(C) is less than or equal to the term of *budgetB*(B) available. The integrity constraint $\mathbf{r_{12}}$ ensures that in the answer set overall sum of the costs for the cycle will be less than or equal to the budget available. The same rules can be applied to limit length traveled or time spent.

Rules $\mathbf{r_{13}}$ and $\mathbf{r_{14}}$ define reachability of nodes, startin from the initial node and using cycle candidates produced by the rules $\mathbf{r_8}$ and $\mathbf{r_9}$. The integrity constraint $\mathbf{r_{15}}$ eliminates answer sets where some nodes are not reached [8].

In order to minimize costs, we add the weak constraint $\mathbf{r_16}$. Here, edges belonging to the cycle are weighted according to their costs and DLVHEX lists optimal answer sets only. For weak constraint syntax see Section 3.9.

4.2.3 Plugin

The external atom &*edge* can be realized in Python as follows.

```
import dlvhex
import networkx as nx # for graph tasks

# get edges from given node in given file
def edges(fileName, currentNode):
  # Sources will be loaded from the file
  g = nx.read_weighted_edgelist(fileName.value().strip('"'),
          nodetype=str, create_using=nx.DiGraph())
  # Output successor nodes of the current node including weight
  for node in g.successors(currentNode.value()):
    weight = g[currentNode.value()][node]['weight']
    # produce one output tuple
    dlvhex.output( (node,int(weight)) )

def register():
  prop = dlvhex.ExtSourceProperties()
```

```
# specify that "edges" produces finite set of output tuples
prop.addFiniteOutputDomain(0)
dlvhex.addAtom("edges",(dlvhex.CONSTANT,dlvhex.CONSTANT),2,prop)
```

This plugin uses the `networkx` Python module for comfortable processing of graphs stored in plain text files. The plugin loads the graph, retrieves successor nodes and returns them as output tuples. Note that this code is for demonstrational purposes only, a more efficient implementation is certainly possible. The "finite output domain" property tells DLVHEX that the instantiation of the external atom will be finite.

Details of the Python plugin API are given in Section 5.

The encoding and plugin load the graph from the file example_4_travel.graph. The file describes the directed weighted graph shown in Figure 2; the first lines of that file as follows.

```
austin boston 2
austin chicago 3
austin detroit 1
boston detroit 2
boston fresno 4
```

4.2.4 Problem Solution

To obtain all optimal answer sets of the above program, and display only the cycle predicate (which contains the solution) execute the following command.

```
$ dlvhex2 example_4_travel.hex --filter=cycle \
        --pythonplugin=example_4_travel.py
```

The DLVHEX solver gives a single optimal answer set as output.

$$\{cycle(austin, boston), cycle(boston, memphis), cycle(detroit, austin),$$
$$cycle(chicago, detroit), cycle(memphis, fresno), cycle(fresno, chicago)\}$$
$$< [11:1] > .$$

Note that only the *cycle* predicate is visible due to the `--filter` commandline option. This makes the answer set easier to read. The cheapest route which satisfies all given constraints is:

Austin → Boston → Memphis → Fresno → Chicago → Detroit → Austin

with the minimum cost of 11.

4.3 Pathfinding Example

The last example is from the group of planning problems and it considers pathfinding for multiple agents.

4.3.1 Problem

Pathfinding for a single agent is the problem of planning a route from an initial location to a goal location in an environment, going around obstacles. Pathfinding for multiple agents also aims to plan such routes for each agent, subject to different constraints, such as restrictions on the length of each path or on the total length of paths, no self-intersecting paths, no intersection of paths/plans,

no crossing/meeting each other. It also has variations for finding optimal solutions, e.g., with respect to the maximum path length, or the sum of plan lengths. These problems are important for many real-life applications, such as motion planning, vehicle routing, environmental monitoring, patrolling, computer games [6]. An ASP formulation for this problem where multiple agents need to find paths from their respective starting locations to their goal locations, ensuring that paths do not collide with static obstacles and that no two agents collide with each other was described in [6]. We extend this formulation by importing the graph using an external atom to consider only parts of a (potentially very large) graph.

The full source code for this example is available at `https://github.com/hexhex/manual/tree/master/example_4_pathfind` (example_4_pathfind.hex, example_4_pathfind.py, and example_4_pathfind.graph).

4.3.2 Encoding

We can encode this problem in the following HEX-program.

$\mathbf{r_1}$: $startingNode(one).$

$\mathbf{r_2}$: $nodeOfDegree(P, 0, P) \text{:-}\ startingNode(P).$

$\mathbf{r_3}$: $nodeOfDegree(F1, DegPlus, F2) \text{:-}\ nodeOfDegree(\_, Deg, F1),$
$\&edges[F1](F2), DegPlus = Deg + 1, \#int(DegPlus).$

$\mathbf{r_4}$: $node(Y) \text{:-}\ nodeOfDegree(X, V, Y).$

$\mathbf{r_5}$: $edge(X, Y) \text{:-}\ nodeOfDegree(X, V, Y).$

$\mathbf{r_6}$: $agent(1).\ agent(2).$

$\mathbf{r_7}$: $start(1, one).\ start(2, four).$

$\mathbf{r_8}$: $goal(1, ten).\ goal(2, eleven).$

$\mathbf{r_9}$: $clear(V) \text{:-}\ node(V), V\ != three$

$\mathbf{r_{10}}$: $path(I, 0, V) \text{:-}\ start(I, V).$

$\mathbf{r_{11}}$: $\{path(I, TP, U) \text{:}\ edge(U, V)\} \text{:-}$
$agent(I), path(I, T, V), TP = T + 1, \#int(TP).$

$\mathbf{r_{12}}$: $\text{:-}\ agent(I), \#int(T), 1 < \#count\{U \text{:}\ path(I, T, U)\}.$

$\mathbf{r_{13}}$: $visit(I, V) \text{:-}\ path(I, T, V).$

$\mathbf{r_{14}}$: $\text{:-}\ goal(I, V), not\ visit(I, V).$

$\mathbf{r_{15}}$: $\text{:-}\ path(I, T, V), path(IP, T, V), X <= XP.$

$\mathbf{r_{16}}$: $\text{:-}\ path(I, T, V), not\ clear(V).$

$\mathbf{r_{17}}$: $\text{:-}\ path(I, T, V), path(I, TP, U), TP = T + 1, not\ \&check[I, U, V]().$

In $\mathbf{r_2}$–$\mathbf{r_5}$ we load part of the graph using the *&edges* external atom which discovers nodes and edges from the external file, limited by the maximum integer value. In $\mathbf{r_6}$–$\mathbf{r_8}$ we represent agents and their start and goal positions. Rule $\mathbf{r_9}$ represents that all vertices except "*three*" are obstacle-free.

Rules $\mathbf{r_{10}}$ and $\mathbf{r_{11}}$ guess a path for each agent, where the agent can visit a new node using an edge, or do nothing. Constraint $\mathbf{r_{12}}$ eliminates all answer

set candidates in which there is more than one vertex visited by a single agent at one time (no agent can be at two different locations at the same time). Rule $\mathbf{r_{13}}$ defines which nodes are visited, using the path. Constraint $\mathbf{r_{14}}$ ensures that each agent reaches its destination node. We ensure that agents do not collide with each other using constraint $\mathbf{r_{15}}$. We also ensure that agents do not go through obstacles using constraint $\mathbf{r_{16}}$.

Constraint $\mathbf{r_{17}}$ represents an external check of the form $\&check[I, U, V]()$ which checks if agent I is able to move from node U to node V. The external atom decides whether that move is valid or invalid, for example because in the abstract representation of the graph certain combinations of properties of agents and edges (that can prevent successful movement) are not represented, e.g., narrow corners, or doors that are too small for certain agents.

Note that by using an external check we can make the planning problem more abstract by creating a sound (efficient) encoding and making it complete using the external check.

4.3.3 Plugin

The external atoms *&edge* and *&check* are realized in Python as follows.

```
import dlvhex
import networkx as nx # for graphs

def edges(currentNode):
  # hardcoded source for graph
  g = nx.read_weighted_edgelist("example_4_pathfind.graph",
        nodetype=str,create_using=nx.DiGraph())
  # output successors
  for node in g.successors(currentNode.value()):
    dlvhex.output( (node,) )

def check(A,V,U):
  if A.value()=='1' and V.value()=='two' and U.value()=='five':
    pass # no path = no tuple = atom false
  else:
    # empty tuple = true, no tuple = false
    dlvhex.output( () )

# Register function
def register():
  prop = dlvhex.ExtSourceProperties()
  prop.addFiniteOutputDomain(0)
  dlvhex.addAtom("edges",(dlvhex.CONSTANT,), 1, prop)
  dlvhex.addAtom("check",
    (dlvhex.CONSTANT, dlvhex.CONSTANT, dlvhex.CONSTANT), 0)
```

See Section 4.2.3 for an explanation of a similar plugin and Section 5 for the Python API.

4.3.4 Problem Solution

Assume we solve the problem using the graph in Figure 3. The graph is again loaded from the external file and not specified as a set of facts. We know that there is an obstacle at node *three*, so any answer set with node *three* in the path is removed.

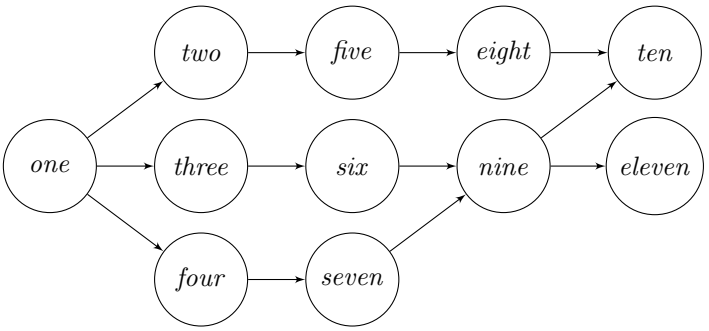

Figure 3: Simulation of the environment used by agents

To obtain all answer sets of the above program, and display only the path predicate (which contains the solution) execute the following command.

```
$ dlvhex2 example_4_pathfind.hex --filter=path --maxint=4 \
        --pythonplugin=example_4_pathfind.py --heuristics=old
```

Note that here we also specify the maximum integer number. If we do not do this, DLVHEX uses the largest integer in the program which is in this case 2, hence without `--maxint` no solution is found. We also specify an evaluation heuristics for performance reasons.

The answer sets of this program are as follows:

$$\{path(1,0,one), path(2,0,four), path(1,1,four), path(2,1,seven),\\ path(1,2,seven), path(2,2,nine), path(1,3,nine), path(2,3,eleven),\\ path(1,4,ten)\}$$

This means that agent 1 follows the path

$$\text{one} \rightarrow \text{four} \rightarrow \text{seven} \rightarrow \text{nine} \rightarrow \text{ten}$$

and at each time agent 2 follows

$$\text{four} \rightarrow \text{seven} \rightarrow \text{nine} \rightarrow \text{eleven}$$

and does not perform a last step.

As the external check excludes the path via *two* and the internal knowledge exludes paths via *three* there is only one answer set.

5 External Interfaces

This section discusses the implementation of external sources. One important design principle was to provide a mechanism to easily add further external atoms, as introduced in Section 3.11, without having to recompile the main application.

Formally, an external atom is defined to evaluate to true or false, depending on a number of parameters:

- An interpretation (a set of atoms)
- A list of input constants
- A list of output constants

However, it is more intuitive and convenient to think of an external atom not as being boolean, but rather functional: depending on a given interpretation and a list of input constants, it returns output tuples for which it is true. For instance, the external atom to import triples from RDF files has the form:

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

where *Uri* stands for a string denoting the RDF-source and X, Y, and Z are variables that represent an RDF-triple (X, Y, Z) from the specified source.

5.1 Information Flow

The interface that is used by DLVHEX to access a plugin follows very closely the previously described "function" semantics. For each predicate (e.g., for $\&rdf$), a retrieval function has to be implemented. This function receives a query object and returns an answer object. The function is always called with ground input parameters. The query object carries the ground input parameters and the input interpretation, while the answer object is a container for the ground output tuples.

5.2 Types of Input Parameters

Previously we said that the semantic function of an external atom may use the complete interpretation for its computation. For practial and for efficiency reasons, often only small parts of the interpretation are used.

This leads to three types of input parameters:

- Constant parameters
- Predicate parameters
- Tuples

A parameter of type *constant* is not related to the interpretation at all, like in the previous example of the RDF-atom where we have a string as a constant input to the external atom.

A parameter is of type *predicate* indicates that all atoms with this predicate in the interpretation are relevant for the semantic evaluation of the external atom. As an example, assume an external atom that calculates the overall price of a number of books given by their ISBN number:

$$\&overallbookprice[isbn](X)$$

The single input parameter of this atom would be of type *predicate*, meaning that not the constant itself is used by the atom's function, but all atoms $isbn(Y) \in I$ with this predicate. Assume the current answer set candidate I is

$$I = \{isbn(\text{“0-19-82183-6”}),\ isbn(\text{“0-201-99954-4”}),\ p(a),\ q(b) \dots\}$$

then the function implementing $\&overallbookprice$ will be called with the following filtered interpretation:

$$I = \{isbn(\text{“0-19-82183-6”}),\ isbn(\text{“0-201-99954-4”})\}$$

where only atoms with predicate *isbn* remain.

A parameter of type *tuple* stands for an arbitrary number of constant input parameters. This is, e.g., useful for string operations like concatenation

$$\&concat[string1, string2, string3, \dots](Out)$$

where the atom is true for the output constant which is the concatenation of all input strings. (Note that for this functionality the interpretation is irrelevant and the function implementing &*concat* will receive $I = \emptyset$.)

Specifying the type of input parameters not only helps to single out the relevant part of the interpretation, but also supports DLVHEX in calculating the dependencies within a HEX-program. Plugins can be implemented either in Python or C++, as shown in the following two subsections.

5.3 Implementing External Atoms in Python

With DLVHEX version 2.4.0, a Python plugin interface was introduced, which supports Python scripts that provide functions to realise custom external atoms.

A Python plugin is a script that starts with `import dlvhex` and contains the following functions:

- a `register` function that registers all external atoms, and
- for each external atom an evaluation function that has the same name as the atom.

5.3.1 Registering External Atoms

The `register` function has the following form:

```
def register():
  dlvhex.addAtom("Atom_Name",(Input_Parameters),Output_Arity)
```

It contains a call of `addAtom` for each external atom. Each call specifies three parameters:

- `Atom_Name` is the name of the external predicate;
- `(Input_Parameters)` is a tuple of arbitrarily many input parameter *types*. Each type is one of `dlvhex.CONSTANT`, `dlvhex.PREDICATE`, or `dlvhex.TUPLE`, corresponding to the concepts introduced in the previous section;
- `Output_Arity` is an integer value representing the output arity of an atom.

Consider the &*concat* external atom introduced in the previous section. It takes an arbitrary number of strings as input and outputs their concatenation. Hence the input type is `dlvhex.TUPLE` and the register function is as follows.[1]

```
def register():
  dlvhex.addAtom("concat", (dlvhex.TUPLE,), 1)
```

If we assume that &*concat* only accepts two strings, the register function would be as follows.

```
def register():
  dlvhex.addAtom("concat", (dlvhex.CONSTANT, dlvhex.CONSTANT), 1)
```

5.3.2 Implementing Semantics of External Atoms

Once an external atom is registered, it has to be implemented in the form of another Python function with an appropriate number of input parameters and output parameters.

[1] Note the extra ",": the Python syntax for creating a tuple with a single element is `(a,)`.

5.3.2.1 Parameter Values (Input)

As introduced in Section 5.2 DLVHEX supports three different input types (constant, predicate, and tuple). The Python function implementing the external atom requires one parameter for each element of the tuple given in `addAtom`. In general an external atom implementation looks as follows.

```
def Atom_Name(Input_Parameter_1, Input_Parameter_2,...):
  Implementation ...
```

Each of the input parameter types is accessed in different way:

- To access data for input parameters of type `CONSTANT`, if the parameter variable is called `var` one obtains its value by writing `var.value()`. Example 5.1 shows how to use a `CONSTANT` parameter.
- To access data for input parameters of type `PREDICATE`, if the parameter variable is `var` then `var.value()` will just give the predicate given as argument. Method `dlvhex.getTrueInputAtoms()` provides a collection of all atoms of relevant predicates that are true in the current candidate. Using `for atm in dlvhex.getTrueInputAtoms():` it is possible to enumerate these atoms. An atom `atm`, for example *edge*(*one*, *four*) corresponds to a tuple (*edge*, *one*, *four*) that can be obtained via `atm.tuple()`. In this tuple, the value of each element again can be obtained using the method `.value()`. Example 5.2 shows how to use a `PREDICATE` parameter.
- To access data for input parameters of type `TUPLE`, if the parameter variable is `var` then this variable will obtain a tuple corresponding with the ground input tuple. For each member `x` of that tuple `var` we obtain its value using `x.value()`. Example 5.3 shows an implementation for `&concat` using a `TUPLE` parameter.

The object returned by the `.value()` method is of type `int` or `str`.[2]

5.3.2.2 Creating Output Tuples

An external atom returns output tuples corresponding to the arity specified in the register function (cf. Section 5.3.1). The DLVHEX command used for output looks as follows:

```
dlvhex.output((Output_Parameter_1, Output_Parameter_2,...))
```

Command to output a tuple containing a single value stored in variable `item` looks as follows.

```
dlvhex.output( (item,) )
```

The output command above returns a constant in a tuple of size one (cf. Example 5.1). If we want for instance to return a tuple of size two containing an integer as second element, the output command will be as follows.

```
dlvhex.output( (node, int(j)) )
```

See also the example in Section 4.2. Note that the output type of the first element can be integer, constant, or string, depending on what is stored in `node`.

[2]The class providing the method `.value()` is of type `dlvhex.ID`, which is a data structure that holds the internal ID of an atom/constant/term within DLVHEX.

5.3.3 Examples

In the following we provide three examples, corresponding to the previously explained input parameter types.

Example 5.1 (download example_5_1.py)**.** This Example uses a constant (string) input parameter. This plugin is used in Example 2.1 to query all direct friends of the person of interest.

```
1  import dlvhex
2  import networkx as nx # for graph tasks.
3
4  # An external atom implementation is similar to a regular function
5  # in Java, C, etc. friendsOfDegree is name of the function which
6  # has one input parameter (personOfInterest). For each
7  # personOfInterest external atom will return its direct friends.
8  def friendsOfDegree(personOfInterest):
9    # graph of the friends will be loaded from the external file
10   g = nx.read_weighted_edgelist("example_2_1.edgelist",
11     nodetype=str,create_using=nx.DiGraph())
12
13   # Take successor nodes of the node of interest.
14   friendList = g.successors(personOfInterest.value())
15   # Output the successor nodes
16   for node in friendList:
17     dlvhex.output( (node,) ) # outputs (tuple of size one)
18
19 # Register function
20 def register():
21   prop = dlvhex.ExtSourceProperties()
22   prop.addFiniteOutputDomain(0) # specify that the graph is finite
23   dlvhex.addAtom("friendsOfDegree", (dlvhex.CONSTANT,), 1, prop)
```

The complete source code for this example is available at `https://github.com/hexhex/manual/tree/master/example_5_1`.

Lines 1–2 import two libraries: every DLVHEX-plugin must import `dlvhex`; and the `networkx` library is needed in this particular example for comfortable graph operations (loading a graph from a file, getting successors of a node).

Lines 8–17 implement the external atom `&friendsOf` which receives the name of a person as string, loads a graph from disk in line 11, and returns all direct friends of the person in the graph in line 17.

Lines 20–23 register `&friendsOf` with a single constant input parameter and a single output parameter, and a futher parameter `prop` which provides meta-information about the atom (required for liberal safety).[3] □

Example 5.2 (download example_4_swim.py)**.** This example shows the Python plugin for external atom &*rq* from the Swimming Example. See Section 4.1 for details of the corresponding program how to run it.

The source code of the plugin is as follows.

```
1  import dlvhex
2
3  RES = { 'ind':'money', 'amalB':'goggles',
4          'altD':'yogamat', 'gansD':'money'}
5
6  # return required resources given swimming locations
```

[3]Advanced Topic: in this case we provide the meta-information that only finitely many friends can be discovered this way, i.e., that the graph is finite.

```
7  def rq(loc_pred):
8    for x in dlvhex.getTrueInputAtoms():
9      arg = x.tuple()[1].value() # get argument
10     if loc_pred == x.tuple()[0]: # check predicate
11       if arg in RES:
12         dlvhex.output( (RES[arg],) )
13
14 # register external atom
15 def register():
16   dlvhex.addAtom("rq", (dlvhex.PREDICATE,), 1)
```

This example uses a predicate input parameter for the external atom `&rq`. This external atom checks requirements for a selected swimming location. In the HEX program the external atom is of the form

$$\&rq[location\_choice](required\_resource)$$

which intuitively evaluates to true whenever a given *location_choice* requires a certain *required_resource*.

From line 7–12 we implement the `rq` function. Line 9 extracts the location from the atom, line 10 checks if the predicate of the current input atom is equivalent to the value of `location_choice` (which specifies the predicate of interest), line 11 checks if we know some requirement and if so line 12 returns that requirements. Lines 15–16 register the external atom `&rq`.

For example, if $goto(amalB)$ is true in the answer set candidate, then one of the `for` loop iterations will contain in `x` the ID of the atom $goto(amalB)$. Then `x.tuple()` is a tuple of IDs of constants *goto* and *amalB*. Using the method `.value()` we obtain the strings of these IDs: `x.tuple()[0].value() = 'goto'` and `inp = x.tuple()[1].value() = 'amalB'`. Accordingly the `if` condition in line 10 is true and line 12 returns `('goggles',)`. This means that the external atom $\&rq[goto](goggles)$ is true. □

Example 5.3 (download example_5_3.py)**.** The third example demonstrates parameter type `dlvhex.TUPLE` which stands for an arbitrary number of constant input parameters. As an example where this is useful we show string concatenation.

```
1  import dlvhex
2
3  # concat has one input parameter of type tuple (=arbitrarily
4  # many constants), which specifies the terms to be concatenated
5  def concat(tup):
6    # start with empty string and sequentialy append all inputs
7    ret = ""
8    for x in tup:
9      ret = ret + x.value()
10
11   # output the final string
12   dlvhex.output((ret, ))
13 # register all external atoms
14 def register():
15   # concat has arbitrarily many input parameters
16   # of type constant (=TUPLE) and its output arity is 1
17   dlvhex.addAtom("concat", (dlvhex.TUPLE, ), 1)
```

The complete source code for this example is available at `https://github.com/hexhex/manual/tree/master/example_5_3`.

This external atom receives a tuple of strings, concatenates them and outputs them as a single string value. Line 7 initializes an empty string variable `ret`,

the loop appends input strings to `ret` and line 12 outputs one tuple with the result. □

5.3.4 Learning additional constraints

This section to be written.

5.3.5 Further Information

More information about methods in the DLVHEX Python API is available online at `http://www.kr.tuwien.ac.at/research/systems/dlvhex/doc2x/group__pythonpluginframework.html`.

5.4 C++

This section to be written.

6 Command Line options

In this section, we briefly describe the meaning of the command line options supported by DLVHEX. Calling DLVHEX without any arguments will show all command line options available. For each option, we indicate whether it requires an argument, and if so, we also describe it. An abstract invocation of DLVHEX looks as follows:

```
$ dlvhex2 [OPTION] FILENAME [FILENAME ...]
```

or for reading the program from standard input

```
$ dlvhex2 [OPTION] --
```

The following set of commands is related with Input, Output and Reasoning options.

	`--`	Parse from standard input.
`-s`	`--silent`	Do not display anything than the actual result
`-f`	`--filter=foo[,bar[,...]]`	Only display instances of the specified predicate(s).
	`--nofacts`	Do not output EDB facts. EDB facts are the facts of the program.
`-n`	`--number=<num>`	Limit number of displayed models to ⟨`num`⟩, Default value is 0, which means to display all models.
`-N`	`--maxint=<num>`	Set maximum integer (# `maxint` in the program takes precedence).
	`--weaksafety`	Skip strong safety check.
	`--strongsafety`	Applies traditional strong safety criteria.
	`--liberalsafety`	Uses more liberal safety condition than strong safety.
	`--mlp`	Use DLVHEX+mlp solver (modular nonmonotonic logic programs).

`--forget`	Forget previous instantiations that are not involved in current computation (mlp setting).
`--split`	Use instantiation splitting techniques.
`--noeval`	Parse the program, but do not evaluate it (only useful with `--verbose`).
`--keepnsprefix`	Keep specified namespace-prefixes in the result.
`--keepauxpreds`	Keep auxiliary predicates in answer sets.

6.1 Plugin Options

`-p`	`--plugindir=DIR`	Specify additional directory where to look for plugin libraries (additionally to the installation plugin-dir and `$HOME/.dlvhex/plugins`). Start with `!` to reset the preset plugin paths, e.g., "`!:/lib`" will use only `/lib/`.

6.2 Performance Tuning Options

`--extlearn[=none,iobehavior,monotonicity,functionality, linearity,neg,user,generalize]`

Learn nogoods from external atom evaluation (only useful with `--solver=genuineii` or `--solver=genuinegi`).

`none`:	Deactivate external learning.
`iobehaviour`:	Apply generic rules to learn input-output behaviour.
`monotonicity`:	Apply special rules for monotonic and antimonotonic external atoms.
`functionality`:	Apply special rules for external atoms which are linear in all predicate parameters.
`linearity`:	Apply special rules for external atoms which are linear in all predicate parameters.
`neg`:	Learn negative information
`user`:	Apply user-defined rules for nogood learning
`generalize`:	Generalize learned ground nogoods to ground nogoods.

By default all options above except "`generalize`" are enabled.

`--supportsets` Exploits support sets for evaluation.

`--evalall` Evaluate all external atoms in every compatibility check, even if previous external atoms already failed. This makes nogood learning more independent of the sequence of external atom checks. Only useful with `--extlearn`.

`--nongroundnogoods` Automatically instantiate learned nonground nogoods.

`--flpcheck=[explicit,ufs,ufsm,aufs,aufsm,none]`

- `explicit`: Compute the reduct and compare its models with the candidate
- `ufs`: Use unfounded sets for minimality checking
- `ufsm`: Use unfounded sets for minimality checking, do not decompose the program for UFS checking.
- `aufs`: Use unfounded sets for minimality checking by exploiting assumptions (default).
- `aufsm`: Use unfounded sets for minimality checking by exploiting assumptions. Do not decompose the program for UFS checking.
- `none`: Disable the check.

`--flpcriterion=[all,head,e,none]`

Defines the kind of cycles whose absence is exploited for skipping minimality checks.

- `all (default)`: Exploit head- and e-cycles for skipping minimality checks
- `head`: Exploit head-cycles for skipping minimality checks
- `e`: Exploit e-cycles for skipping minimality checks
- `none`: Do not exploit head- or e-cycles for skipping minimality checks

`--noflpcriterion` Do no apply decision criterion to skip the FLP check. (equivalent to `--flpcriterion=none`)

`--ufslearn=[none,reduct,ufs]`

Enable learning from UFS checks (only useful with `--flpcheck=[a]ufs[m]`).

- `none`: No learning
- `reduct`: Learning is based on the FLP-reduct
- `ufs (default)`: Learning is based on the unfounded set

`--eaevalheuristics=[always,periodic,inputcomplete,eacomplete,post,never]`

Selects the heuristic for external atom evaluation.

`always`: Evaluate whenever possible.

`periodic`: Evaluate in regular intervals.

`incomplete`: Evaluate whenever the input to the external atom is complete.

`eacomplete`: Evaluate whenever all atoms relevant for the external atom are assigned.

`post`: Only evaluate at the end (default).

`never`: Only evaluate at the end and also ignore custom heuristics provided by plugins.

Except for heuristics "never", custom heuristics provided by external atoms overrule the global heuristics for the particular external atom.

`--ufscheckheuristic=[post,max,periodic]`

Specifies the frequency of unfounded set checks (only useful with `--flpcheck=[a]ufs[m]`).

`post`: Do UFS check only over complete interpretations (default).

`max`: Do UFS check as frequent as possible and over maximal subprograms.

`periodic`: Do UFS check in periodic intervals.

`--modelqueuesize=N` Size of the model queue, i.e. number of models which can be computed in parallel. Default value is 5. The option is only useful for clasp solver.

`--solver=S` Use S as ASP engine, where S is one of DLV, DLVDB, LIBDLV, LIBCLINGO, GENUINEII, GENUINEGI, GENUINEIC, GENUINEGC (`genuineii`=(i)nternal grounder and (i)nternal solver; `genuinegi`=(g)ringo grounder and (i)nternal solver `genuineic`=(i)nternal grounder and (c)lasp solver; `genuinegc`=(g)ringo grounder and (c)lasp solver).

`--claspconfig=C` If clasp is used, configure it with `C` where `C` is parsed by clasp config parser, or `C` is one of the predefined strings frumpy, jumpy, handy, crafty, or trendy.

`-e --heuristics=H` Use `H` as evaluation heuristics, where `H` is one of

`old`: Old dlvhex behavior

`trivial`: Use component graph as eval graph (much overhead)

`easy`: Simple heuristics, used for LPNMR2011

`greedy (default)`: Heuristics with advantages for external behaviour learning

`monolithic`: Put entire program into one unit

`manual:<file>:`	Read "collapse" ⟨ idxs ⟩ share ⟨idxs⟩ commands from ⟨file⟩ where component indices ⟨idx⟩ are from `--graphviz=comp`
`asp:<script>:`	Use asp program ⟨`script`⟩ as eval heuristic
`--forcegc`	Always use the guess and check model generator.
`-m --modelbuilder=M`	
	Use `M` as model builder, where `M` is one of (online,offline).
`--nocache`	Do not cache queries to and answers from external atoms.
`--iauxinaux`	Keep auxiliary input predicates in auxiliary external atom predicates (can increase or decrease efficiency).
`--constspace`	Free partial models immediately after using them. This may cause some models to be computed multiple times. (Not with monolithic.)

6.3 Debugging and General Options

`--dumpevalplan=F`	Dump evaluation plan (usable as manual heuristics) to file `F`.
`-v --verbose[=N]`	Specify verbose category (if option is used without [`=N`] then default is 1):
`1`:	Program analysis informations (including dot-file)
`2`:	Program modifications by plugins
`4`:	Intermediate model generation info
`8`:	Timing information (only if configured with `--enable-benchmark`)
	Values are checked bitwise (sum up values for multiple categories).
`--dumpstats`	Dump certain benchmarking results and statistics in CSV format. (Only if configured with –enable-benchmark.)
`--graphviz=G`	Specify comma separated list of graph types to export as .dot files. Default is none, graph types are:
`dep`:	Dependency Graph (once per program)
`cycinp`:	Graph for analysis cyclic predicate inputs (once per G&C-eval unit)
`comp`:	Component Graph (once per program)
`eval`:	Evaluation Graph (once per program)
`model`:	Model Graph (once per program, after end of computation)

`imodel`: Individual Model Graph (once per model)

`attr`: Attribute dependency graph (once per program)

`--version` Shows version information.

Plugin help for dlvhex-manualevalheuristicsplugin[internal]:

`--manualevalheuristics-enable`

Enable parsing and processing of "`#evalunit(...)`" instructions.

Plugin help for dlvhex-manualevalheuristicsplugin[internal]:

`--query-enable=[true,false]`

Enable or disable the querying plugin (default is disabled).

`--query-brave` Do brave reasoning.

`--query-all` Give all witnesses when doing ground reasoning.

`--query-cautious` Do cautious reasoning.

Plugin help for dlvhex-aggregateplugin[internal]:

`--aggregate-enable[=true,false]`

Enable aggregate plugin (default is enabled).

`--aggregate-mode=[native,ext]`

Enable aggregate plugin (default is enabled).

`native (default)`: Keep aggregates (but simplify them to some basic types).

`ext`: Rewrite aggregates to an external atoms.

`--aggregate-allowaggextcycles`

Allows cycles which involve both aggregates and external atoms. If the option is not specified, such cycles lead to abortion; if specified, only a warning is printed but the models might be not minimal. With `--aggregate-mode=ext`, the option is irrelevant as aggregates are replaced by external atoms (models will be minimal in that case). See examples/aggextcycle1.hex..

Plugin help for dlvhex-strongnegationplugin[internal]:

`--strongnegation-enable[=true,false]`

Enable or disable strong negation plugin (default is enabled).

Plugin help for dlvhex-weakconstraintplugin[internal]:

`--weak-enable[=true,false]`

Enable or disable weak constraint plugin (default is enabled). `--weak-allmodels` Display all models also under weak constraints.

Plugin help for dlvhex-functionplugin[internal]:

`--function-maxarity=<N>`

Maximum number of output terms in functionDecompose.

`--function-rewrite` Rewrite function symbols to external atoms.

Plugin help for dlvhex-choicePlugin[internal]:

`--choice-enable[=true,false]`

Enable choice rules (default is enabled).

Plugin help for dlvhex-conditionalLiteralPlugin[internal]:

`--conditinal-enable[=true,false]`

Enable conditional literals (default is enabled).

Plugin help for dlvhex-pythonplugin[internal]:

`--python-plugin=[PATH]`

Add Python script "PATH" as new plugin.

`--python-main=PATH` Call method "main" in the specified Python script (with dlvhex support) instead of evaluating a program.

`--python-arg=ARG` Passes arguments to Python (sys.argv) (can be used multiple times).

7 Input-related warnings and errors

This section explains the most frequent errors, warnings, and info messages related to inappropriate inputs or command line options. All messages are printed to the standard error stream.

Warning and error messages are prefixed with a number that indicates the verbosity level of the message (see `--verbose` commandline option), this is intended to allow for filtering using `grep` and similar tools.

7.1 Syntax Errors

In this section we consider errors emitted during the parsing and checking of logic programs. These errors include information to ease finding and fixing the problem. Each of the error messages shows a line in which the error appears, followed by the type of the error and a short description.

```
8 unparsed "go :- goto(X)"
8 ----------^
8 GeneralError: Syntax Error: Could not parse complete input!
```

To correct this error, investigate the indicated location and make sure the input conforms to the grammar in Section 3 (e.g., check for missing periods, unmatched parentheses).

7.2 Plugin-related errors

In this section we explain exceptions and errors which may occur while working with external atoms or plugins in which we have implemented desired external atom functionality (e.g. Python or C++ plugin).

The following exception occurs whenever we try to load a plugin from a nonexisting file, either from the Python or C++ file.

```
Exception: nonexisting_file_name.py: no such file
```

`nonexisting_file_name.py` does not exist in the current directory and should be replaced with the right file which contains implementation for the external atom(s) used in HEX-program.

The following error occurs if we do not provide an external atom specification and implementation in the source file but we use it in the HEX-program. The error looks as below:

```
GeneralError: Fatal: did not find plugin atom for predicate
"ext_atom"
```

From the description above we can see that there is no plugin atom for the predicate `ext_atom` in the source file which is passed as parameter from the command line.

Another error occurs if the output arity (i.e., the number of arguments in square or round brackes) of an external atom does not match its specification in the source file. The error is given below:

```
GeneralError: External Atom &rq[ind](C,A) has a wrong
output arity (should be 1)
```

7.3 Safety Checking

If any of the variables used in the HEX-program does not satisfy safety conditions listed below, the program is not safe and an error occurs. We use some examples and explanations from [1] in the following.

7.3.1 Regular Safety

DLVHEX imposes a safety condition on variables in rules. This guarantees that a rule has only finitely many ground instances.

7.3.1.1 Standard, Arithmetic and Comparative Predicates A variable X in an aggregate-free rule is safe if at least one of the following conditions is satisfied:

- X occurs in a positive standard predicate in the body of the rule
- X occurs in a strong negated standard predicate in the body of the rule

- X occurs in the last argument of an arithmetic predicate A and all other arguments of A are safe.

A rule is safe if all its variables are safe.

Example 7.1. Safe rules and Constraints

$$
\begin{aligned}
a(X) &:\text{-}\ not\ b(X), c(X).\\
a(X) &:\text{-}\ X >= Y, node(X), node(Y).\\
a(Y) &:\text{-}\ number(X), \#precc(X, Y).\\
a(Z) &:\text{-}\ number(X), \#succ(X, Y), Z = X + Y.\\
&:\text{-}\ number(X), number(Y), \#mod(X, Y, 2).\\
&:\text{-}\ a(Y), not\ b(Y), not\ c(Y).
\end{aligned}
$$

□

Example 7.2. Unsafe Rules and Constraints

$$
\begin{aligned}
a(X) &:\text{-}\ b(Y).\\
a(X) &\vee\ \text{-}\ a(X).\\
a(X) &:\text{-}\ not\ b(X).\\
a(X) &:\text{-}\ number(Y), X = Y + Z.\\
a(X) &:\text{-}\ number(Y), \#succ(X, Y).\\
&:\text{-}\ not\ number(X), \#succ(X, Y).\\
&:\text{-}\ not\ b(Y).\\
&:\text{-}\ X >= Y, node(X).
\end{aligned}
$$

□

7.3.1.2 Aggregates A variable X appearing in the symbolic set of an aggregate is safe if it does not appear elsewhere outside the aggregate atom and at least one of the following conditions is satisfied:

- X occurs in a positive standard predicate in the symbolic set
- X occurs in a true negated standard predicate in the symbolic set
- X occurs in the last argument of an arithmetic predicate A in the symbolic set and all other arguments of A are safe

All other variables (including guards) appearing in an aggregate atom have to be made safe by some other literal of the body.

Example 7.3. Safe Rules and Constraints with Aggregates

$$
\begin{aligned}
a(X) &:\text{-}\ node(X), \#count\{V : edge(V, X)\} >= 0.\\
a(X) &:\text{-}\ node(X), not\ \#count\{V : edge(V, X)\} = 0\\
&:\text{-}\ \#count\{V : edge(V, Y), not\ edge(Y, V)\} = X, X \geq 2.\\
&:\text{-}\ not\ node(X), \#count\{V : edge(V, Y)\} = X
\end{aligned}
$$

□

Example 7.4. Unsafe Rules and Constraints with Aggregates

$$
\begin{aligned}
&a(X) \text{ :- } not\ node(X), \#count\{V : edge(V,X)\} >= 0.\\
&a(X) \text{ :- } node(X), \#count\{V : edge(V,X)\} >= Z.\\
&a(X) \text{ :- } node(X), \#count\{V : edge(V,Y), not\ edge(V,Y)\} >= 0.\\
&\quad \text{ :- } \#count\{V : edge(V,Y)\} >= 0, X > Y.\\
&\quad \text{ :- } not\ node(X), \#count\{V : edge(V,Y)\} > X.
\end{aligned}
$$

□

7.3.1.3 Arithmetic predicates By evaluating a program with arithmetic predicates it is possible to derive new numeric constants, different from those already occurring in the program. In case of arithmetic rules, this could cause the non-termination of the evaluation so an error message is issued in this case.

Example 7.5. Non finite domain program

$$
\begin{aligned}
&d(0).\\
&d(Y) \text{ :- } d(X), Y = X + 1.
\end{aligned}
$$

□

To safely evaluate this kind of programs an upper integer limit N has to be specified either on the command-line (cf. Section 6) or in the program using

```
#maxint=N.
```

7.3.1.4 Complex Terms Evaluation of a program might not terminate if a complex term occurs in the head of a recursive rule.

Example 7.6. Non finite domain program

$$
\begin{aligned}
&p(0).\\
&p(f(X)) \text{ :- } q(X).\\
&q(X) \text{ :- } p(X).
\end{aligned}
$$

□

Some programs can be safely evaluated even if there are complex terms appearing in the head of a rule. This is the case when all arguments of a functional term are restricted to range over a finite domain thanks to the presence of some other atoms in the body.

Example 7.7. Finite domain program

$$
\begin{aligned}
&p(0). \quad r(0).\\
&p(f(X)) \text{ :- } r(X), q(X).\\
&q(X) \text{ :- } p(X).
\end{aligned}
$$

□

When a program is not recognized to have a finite domain and termination thus cannot be guaranteed, an error is issued.

7.3.2 Strong Safety

By evaluating a program with external atoms it is possible to derive new numeric constants, different from those already occurring in the program. Moreover it is possible to generate a cycle over such a value-inventing external atom. Such a cycle can cause the non-termination of the evaluation by generating (inventing) more and more new constant terms. In such programs the strong safety condition is violated.

An atom $b = \&g[X](Y)$ in a rule r of the program is *strongly safe* if either there is no cyclic dependency over b or every variable in Y occurs also in a positive ordinary atom not depending on b. A program is safe, if every external atom in a rule is strongly safe.

Example 7.8. Consider the following program:

$\mathbf{r_1}$: $p(a)$.
$\mathbf{r_2}$: $q(aa)$.
$\mathbf{r_3}$: $s(Y) :\text{-}\, p(X), \&concat[X, a](Y)$.
$\mathbf{r_4}$: $p(X) :\text{-}\, s(X), q(X)$.

□

It is not *strongly safe* because Y in the cyclic external atom $\&concat[X, a](Y)$ in $\mathbf{r_3}$ does not occur in an ordinary body atom that does not depend on $\&concat[X, a](Y)$. When we run the above program with commandline option `--strongsafety` enabled (cf. Section 6), the following error is generated:

```
GeneralError: Syntax Error: [Rule] is not strongly safe!
Variable [Var] fails strong safety check in rule [Rule].
```

To make $\mathbf{r_3}$ strongly safe we could add an ordinary atom in order to break the cycle. $\mathbf{r_3}$ could be modified as follows:

$$s(Y) :\text{-}\, p(X), \&concat[X, a](Y), q(Y).$$

Adding atom $q(Y)$ makes program strongly safe since Y appears in the body atom which does not depend on $\&concat[X, a](Y)$.

Along with the error message, the affected `[Rule]` and a list of all unsafe variable occurrences `[Var]` are reported. The first action to take usually consists of checking whether variable `[Var]` is actually in the scope of any atom (in the positive body of `[Rule]`) that can bind it. It can also be helpful to check for variables that occur in aggregate elements (cf. Section 3.8) or conditional literals (cf. Section 3.7), it might be necessary to constraint them using additional positive atoms in conditions.

7.3.3 Liberal Safety

Strong domain-expansion safety is overly restrictive, as it also excludes programs that are clearly finitely groundable. To overcome unnecessary restrictions of strong safety, liberal domain-expansion safety (lde-safety) has been introduced [5], which incorporates both syntactic and semantic properties of a program. All lde-safe programs have finite groundings with the same answer sets.

Unlike strong safety, liberal de-safety is not a property of entire atoms but of argument positions of atoms which we call attributes. Intuitively, an attribute is lde-safe, if the number of different terms in an answer-set preserving grounding (i.e. a grounding which has the same answer sets if restricted to the positive atoms as the original program) is finite. A program is lde-safe, if all its attributes are lde-safe [3].

Since the program from Example 7.8 is finitely restrictable, the cycle is "broken" by $q(X)$ in $\mathbf{r_4}$, it is also *liberally safe*. The program can be evaluated successfully while option `--liberalsafety` is enabled (cf. Section 6). The output is as follows.

```
{p(a),q(aa),p(aa),s(aa),s(aaa)}
```

For more details about liberal safety we refer to [5].

References

[1] Robert Bihlmeyer, Wolfgang Faber, Giuseppe Ielpa, Vincenzino Lio, and Gerald Pfeifer, *DLV User Manual*, The DLV Project, April 2009.

[2] Francesco Calimeri, Wolfgang Faber, Martin Gebser, Giovambattista Ianni, Roland Kaminski, Thomas Krennwallner, Nicola Leone, Francesco Ricca, and Torsten Schaub, *Asp-core-2 input language*, 2013.

[3] Thomas Eiter, Michael Fink, Giovambattista Ianni, Thomas Krennwallner, Christoph Redl, and Peter Schüller, *A model building framework for answer set programming with external computations*, Theory and Practice of Logic Programming (2015), To appear, arXiv:1507.01451 [cs.AI].

[4] Thomas Eiter, Michael Fink, Thomas Krennwallner, and Christoph Redl, *Conflict-driven ASP solving with external sources*, CoRR **abs/1210.1649** (2012).

[5] Thomas Eiter, Michael Fink, Thomas Krennwallner, and Christoph Redl, *Domain expansion for ASP-programs with external sources*, Tech. Report INFSYS RR-1843-14-02, Institut für Informationssysteme, Technische Universität Wien, A-1040 Vienna, Austria, September 2014.

[6] Esra Erdem, Doga Gizem Kisa, Umut Öztok, and Peter Schüller, *A general formal framework for pathfinding problems with multiple agents*, Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence, July 14-18, 2013, Bellevue, Washington, USA., 2013.

[7] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub, *Answer set solving in practice*, Synthesis Lectures on Artificial Intelligence and Machine Learning, Morgan and Claypool Publishers, 2012.

[8] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Marius Lindauer, Max Ostrowski, Trosten Schaub Javier Romero, and Sven Thiele, *Pottasco user guide*, 2015.

[9] *Gnu lesser general public license*, `https://www.gnu.org/copyleft/lesser.html`, Last retrieved 2015-07-02.

[10] *Traveling salesperson problem*, https://en.wikipedia.org/wiki/Travelling_salesman_problem, Accessed: 2015-29-07.