Answer Set Programming with External Source Access*

Thomas Eiter[1], Tobias Kaminski[1], Christoph Redl[1], Peter Schüller[2], and Antonius Weinzierl[1]

[1] Technische Universität Wien, Institut für Informationssysteme, Knowledge Based Systems Group, Vienna, Austria
{eiter,kaminski,redl,weinzierl}@kr.tuwien.ac.at
[2] Marmara University, Faculty of Engineering, Department of Computer Engineering, Istanbul, Turkey
peter.schuller@marmara.edu.tr

Abstract. Access to external information is an important need for Answer Set Programming (ASP), which is a booming declarative problem solving approach these days. External access not only includes data in different formats, but more general also the results of computations, and possibly in a two-way information exchange. Providing such access is a major challenge, and in particular if it should be supported at a generic level, both regarding the semantics and efficient computation. In this article, we consider problem solving with ASP under external information access using the DLVHEX system. The latter facilitates this access through special external atoms, which are two-way API style interfaces between the rules of the program and an external source. The DLVHEX system has a flexible plugin architecture that allows one to use multiple predefined and user-defined external atoms which can be implemented, e.g., in Python or C++. We consider how to solve problems using the ASP paradigm, and specifically discuss how to use external atoms in this context, illustrated by examples. As a showcase, we demonstrate the development of a HEX program for a concrete real-world problem using Semantic Web technologies, and discuss specifics of the implementation process.

1 Introduction

The rise of the World Wide Web and a growing trend towards computation in distributed systems has increased the need for accessing external information sources in logic programs. More and more also multiple sources must be accessed, which moreover may be of different kind and provide their information in heterogeneous formats. There is a broad range from light-weight data access (e.g., based on XML, RDF, or relational data repositories) to knowledge-intensive access (e.g., OWL resp. description logic knowledge bases), and from

* This research has been supported by the Austrian Science Fund (FWF) projects P27730 and W1255-N23, and by the Scientific and Technological Research Council of Turkey (TUBITAK) Grant 114E777. The final publication is available at Springer via https://doi.org/10.1007/978-3-319-61033-7_7.

access to information sources that merely provide data (as, e.g., in dictionaries or thesauri), to sources providing computation services that are instantaneously executed (as, e.g., route planning to get from A to B) or may return a result at a later stage of a computation.

The variety of source access with its dynamic aspects poses a challenge for proper modelling and efficient evaluation in the context of declarative programming, where desired computation results are semantically described rather than obtained after running through a prescribed sequence of computation commands. This is in particular true for Answer Set Programming (ASP) [73,77,84], which is a declarative problem solving approach in which a problem is described by the rules of a nonmonotonic logic program such that the answer sets [59] (i.e., specific models) of the program correspond to the solutions of the problem. After computing the answer sets using an ASP solver, the solutions can be extracted from them. Due to the availability of increasingly efficient and expressive such solvers (e.g., smodels [96], dlv [70], ASSAT [75], GRINGO plus CLASP [56, 57]), and WASP [1], the ASP approach has been successfully used for applications in different areas and disciplines, cf. [14, 48]. However, these solvers provide no or only limited support for external information access.[3]

The need for external information access in ASP has been recognized early on and led to theoretical formalisms such as logic programs with generalized quantifiers [38], and later to DLV-EX programs [18] and the more expressive HEX programs [42]. The latter pick up notions in [18,38] and provide a bidirectional interface between a nonmonotonic logic program and other sources, via designated *external atoms*. These atoms abstractly define external predicates whose valuation is determined by external computation. For example, a rule

$$pointsTo(X, Y) \leftarrow \&hasHyperlink[X](Y), url(X)$$

may informally determine pairs (X, Y) of URLs, where X actually links Y on the Web. Here, $\&hasHyperlink$ is an *external predicate* that is associated with an external source; X is the input for the latter and Y is a result returned, which is determined in whatever (computable) way. Notably, the input of external atoms can also comprise predicate names, not only constants; e.g.,

$$pointsTo(X, Y) \leftarrow \&hasAdmissbleHyperlink[X, black\_list](Y), url(X)$$

would be a variant of the previous rule, where $black\_list$ is a predicate that contains URLs which should be excluded from retrieval.

The abstract concept of an external atom has been realized in the open-source software DLVHEX[4] as an API, which provides a suite of external atoms and, by means of a plugin mechanism, allows the user to tailor external atoms for her needs using Python or C++. This makes the system very powerful; depending on the external evaluation cost, HEX programs offer a range of problem solving capacity, from Σ_2^p for polynomial-time external atoms in the ground case to Turing-completeness in general. Moreover, external atoms may return values that do not occur in the program itself (this is known as *value invention*), which

[3] For more information, see the related work section.

[4] `www.kr.tuwien.ac.at/research/systems/dlvhex`

by recursion may in principle lead to an infinite domain; this is in analogy to the infinite universes of existential rules or description logic ontologies, which result from skolemization.

The generic nature of external atoms, which are blackboxes in general, combined with possible predicate input and/or value invention poses a big challenge for the development of an efficient solver for HEX programs. In the last years, a number of advanced methods and techniques have been researched which have led to significant improvements [29–31,34,35]. Furthermore, an open software architecture that supports a flexible plugin mechanism and is easy to use also for non-experts poses a further challenge, which has been addressed in parallel [90].

In this paper, we present in a tutorial style fashion the HEX formalism as well as the DLVHEX system, which constitutes the state-of-the-art solver for HEX programs. At this, we take a user-centric view, where we omit many technical details. Particular attention is payed to the use of HEX programs and DLVHEX for interoperability on the Semantic Web – and indeed the original development of HEX programs was driven by this issue, as a generalization of a concrete combination of rules and ontologies, a topic that emerged as necessary in the Semantic Web Layer cake proposed by Tim Berners-Lee and has led to a stream of works and a plethora of different approaches [26, 83]. While HEX programs support problem solving at different levels of abstraction, we focus here on the basic end-user level where external atoms can be utilized in different ways in order to enrich the problem solving capacity of ASP.

More specifically, the presentation is structured along the following sections:

– In the next section, we give an introduction to the syntax and semantics of answer set programs and HEX programs. The part on ordinary answer set programs is kept deliberately short and compact, as a number of texts exist that provide an ample introduction to the subject, e.g. [5,13,14,39]; see also Section 7 for further pointers. Furthermore, we do not consider the full repertoire of language constructs that is available in ASP, but concentrate on a core part that is sufficient from a conceptual perspective.

– In Section 3 then, we turn to the issue of using HEX programs. We provide a basic methodology to this end, which enhances the methodology of ordinary ASP programs with the use of external atoms; different such uses, for information outsourcing and computation outsourcing, respectively, will be discussed following [47], as well as typical kinds of external sources. Furthermore, we will go over example encodings of two quite diverse HEX application scenarios, viz. RDF graph exploration in the Semantic Web and the AngryHEX agent for playing Angry Birds, which has been suggested as a low-cost AI challenge for developing programs that outperform human capabilities.[5]

– In Section 4, based on [90] we introduce the DLVHEX system, which is an elaborated software platform for designing and evaluating HEX programs. We will present the system architecture, the Python programming interface for developing external atom implementations, and specific annotations of external

[5] `https://aibirds.org/`

source properties that are important for deciding whether a finite portion of the instantiated rules is sufficient for evaluation and moreover, for evaluation efficiency.

– In Section 5 we go over a full-fledged case study in the area of semantic route planning, that is route planning under further semantic constraints. The application program will be developed stepwise, where in each step additional aspects are addressed; code examples show the implementation of simple external sources, and access to an ontology in a lightweight Description Logic through an external atom is illustrated.

– In the subsequent Section 6, we provide an overview of further HEX applications and HEX extensions. Furthermore, we discuss related work, where in particular we compare DLVHEX to the CLINGO system[6] [56, 57], its closest relative.

– We conclude the paper in Section 7 with a brief summary and outlook; pointers to further material and resources can be found in the appendix.

2 ASP and HEX Programs

In this section, we formally introduce the syntax and semantics of HEX programs; for more details and background, see e.g. [29, 42, 43, 94].

2.1 ASP

In this section we briefly introduce ASP and its underlying concepts. For a more detailed introduction see [39].

From Procedural Programming to Logic Programming. In computer science, all students are taught programming in procedural programming languages like Java, C/C++, Python, and many others. The basic building blocks of procedural programming differ a lot from logic programming and providing a full introduction is beyond the scope of this article. The following paragraphs, however, may help bridging the gap. A procedural programming language is about the contents of the machines memory, i.e., bits organized in basic data types of bytes, integers, characters, arrays, and potentially structures as well as objects. Procedural programs modify data using instructions that are executed one after another, i.e., instructions like addition, subtraction for basic bit manipulation; if, else, and switch for conditional checks; various loops for repeating instructions, and functions or methods to organize sequences of instructions.

In contrast to that, logic programming is about a statement being either true or false. A statement itself may be structured or not: a statement can be unstructured like *jaguar_is_an_animal* or structured like *is_bigger_than*(*45*, *42*). A logic program expresses whether a statement is true or false by rules, basically just if-then expressions. In principle, a logic program can only influence whether a statement is true or false, it cannot otherwise modify statements.

[6] A tutorial covering hybrid answer set solving with the CLINGO system can also be found in this volume [67].

Data in logic programs is represented by structured statements, called atoms. An atom is composed of a predicate name (e.g., *is_bigger_than*) and a sequence of terms, e.g., $(42, 45)$. Terms can be simple constants, or again be structured using function symbols. For readability, however, we will ignore function symbols in the most of what follows. Terms and atoms originate from formal logic, specifically from first-order logic, the most prominent logic formalism that is widely used in mathematics.

An alternative view on atoms is based on relational data bases (e.g. SQL): every predicate can be considered the name of a table while its terms are the values of attributes in the table. In that view, a true atom $at(t_1, \ldots, t_n)$ is a tuple $(t_1, \ldots, t_n)$ that is in the table *at* while a false atom is simply not in the table. Hence, logic programming may be seen as a (powerful) form of database querying. Formally, statements are expressions in a relational language or first-order language.

Syntax Statements of the form $relation\_name(t_1, \ldots, t_n)$ where each t_i is a constant or a variable, are formalized as relational languages. A relational language is the set of all statements that can be expressed over a relational signature.

Definition 1 (Relational signature). *A* relational signature *is a tuple* $S = (\mathcal{C}, \mathcal{P}, \mathcal{X})$ *of pair-wise disjoint sets of* constants*,* predicate symbols*, and* variables*, respectively. We assume that predicate symbols* $p \in \mathcal{P}$ *come with an associated* arity $n \in \mathbb{N}$*, denoted by* $p/n \in \mathcal{P}$*.*

Intuitively, a constant denotes something the logic program speaks about, i.e., it denotes an entity like the number 43 or *tomatoes*. Predicate symbols are used to denote relations, e.g., the $\leq$ relation or *is_edible*. Variables may denote any element out of a set of possible candidates. Variables may occur in any place where a constant may occur.

Usually constants in $\mathcal{C}$ are denoted with first letter in lower case while variables in $\mathcal{X}$ are denoted with first letter in upper case.

Definition 2 (Terms, Atoms). *Given a relational signature* $S = (\mathcal{C}, \mathcal{P}, \mathcal{X})$*, an element of* $\mathcal{C} \cup \mathcal{X}$ *is called a* term*. Furthermore, if* $p/n \in \mathcal{P}$ *is a* n*-ary predicate symbol and* $t_1, \ldots, t_n$ *are terms, then* $p(t_1, \ldots, t_n)$ *is an* atom*. The set of all atoms is denoted by* $\mathcal{A}_S$*.*

If the signature S is clear from the context, one also writes simply $\mathcal{A}$ for $\mathcal{A}_S$.

Given the *is_edible* relation and the constant *tomatoes*, one can form the atom *is_edible*(*tomatoes*) to denote that tomatoes are edible. Atoms may contain variables as, e.g., in *is_edible*(X). An atom that contains no variables is called *ground*. Hence, *is_edible*(*tomatoes*) is ground while *is_edible*(X) is not. We likewise say a term is *ground*, if it contains no variable; that is, if it is from $\mathcal{C}$.

Note that *is_edible*(43) also forms an atom, which intuitively is a false statement while *is_edible*(*tomatoes*) intuitively is a true statement. On the other hand, however, a rotten tomato should not be considered edible and we may

even think of a fantasy story where a mythical creature is eating numbers. Statements therefore may be true or false depending on their interpretation.

Formally, an interpretation for logic formulas (an ASP program is a set of a certain kind of formulas) interprets all constants of the logic formula with entities or individuals, which are called the universe of discourse. For classical logic, the universe may be any set, e.g. the set of natural numbers, and an interpretation is then free to interpret the constant *tomatoes* with the number 51.[7] Since this kind of freedom is not always intuitive and not needed, logic programs consider interpretations where the universe is from the set of symbols of the relational signature and each symbol is interpreted by itself, i.e., *tomatoes* is interpreted as *tomatoes* and 43 is interpreted as 43.

Definition 3 (Herbrand Universe, Herbrand Base, Interpretation). *Given a relational signature $S = (\mathcal{C}, \mathcal{P}, \mathcal{X})$, the* Herbrand universe *HU is the set of all ground terms wrt. S and the* Herbrand base *HB is the set of all ground atoms wrt. S. An (Herbrand)* interpretation *is any set $I \subseteq HB$. Here $a \in I$ is read as a is true under I, and false otherwise.*

In the following, we will assume the relational signature to be given implicitly by the program at hand. Readers knowledgeable in formal logic may observe that the given notion of an interpretation is very simplified compared to the usual notion, yet an interpretation in our terms can be easily extended to a first-order logic interpretation I on the universe HU.

Observe that Herbrand interpretations cannot interpret two different constants by the same entity, they implicitly assume that different constants denote different entities. They follow the so-called unique-name-assumption (UNA).

Logic Programs. Logic programs are comprised of rules. A rule expresses that if something holds, other things have to hold, i.e., a rule is simply an if-then expression. The if-part may contain several conditions, some possibly containing negation, which all have to hold while the then-part may also contain several conditions of which at least one has to hold whenever the if-part holds.

Definition 4 (Rule). *A (disjunctive) rule r is of the form*

$$A_1 \vee \ldots \vee A_m \leftarrow L_1 \ldots, L_n, \quad m, n \geq 0 \tag{1}$$

where $A_1, \ldots, A_m$ are atoms and $L_1, \ldots, L_n$ are literals, i.e., an atom or a negated atom, written as **not** *b, where b is an atom. A rule is* ground*, if all atoms occurring in it are ground.*

The intuition of a rule is that: *if* L_1 to L_n all hold, *then* one of A_1 to A_m must also hold. Given an interpretation I, an atom a holds if $a \in I$ while a negated atom **not** a holds if $a \notin I$. The atoms occurring left of the $\leftarrow$ are called the head atoms while the literals occurring right of it are the body atoms. Formally, for a rule r of the form (1), the *head* is the set $head(r) = \{A_1, \ldots, A_m\}$

[7] As the elements of $\mathcal{C}$ need to be interpreted they are thus called *constant symbols* in classical logic.

while the *body* is the set $body(r) = \{L_1, \ldots, L_n\}$. Rules then can be read as "if the whole body holds, some element of the head must hold".

A rule r with empty body, i.e., $body(r) = \emptyset$, is called a *fact* and a rule with empty head, i.e., $head(r) = \emptyset$, is called a *constraint*.

Example 1. Consider the following three rules:

$$day \vee night.$$
$$\leftarrow sunshine, raining.$$
$$sunshine \leftarrow day, \mathbf{not}\ raining.$$

The first rule is a fact which expresses that it is day or night. The second rule is a constraint and expresses that it cannot be the case that both the sun shines and it is raining. The third rule is neither a fact nor a constraint, and it expresses that whenever it is day and not raining, then the sun shines. Observe that all rules are ground.

The **not** in rule bodies is called *default negation* in ASP since atomic pieces of information that are not known to be true are presumed to be false by default. In this way, ASP implements reasoning under the *Closed World Assumption* (*CWA*), where complete knowledge about atomic facts is assumed. For instance, in Example 1, the atom *raining* is not stated as a fact and cannot be derived by any of those rules, therefore *raining* is false by default. Subsequently, the body of the last rule is satisfied if *day* is true.

Rules that contain variables have to be *safe*, i.e., all variables that occur in the rule must also occur in some positive literal of the rule. Effectively, this allows an implementation to ensure that the relevant range of variables is finite whenever the set of constants $\mathcal{C}$ is finite.

Example 2. Consider the following rules:

$$r_1: \quad p(X) \leftarrow q(X,Y), at, \mathbf{not}\ r(X).$$
$$r_2: \quad p(X) \leftarrow \mathbf{not}\ t(Z).$$

Rule r_1 is safe because every variable (X and Y) occurring in it occurs in its positive body, specifically, in $q(X,Y)$. Rule r_2 on the other hand is not safe for two reasons: X occurs not in its positive body and neither does Z. Intuitively, it is not clear which value for X should be chosen once the body of r_2 is true. Letting $p(X)$ hold for all possible values of X, i.e., for the whole universe HU, seems far too much. Likewise, for Z in the negative body. If one lets it range over HU, it expresses that the rule fires unless for every $u \in HU$ it holds that $t(u)$ is true. Intuitively, it thus makes sense to exclude rules that are not safe. From a computational perspective, not-safe rules are also hard to deal with since the universe may be infinite and hence it is impossible to treat each atom individually within finite time.

In the following we consider only safe rules.

Example 3. The following rule expresses that whenever some X is reachable from Y and Y is reachable from Z, then Z is reachable from X.

$$\mathit{reachable}(X, Z) \leftarrow \mathit{reachable}(X, Y), \mathit{reachable}(Y, Z).$$

This rule is not ground as it contains the variables X, Y, and Z. They are variables, because their initial letters are in upper case. Variables occurring in a rule can be seen as implicitly universally quantified, i.e., the if-then statement expressed by the rule has to hold for all X, Y, and Z. Note that the rule is safe, because all variables X, Y, and Z occur in the positive body.

Logic programs are simply sets of rules, formally:

Definition 5 (Logic Program). *A* logic program *is a finite set of rules.*

A program P is ground, if all rules $r \in P$ are ground.

Example 4. The following is a logic program consisting of three rules:

$$\begin{aligned} \mathit{reachable}(X, Y) &\leftarrow \mathit{connection}(X, Y). \\ \mathit{reachable}(X, Z) &\leftarrow \mathit{reachable}(X, Y), \mathit{reachable}(Y, Z). \\ \mathit{not\_reachable}(X, Y) &\leftarrow \mathit{location}(X), \mathit{location}(Y), \mathbf{not}\ \mathit{reachable}(X, Y). \end{aligned}$$

The program can be used to compute all pairs of locations (e.g. in a city) which are not reachable from each other by taking connections in the public transport system alone. For this, facts such as $\mathit{connection}(a, b)$ and $\mathit{location}(a)$ need to be added, representing a concrete problem instance, i.e., a public transport network in a city.

The first rule states that one location is reachable from another one if it is possible to take a direct connection. The second rule computes the transitive closure of the *connection* relation as described in Example 3. In the third rule, default negation is used to obtain all pairs of locations that are not in the transitive closure. Note that for this rule to be safe, the variables X and Y must be bound by the positive atoms $\mathit{location}(X)$ and $\mathit{location}(Y)$. Otherwise, not all variables would occur in a positive body atom. Further note that variables having the same name but occurring in different rules are treated like distinct variables.

Semantics In order to define the semantics of rules and programs, we first need to define when an interpretation satisfies a rule; this in turn depends on the satisfaction of its components. Based on this, answer sets of a program can be defined as special interpretations that satisfy all rules in a program. We first consider the ground case, which can then be naturally lifted to programs with variables.

Satisfaction for ground programs For ground rules, satisfaction for rules is as in classical logic.

Definition 6 (Satisfaction, Model). *An interpretation I* satisfies

- *a ground atom a, denoted $I \models a$, if $a \in I$,*
- *a negated ground atom* **not** *a, denoted $I \models$* **not** *a if $I \not\models a$,*
- *a conjunction $L_1, \ldots, L_n$ of ground literals, denoted $I \models L, \ldots, L_n$, if for each $i \in \{1, \ldots, n\}$ it holds that $I \models L_i$,*
- *a disjunction $A_1 \vee \ldots \vee A_m$ of ground atoms, denoted $I \models A_1 \vee \ldots \vee A_m$, if there exists $k \in \{1, \ldots, m\}$ with $I \models A_k$, and*
- *a ground rule r, denoted $I \models r$, if $I \models body(r)$ implies that $I \models head(r)$, i.e., if all literals in the body hold at least one atom in the head is true.*

An interpretation I is a model *of a ground program P, if $I \models r$ for each rule $r \in P$. A model I is* minimal *if there is no other model $I' \subset I$.*

Given a rule r and an interpretation I, if the body of r holds under I, i.e., if $I \models body(r)$, then the rule r is said to *fire* under I.

The correct semantics of ground rules containing negation was heavily discussed in the past and multiple approaches have been introduced. For programs without negation, however, there was early consensus that the minimal models are most fitting. It best captures the intuition that a rule's head should only hold, if the body of the rule holds.

Example 5. Consider the following program:

$$P = \{\, b. \quad a \leftarrow b. \quad c \leftarrow d. \,\}$$

The interpretation $I = \{a, b, c\}$ is a model of P, i.e., $I \models P$, because I satisfies each rule of P. Note that $I \models c \leftarrow d$, which may not seem intuitive, because the head of the rule is true although its body is not true. The notion of a model is therefore not sufficient to capture the intuitive meaning of this program. The (unique) minimal model $I' = \{a, b\}$ also satisfies all rules of P and for this program, it is close to our intuitive understanding of a rule, namely that its head atom is only there if the body is satisfied. Note that $a \leftarrow b$ fires under I' while $c \leftarrow d$ does not fire under I'.

Example 6. Consider the program $P = \{a \vee b\}$. Clearly, this program has three models, viz. $I_1 = \{a\}$, $I_2 = \{b\}$ and $I_3 = \{a, b\}$, of which intuitively I_1 and I_2 are preferable to I_3 because that model contains an unnecessary atom; however, by the perfect symmetry between a and b in the program, it is not justified to prefer I_1 over I_2 or vice versa. If we add the rule $b \leftarrow a$, for the resulting program

$$P' = \{a \vee b. \quad b \leftarrow a.\}$$

I_1 is no longer a model; in this case, $\{b\}$ is the only intended model.

Answer Sets ASP adopts a multiple models approach, i.e., a given program P can have multiple models that are considered to be correct and these models can be disjoint from each other; this may even be the case if the program does not contain disjunctive rules. Intuitively, an answer set is a model of the program that can be (re-)constructed by rule application. Once a rule is applicable and fires, it has to stay applicable throughout the whole construction and also in the final model.

Example 7. Consider a program with negation as follows:

$$P = \{\, a \leftarrow \textbf{not}\ b. \quad b \leftarrow \textbf{not}\ a. \,\}$$

This program has two minimal models, $I = \{a\}$ and $I' = \{b\}$. Under I the first rule fires while the second does not, while under I' it is the other way round. Answer-set semantics now declares both models to be correct, because each captures the intuitive meaning of the rules: in I the atom a is true and b is false, so the first rule does fire, deriving a and the second rule does not fire, hence not deriving b. Intuitively, I can be reconstructed from P by letting the first rule fire to obtain a, ensuring that it will fire later on fixes b to be false, hence the second rule is not applicable and I is successfully reconstructed. Thus I is an answer set. Considering I', the same holds vice versa, i.e., both I and I' capture the meaning of the rules in P.

In order to define answer sets formally, the notion of a reduct is important. Intuitively, the reduct with respect to an interpretation I and a program P is obtained by removing all rules from P which cannot fire under I.

Definition 7 (FLP-Reduct). *Given a program P and an interpretation I, the* FLP-reduct P^I *of P wrt. I is obtained as follows: delete from P all rules r with $I \not\models r$, i.e., $P^I = \{r \in P \mid I \models r\}$.*

Answer sets of a program P are then defined as follows:

Definition 8 (Answer-Set). *An interpretation I is an* answer set *of P if I is a minimal model of P^I.*

Intuitively, an answer set is such an interpretation which is (re-)constructable under the rules that fire in the interpretation. Due to this, answer sets are also called stable models.

Example 8. Consider the following program P containing a fact and two rules using default negation:

$$\begin{aligned}
&\mathit{restaurant}(\mathit{osteria}).\\
&\mathit{indoor}(\mathit{osteria}) \leftarrow \mathit{restaurant}(\mathit{osteria}), \textbf{not}\ \mathit{outdoor}(\mathit{osteria}).\\
&\mathit{outdoor}(\mathit{osteria}) \leftarrow \mathit{restaurant}(\mathit{osteria}), \textbf{not}\ \mathit{indoor}(\mathit{osteria}).
\end{aligned}$$

Intuitively, the program states that *osteria* is a *restaurant*, and that it is either an *outdoor* or an *indoor* restaurant. Now, we consider all interpretations that satisfy the rules in P, and start with:

$$I_1 = \{\mathit{restaurant}(\mathit{osteria}), \mathit{indoor}(\mathit{osteria})\}.$$

Since I_1 does not satisfy the last rule, the corresponding FLP-reduct P^{I_1} is the following:

$$\begin{aligned}
&\mathit{restaurant}(\mathit{osteria}).\\
&\mathit{indoor}(\mathit{osteria}) \leftarrow \mathit{restaurant}(\mathit{osteria}), \textbf{not}\ \mathit{outdoor}(\mathit{osteria}).
\end{aligned}$$

As the atom $outdoor(osteria)$ is not contained in I_1, the body of the remaining rule is satisfied under I_1, and $outdoor(osteria)$ needs to be true in every model of P^{I_1}. Hence, we can verify that I_1 is a minimal model of P^{I_1}, such that I_1 qualifies as an answer set of P. Analogously, we derive that

$$I_2 = \{restaurant(osteria), outdoor(osteria)\}$$

is an answer set as well. Because $restaurant(osteria)$ must be true in any model of P due to the fact in the program, there is only one remaining interpretation to consider, which is:

$$I_3 = \{restaurant(osteria), indoor(osteria), outdoor(osteria)\}.$$

The FLP-reduct P^{I_3} only contains the fact $restaurant(osteria)$. as none of the two rule bodies in P are satisfied by I_3. Because both $indoor(osteria)$ and $outdoor(osteria)$ could be removed from I_3 while the interpretation would still satisfy P^{I_3}, I_3 is not a minimal model of the FLP-reduct, and thus, not an answer set of P.

Example 9. Next, we consider the following program P, which, in addition to default negation in rule bodies, also employs disjunction in the head of a rule:

$$\begin{aligned}
&restaurant(osteria).\\
&indoor(osteria) \vee outdoor(osteria) \leftarrow restaurant(osteria).\\
&eat(osteria) \leftarrow indoor(osteria), raining.\\
&eat(osteria) \leftarrow outdoor(osteria), \textbf{not}\ raining.
\end{aligned}$$

Accordingly, P encodes that *osteria* is an *indoor* or an *outdoor restaurant* (now, by using a disjunctive head), and that we *eat* there if it is an *indoor restaurant* and it is *raining*, or if it is an an *outdoor restaurant* and it is not *raining*. Again, we check for different interpretations if they are answer sets by constructing the respective FLP-reducts. First, consider the interpretation

$$I_1 = \{restaurant(osteria), indoor(osteria)\}.$$

In the FLP-reduct P^{I_1}, the last two rules are both removed since *osteria* is not an *outdoor restaurant* and I_1 does not contain the atom *raining*, resulting in the reduct:

$$\begin{aligned}
&restaurant(osteria).\\
&indoor(osteria) \vee outdoor(osteria) \leftarrow restaurant(osteria).
\end{aligned}$$

It is easy to see that I_1 is indeed an answer set because it is not a model of P^{I_1} anymore if one of the two atoms is removed from the interpretation. When checking the interpretation

$$I_2 = \{restaurant(osteria), outdoor(osteria), eat(osteria)\}$$

we obtain a reduct P^{I_2} that still contains the last rule as *osteria* is now assumed to be an *outdoor restaurant*:

$$\begin{aligned}
&\mathit{restaurant}(\mathit{osteria}).\\
&\mathit{indoor}(\mathit{osteria}) \vee \mathit{outdoor}(\mathit{osteria}) \leftarrow \mathit{restaurant}(\mathit{osteria}).\\
&\mathit{eat}(\mathit{osteria}) \leftarrow \mathit{outdoor}(\mathit{osteria}), \mathbf{not}\ \mathit{raining}.
\end{aligned}$$

By checking minimality we find that I_2 is another answer set for P. Finally, consider the following interpretation, which also contains the atom *raining*:

$$I_3 = \{\mathit{restaurant}(\mathit{osteria}), \mathit{indoor}(\mathit{osteria}), \mathit{raining}\}.$$

The corresponding FLP-reduct P^{I_3} is identical to P^{I_1}, but now assumes that it is *raining*, which is not supported by any rule or fact, such that I_3 does not represent an answer set. In fact, there are no further answer sets for P. Note that any interpretation containing both $\mathit{indoor}(\mathit{osteria})$ and $\mathit{outdoor}(\mathit{osteria})$ cannot be a minimal model of the respective reduct because the head of the first rule is already satisfied when only one of them is true.

Historically, there are several slightly different notions of a reduct (e.g. the seminal GL-reduct [58, 59], which removes negative literals from rules), but for ASP programs as introduced above, they are equivalent. In fact, there are many quite diverse definitions of answer set, cf. [74], which indicates some intrinsic interest of this notion.

Answer Sets of Nonground Programs The semantics for ground programs can be extended to programs with variables by transforming the latter into an equivalent ground program. This is achieved by substituting each occurring variable with all possible constants. For this, let a *substitution* $\sigma : \mathcal{X} \cup \mathcal{C} \to \mathcal{C}$ be a mapping from terms to constants such that σ is the identity function on constants, i.e., $\sigma(c) = c$ for any $c \in \mathcal{C}$. Given an atom $a = p(t_1, \ldots, t_n)$ the ground atom obtain from applying σ to a, denoted by $a\sigma$ is $p(t_1\sigma, \ldots, t_n\sigma)$. Given a rule r of the form (1), the ground rule obtained from applying σ to r, denoted by $r\sigma$ is $A_1\sigma \vee \ldots \vee A_m\sigma \leftarrow L_1\sigma \ldots, L_n\sigma$.

Definition 9 (Grounding). *The grounding of a rule r, denoted by $\mathit{grnd}(r)$ is the set of all possible substitutions applied to r, i.e., $\mathit{grnd}(r) = \{r\sigma \mid \sigma \text{ is a substitution}\}$. The grounding of a program P is the grounding of each rule, i.e., $\mathit{grnd}(P) = \bigcup_{r \in P} \mathit{grnd}(r)$.*

The answer-sets of a non-ground program P are then simply the answer-sets of $\mathit{grnd}(P)$.

Example 10. Reconsider two of the non-ground rules from Example 4 forming the following program P:

$$\begin{aligned}
&\mathit{reachable}(X, Y) \leftarrow \mathit{connection}(X, Y).\\
&\mathit{reachable}(X, Z) \leftarrow \mathit{reachable}(X, Y), \mathit{reachable}(Y, Z).
\end{aligned}$$

Since P does not contain any constants, we obtain $grnd(P) = \emptyset$. Intuitively, this makes sense because there are no locations for which reachability could be derived. Hence, the only answer set of P is the empty set. We introduce constants into the encoding by extending P in the following way:

$$P' = P \cup \{\ connection(a, b).\quad connection(b, c).\ \}$$

We obtain the grounding of P' by replacing all variables by constants in all possible ways and aggregating the resulting ground rules. The ground program $grnd(P')$ is represented by the following rules:

$$\begin{array}{ll} reachable(a, b) \leftarrow connection(a, b). & reachable(c, b) \leftarrow connection(c, b). \\ reachable(b, a) \leftarrow connection(b, a). & reachable(c, a) \leftarrow connection(c, a). \\ reachable(b, c) \leftarrow connection(b, c). & reachable(a, c) \leftarrow connection(a, c). \end{array}$$

$$\begin{array}{l} reachable(a, b) \leftarrow reachable(a, b), reachable(a, b). \\ reachable(b, a) \leftarrow reachable(b, a), reachable(b, a). \\ reachable(b, c) \leftarrow reachable(b, c), reachable(b, c). \\ reachable(c, b) \leftarrow reachable(c, b), reachable(c, b). \\ reachable(c, a) \leftarrow reachable(c, a), reachable(c, a). \\ reachable(a, c) \leftarrow reachable(a, c), reachable(a, c). \end{array}$$

The resulting program $grnd(P')$ has the single answer set $\{connection(a, b), connection(b, a), reachable(a, b), reachable(b, c), reachable(a, c)\}$, because informally speaking, b can be reached from a, and c from b, with a single connection, and c can be reached from a via b.

Note that the grounding contains many rules that do not fire w.r.t. the mentioned answer set. However, the essential point is that the set of ground rules which is needed for deriving the correct answer set is over-approximated by the grounding step, such that $grnd(P')$ has the same answer set(s) as P'.

Properties of Answer Sets All answer sets satisfy certain properties, of which we present some in the following. First, it holds that each answer set is a minimal model.

Proposition 1. *Given a program P and an answer set A of P, then $A \models P$ and there exists no answer set $A' \neq A$ of P with $A' \subseteq A$.*

From minimality follows that answer sets are incomparable wrt. $\subseteq$. Formally:

Corollary 1. *Given two different answer sets A, A' of a program P, then $A \not\subseteq A'$ and $A' \not\subseteq A$ both hold.*

Given a program P, an interpretation I, and an atom a occurring in P, then a is said to be *supported* in I, if there is a ground rule $r \in grnd(P)$ such that $I \models body(r)$ and $a \in head(r)$. Intuitively, an atom is supported in I, if its presence is supported by the rules that fire in I, i.e., a is contained in the head of a firing rule. A model I is supported, if each atom $a \in I$ is supported in I.

Proposition 2. *Let A be an answer set of a program P, then A is a supported model.*

In Example 7, we have already illustrated that atoms which are not supported should not be derived because there is no necessity for them to appear in a model. However, not all supported models are also answer sets. In fact, answer sets adhere to a stronger property called foundedness, which intuitively excludes positive cycles supporting itself.

Example 11. Consider the program

$$P = \{\, a \leftarrow b. \quad b \leftarrow a. \,\},$$

where there is a cyclic dependency involving the atoms a and b. This program has two models, namely $I = \{a, b\}$ and $I' = \emptyset$. According to our previous observation, only I' should be the intended model as it represents a subset of I, i.e. it is the minimal model. Intuitively, this makes sense because, even though both atoms are supported by a positive rule body under I now, this support is cyclic and hence, not founded by a positive rule body depending on neither a nor b. In other words, we have no reason independent from a and b to believe either of them.

Example 12. Although every answer set is a minimal supported model, the converse does not hold. Consider the following program:

$$P = \{\, a \leftarrow a. \quad a \leftarrow \mathbf{not}\ a. \,\}$$

The interpretation $I_1 = \emptyset$ satisfies the body of the second rule, but not its head, therefore I_1 is not a model of P, i.e., $I \not\models P$. The interpretation $I_2 = \{a\}$ on the other hand satisfies the heads of both rules, therefore $I_2 \models P$. Furthermore, each atom in I_2 is supported by some rule, namely the first one. Thus, I_2 is a supported model and since I_1 is not a model, I_2 is the minimal supported model of P.

Considering answer sets now, we observe that I_1 is not an answer set because it is not an model of P. The reduct wrt. I_2 is $P^{I_2} = \{a \leftarrow a.\}$ and the minimal model of this program is $I' = \emptyset$. Therefore I_2 is not a minimal model of P^{I_2} and hence I_2 is not an answer set of P. In fact, P has no answer sets. Intuitively, the first rule of P is only deriving a from the presence of a while the second rule is contradictory in itself and can only be satisfied if a is true. Together P requires a to hold but gives only a self-cyclic reason for a to hold, which is not enough. Therefore it makes sense for P to have no answer sets.

In conclusion, the notion of answer set is different from the notion of minimal supported model and answer sets have to satisfy more conditions than minimal supported models even. In some sense, answer sets are minimal derivable models, specifically excluding positive self-support.

The computational complexity of finding answer sets that contain no disjunction in any rule heads is **NP**, i.e., under common assumptions, there is no feasible algorithm to construct answer sets. The best known algorithms for constructing answer sets have an exponential run time in the worst case.

Proposition 3 (Computational Complexity: Non-Disjunctive Programs [78]). *Given a ground program P without disjunction, deciding whether P has an answer set is* **NP***-complete.*

If the input program contains disjunction, the complexity rises even further. Formally, the complexity is at the second level of the polynomial hierarchy. This means that an algorithm to construct answer sets of a disjunctive logic program following an **NP**-style guess and check approach would need to solve subproblems that are by themselves **NP**-complete.

Proposition 4 (Computational Complexity: Disjunctive Programs [28]). *Let P be a ground program including disjunction, then deciding whether P has an answer set is Σ_2^P-complete in the worst case.*

Luckily, despite these results, ASP solving works quite well in practice; this is because the worst case is often not encountered in practical problems. For further background and results on the complexity of logic programs, we refer to [23].

Further ASP Constructs The rules presented so far already allow to express many problems, but some conditions are cumbersome to express using rules only. Therefore ASP allows more constructs, mainly for more convenience. One of those constructs are *aggregates* in rule bodies to count or sum over some values. Briefly, an aggregate atom starts with # followed by the name of the aggregate function, e.g., *count*, *sum*, *min*, *max*, *avg*, a collection of aggregate elements $\{t_1, \ldots, t_m : l_1, \ldots, l_n\}$ followed by a relation symbol, e.g., $\leq$, $<$, or$=$ and a term. The aggregate elements $t_1, \ldots, t_m : l_1, \ldots, l_n$ are comprised of terms $t_1, \ldots, t_m$ and literals $l_1, \ldots, l_n$.

Example 13. Assume one wants to count the number of stations in a train network where each station is given by the predicate $station(Name)$. This is possible using rules alone but very inconvenient. An aggregate allows counting directly as follows:

$$num\_stations(C) \leftarrow \#count\{X : station(X)\} = C.$$

Intuitively, the aggregate $\#count\{X : station(X)\} = C$ counts all X which are station names and assigns the number of such to the variable C.

In order to find optimal answer sets, *weak constraints* may be used. Intuitively, a weak constraint is like an ordinary constraint but an answer set may violate the weak constraint incurring a penalty of some specified cost. In the presence of weak constraints, answer sets with lowest cost are considered optimal. A weak constraint is of the form

$$\leftsquigarrow Body.\ [C@L, t_1, \ldots, t_n] \tag{2}$$

where additional cost C at level L is added to the answer sets if $Body$ is satisfied, and $t_1, \ldots, t_n$ are terms. Cost C can be incurred on different priority levels L: cost on higher levels is minimized before cost on lower levels is minimized. The terms $t_1, \ldots, t_n$ serve to count multiple times those same cost, e.g., 3@0, that appear in different rules.

Example 14. Consider some route planner where the duration of a trip should be minimized with highest priority and the number of stops should be minimal, but is less important than the duration.

$$\leftsquigarrow \mathit{trip\_duration}(T).[T@2]$$
$$\leftsquigarrow \mathit{trip\_stop}(X).[1@1, X]$$

For a duration T of a trip, the first rule incurs cost T at level 2. The second rule incurs a cost of 1 at level 1, and in order to count the cost of every stop, the term X is used in $[1@1, X]$. For illustration, assume that the above weak constraints are part of a larger program with three answer sets,

$$A_1 = \{\mathit{trip\_duration}(5), \mathit{trip\_stop}(a)\},$$
$$A_2 = \{\mathit{trip\_duration}(3), \mathit{trip\_stop}(a), \mathit{trip\_stop}(c), \mathit{trip\_stop}(d)\}, \text{and}$$
$$A_3 = \{\mathit{trip\_duration}(3), \mathit{trip\_stop}(e), \mathit{trip\_stop}(d)\}.$$

Then the cost of A_1 are 5@2 and 1@1, the cost of A_2 are 3@2 and 3@1 while the cost of A_3 are 3@2 and 2@1. Higher levels have higher minimization priority, so A_1 is less optimal than A_2 and A_3. Both A_2 and A_3 have the same cost on level 2, so the lower level 1 is used for comparison and here the answer set A_3 has smaller cost. Therefore A_3 is the optimal answer set given the above weak constraints.

Example 15. To illustrate the usage of the terms in weak constraints consider the following programs:

$$P_1 = \{a. \quad \leftsquigarrow a.[3@0, t] \quad \leftsquigarrow a.[4@0, t]\}$$
$$P_2 = \{a. \quad \leftsquigarrow a.[3@0, t] \quad \leftsquigarrow a.[3@0, t]\}$$
$$P_3 = \{a. \quad \leftsquigarrow a.[3@0, t] \quad \leftsquigarrow a.[3@0, o]\}$$

P_1 has one answer set $A = \{a\}$ with cost 7@0. P_2 has the same answer set A with cost 3@0 although both weak constraints are violated. P_3 has the answer set A with cost 6@0, because the different terms lead to 3@0 being present twice, once with t and once with o as term.

For interoperability of different ASP implementations, the ASP language has been standardized in the *ASP-Core-2 input language format* [19] which allows several more constructs like choice rules (e.g. a rule $2 \leq \{a, b, c\} \leq 2 \leftarrow d.$ which expresses that whenever d holds exactly two of a, b, and c have to hold), conditional literals, and queries.

2.2 Important Classes of Logic Programs

Often one can formulate a specific problem without making use of all constructs available in logic programming and it turns out that restricted programs are often easier and faster to evaluate.

Recall that the computational complexity of programs with disjunction is significantly higher than the complexity of programs without it. In some cases,

however, the disjunction can be removed by *shifting* it from the head into the body using negation. Consider a rule $a \vee b \leftarrow c, \mathbf{not}\ d.$ and observe that this rule has the same answer sets as the two rules

$$a \leftarrow c, \mathbf{not}\ d, \mathbf{not}\ b.$$
$$b \leftarrow c, \mathbf{not}\ d, \mathbf{not}\ a.$$

where the disjunction has been shifted into the rule bodies. The intuition behind this shifting is that whenever the original rule fires, one of a or b becomes true, but not both. The latter two rules express this directly making use of negation to avoid that both become true at the same time. Of course, this is only correct, if no other rule has a and b in the head, because otherwise both might be true. Shifting can be done if the program is *head-cycle free* (cf. [7, 70] for a formal definition).

Furthermore, a program P is called *normal*, if each rule $r \in P$ is normal, that is $|head(r)| \leq 1$; thus P is normal if it contains no disjunction at all. The semantics of normal programs is easier to evaluate (cf. Proposition 3) and the minimal models of such programs can be operationally computed. The *immediate consequences operator* $T_P : HB \rightarrow HB$ for a normal program P is an operator on interpretations such that $T_P(I) \mapsto I'$ where $I' = \{head(r) \mid r \in P, I \models body(r)\}$. Intuitively, the operator takes an interpretation and returns the heads of all rules that fire in the given interpretation. Answer sets of a normal logic program P can be characterized as the least fixpoint of the operator applied to the corresponding reduct, formally: I is an answer set if $I = lfp(T_{P^I}(\emptyset))$. The least fixpoint is obtained simply by applying the operator recursively until its result no longer changes.

Example 16. Consider the program

$$P = \{\, a \leftarrow \mathbf{not}\ b. \quad b \leftarrow \mathbf{not}\ a. \quad c \leftarrow a. \,\}$$

and note that it is a normal program. Consider the interpretation $I = \{a, c\}$ which yields the reduct $P^I = \{\, a \leftarrow \mathbf{not}\ b. \quad c \leftarrow a. \,\}$. Applying the immediate consequences operator yields

$$I' = T_{P^I}(\emptyset) = \{a\}; \qquad I'' = T_{P^I}(I') = \{a, b\}; \qquad I'' = T_{P^I}(I'') = \{a, b\}$$

thus $I'' = I$ is the least fixpoint, i.e., $lfp(T_{P^I}) = I$ and consequently I is an answer set of P.

Another important class of logic programs is the class of Horn programs. A logic program is *Horn*, if it is normal and each rule of the form (1) contains in its body only positive literals, i.e., the body is a conjunction of atoms. The complexity of Horn programs is in **P** and thus Horn programs are far easier to evaluate than normal programs. In fact, every Horn program that has a model has a unique minimal model and this model is its (single) answer set.

2.3 HEX Program Syntax

HEX extends ASP by external atoms, that are special atoms to access external information sources. As such, external atoms may only occur in the body of a

rule, since the external source can only be queried for information. To distinguish external atoms from ordinary atoms, the names of external atoms start with the & symbol. The set of *external predicate names* is denoted by $\mathcal{G}$, which is disjoint from the set of terms and variables. A relational signature for a HEX program therefore is a quadruple $S = (\mathcal{C}, \mathcal{P}, \mathcal{X}, \mathcal{G})$.

External atoms may receive as input ordinary terms as well as the extensions of predicates. To specify that an external atom shall receive as input the whole extension of a predicate, the predicate name, i.e., an element from $\mathcal{P}$, is provided as input.

Definition 10 (External Atom). *An* external atom *over a relational signature $S = (\mathcal{C}, \mathcal{P}, \mathcal{X}, \mathcal{G})$ is of the form*

$$\&g[Y_1, \ldots, Y_n](X_1, \ldots, X_m) \tag{3}$$

where $Y_1, \ldots, Y_n$ is a list (called input *list) of terms and predicate names from $\mathcal{C} \cup \mathcal{X} \cup \mathcal{P}$ and $X_1, \ldots, X_m$ is a list of terms from $\mathcal{C} \cup \mathcal{X}$ (called* output *list), and $\&g \in \mathcal{G}$ is an external predicate name. We assume that $\&g$ has fixed lengths $in(\&g) = n$ and $out(\&g) = m$ for input and output lists, respectively.*

In the ground case, the input terms $Y_1, \ldots, Y_n$ intuitively consist of individual constants (e.g. *tomatoes*) and predicate names (e.g. *edge*). An external atom provides a way for deciding the truth value of an output tuple depending on the input tuple and a given interpretation.

Example 17. Consider an external atom $\&concat[X, Y](Z)$ that takes two input constants and returns an output constant representing the string obtained from concatenating the string representations of the two input constants. This external atom depends only on constants from the program with which the external atom is instantiated during grounding. For instance, in the following rule the external atom is called with a first name and a last name, and the full name is retrieved.

$$fullname(Z) \leftarrow \&concat[X, Y](Z), firstname(X), lastname(Y).$$

When grounding the HEX program containing the previous rule as well as the two facts $firstname(bob)$ and $lastname(dylan)$, we obtain a rule that contains the ground instance $\&concat[bob, dylan](bobdylan)$ of $\&concat[X, Y](Z)$. The atom $\&concat[bob, dylan](bobdylan)$ evaluates to true, and $fullname(bobdylan)$ can be derived.

Often terms alone do not suffice as input to an external atom. This is the case whenever the output of an external atom (respectively the truth value of a ground external atom), depends on the extension of one or more predicates in a given HEX program.

Example 18. For instance, suppose we want to retrieve reachability information w.r.t. the transport network from Example 4 via an external atom instead of computing it by means of program rules, e.g. in order to apply a dedicated algorithm.

The external atom $\&reachable[connection, a](X)$ may be devised for computing the nodes which are reachable from node a in a graph represented by atoms of form $connection(u, v)$. In this case, the external atom has a predicate name as well as a constant term as input parameters.

Intuitively, given an interpretation I, $\&reachable[connection, a](X)$ will be true for all ground substitutions $X \mapsto b$ such that b is a node in the graph whose set of edges is $\{(u, v) \mid connection(u, v) \in I\}$, and there is a path from a to b in that graph.

An external atom of the form (3) for which it holds that $n = 0$ is an atom that only imports external information, while an external atom with $m = 0$ imports no information but can be either true or false. Hence, the latter behaves like a Boolean predicate and may be used as an external checker, e.g., to run a specific checking algorithm.

Example 19. Consider an external atom $\&importConnections[](X, Y)$ which returns all connections of some public transport network. Here, we have that $n = 0$ and thus, the evaluation of the external atom does not depend on information derived from the HEX program in which it is used. However, a rule of the form

$$connection(X, Y) \leftarrow \&importConnections[](X, Y), location(X), location(Y).$$

could be used to, e.g., import all connections between locations from a given set into the program from Example 4.

Alternatively, consider the atom $\&distanceLessThan[connection, X, Y, N]()$, which does not have any output parameters, i.e. $m = 0$. Suppose it constitutes a Boolean predicate that evaluates to true if and only if location X has distance less than N from location Y in the transport network represented by the extension of the predicate *connection*. Then, it could be used in a HEX program in a constraint such as

$$\leftarrow \textbf{not } \&distanceLessThan[connection, X, Y, 5](), location(X), location(Y).$$

to ensure that no two locations have distance greater or equal 5 from each other in the network induced by the predicate *connection*.

A *HEX-literal* is either an ordinary literal, an external atom, or a default-negated external atom. Rules in HEX then are exactly like ordinary rules in ASP except that the literals in the body may contain external atoms.

Definition 11 (HEX rule). *A* HEX rule r *is of the form*

$$A_1 \vee \ldots \vee A_m \leftarrow L_1 \ldots, L_n. \tag{4}$$

where all A_i are atoms, and all L_j are either literals or HEX-literals, for $1 \leq i \leq m$, $1 \leq j \leq n$, $m, n \geq 0$.

In the following, we call HEX-rules just rules.

Example 20. Consider an external atom to query a web-based weather report which receives as input a set of pairs of dates and locations one is interested and reports the set of all weather conditions that occur at some of the locations on the specified date as output. Such an external atom might be

$$\&weatherreport[dateLocationPredicate](WeatherConditions).$$

Let *goto* be a predicate containing pairs of days and cities to be visited. Then, the following constraint excludes extensions of the predicate *goto* where bad weather occurs in some city on the day of visit:

$$\leftarrow \&weatherreport[goto](W), badweather(W).$$

Definition 12 (HEX program). *A HEX* program *is a set P of (HEX) rules.*

A rule is *ordinary* if no external atom occurs in it, and a program is ordinary if all its rules are ordinary. The notions of constraint and fact carry over from ordinary rules. In practice, we shall be interested in finite programs only, while theoretically, programs may be infinite.

Example 21 (continued). Consider the following program Π_{goto} to decide on what day to go to which city for planning a city trip, but exclude trips where the (external) weather report indicates that bad weather occurs during the trip.

$$badweather(rain). \qquad badweather(snow).$$
$$goto(1, paris) \vee goto(1, london).$$
$$goto(2, paris) \vee goto(2, london).$$
$$\leftarrow \&weatherreport[goto](W), badweather(W).$$

The facts in the first row state that snow and rain are bad weather, the rules in the second and third line choose a destination for the first and second day, respectively, which can be Paris or London (and possibly the same city for both days), and the constraint excludes extensions of the predicate *goto* such that bad weather is expected on the chosen trip.

2.4 HEX Program Semantics

The semantics of HEX programs generalizes the answer-set semantics of ordinary programs. The notion of a *Herbrand base HB* for HEX is analogous to ordinary ASP, i.e., *HB* is the set containing all ground ordinary atoms and all ground external atoms. The grounding of a rule r, $grnd(r)$, is defined accordingly, and the grounding of P is given by $grnd(P) = \bigcup_{r \in P} grnd(r)$. Unless specified otherwise, the relational signature $S = (\mathcal{C}, \mathcal{P}, \mathcal{X}, \mathcal{G})$ is implicitly given by P, but different from the 'usual' ASP setting, the set $\mathcal{C}$ of constants used for grounding a program is only partially given by the program itself; in HEX, external computations may introduce new constants that are relevant for semantics of the program.

The notion of interpretation for ordinary logic programs naturally extends to HEX programs, where the valuation of external predicates depends only on (i) the valuation of the ordinary predicates and (ii) some external semantics. Formally, we define interpretations of HEX programs as follows.

Definition 13 (HEX interpretation). *An* interpretation relative to *a HEX program P is any subset $I \subseteq HB$ that contains no external atoms.*

Satisfaction of ordinary atoms with respect to an interpretation I is then as usual; for external atoms, we use the notion of an oracle function.

Definition 14 (Oracle Function). *Every external predicate name $\&g \in \mathcal{G}$, has an associated decidable $(n+m+1)$-ary Boolean function $f_{\&g}$, called* oracle function*, which maps each tuple $(I, \vec{y}, \vec{x})$ to either $\mathbf{T}$ or $\mathbf{F}$, where $I \subseteq HB$ is an interpretation, $\vec{y} = y_1, \ldots, y_n$, $n = in(\&g)$, $\vec{x} = x_1, \ldots, x_m$, $m = out(\&g)$, $x_i \in \mathcal{C}$, $y_j \in \mathcal{C} \cup \mathcal{P}$, and $m, n \geq 0$.*

In the following we make the restriction that for any oracle function $f_{\&g}$, interpretation I and input vector $\vec{y}$, there are only finitely many vectors $\vec{x}$ such that $f_{\&g}(I, \vec{y}, \vec{x}) = \mathbf{T}$.

This definition of external atom semantics is very general; indeed an external atom may depend on every part of the interpretation. For practical reasons, external atom semantics is usually restricted so that it depends only on the extension of those predicates in I that are given in the input list.

Example 22 (continued). Suppose that the weather report for *paris* is *sun* on day 1 and day 2 of the trip, and for *london* the forecast indicates *rain* for both days. The oracle function $f_{\&weatherreport}(I, goto, W)$ corresponding to this information evaluates to $\mathbf{T}$ if and only if:

$\{goto(1, london), goto(2, london)\} \subseteq I$ and $W = rain$,
$\{goto(1, london), goto(2, paris)\} \subseteq I$ and $W = sun$ or $W = rain$,
$\{goto(1, paris), goto(2, london)\} \subseteq I$ and $W = sun$ or $W = rain$, or
$\{goto(1, paris), goto(2, paris)\} \subseteq I$ and $W = sun$.

In all other cases, $f_{\&weatherreport}(I, goto, W)$ evaluates to $\mathbf{F}$.

Definition 15 (Satisfaction of External Atom). *An interpretation $I \subseteq HB$ is a* model *of a ground external atom $a = \&g[\vec{y}](\vec{x})$, denoted $I \models a$, if $f_{\&g}(I, \vec{y}, \vec{x}) = \mathbf{T}$.*

The notion of satisfaction for ordinary atoms, literals, rules, and programs carries over directly from disjunctive logic programs.

Given a HEX program P, the *FLP-reduct* P^I of P with respect to $I \subseteq HB$ is the same as for ordinary programs, i.e., P^I is the set of all $r \in grnd(P)$ such that $I \models body(r)$.

Definition 16 (Answer Set of a HEX Program). *An interpretation $I \subseteq HB$ is an* answer set *of a HEX program P if, I is a minimal model of P^I. We denote by $\mathcal{AS}(P)$ the set of all answer sets of P.*

Observe that if P has no external atoms, then the answer sets according to the above definition are exactly the answer sets for ordinary ASP programs. In other words, HEX programs are a conservative extension of disjunctive [59] (resp., normal [58]) logic programs under the answer set semantics.

Example 23 (continued). Suppose that the weather report for *paris* is *sun* on day 1 and day 2 of the trip, and for *london* the forecast indicates *rain* for both days, i.e., $f_{\&weatherreport}(I, goto, W)$ from Example 22 is employed. In this case, $I \models \&weatherreport[goto](sun)$ holds if $I \models goto(1, paris)$ or if $I \models goto(2, paris)$. Moreover, it holds that $I \models \&weatherreport[goto](rain)$ if $I \models goto(1, london)$ or if $I \models goto(2, london)$, and Π_{goto} has one answer set:

$$\{goto(1, paris), goto(2, paris), badweather(snow), badweather(rain)\}$$

If weather reports of both cities are sunny for the two days, i.e., if another oracle function is employed, we obtain three further answer sets where London is visited on the first, the second, or on both days, respectively. Finally if the weather report for both cities is *snow* for days 1 and 2, there is no answer set.

3 Methodology

We next present basic methodology for using HEX to solve declarative problems. At this, applying the methodology presented in this section not only helps in formulating a HEX encoding for a problem at hand, but also has a potential impact on the efficiency of the solving process. In practice, when computing the answer sets of a HEX program, the evaluation of external sources for determining the truth values of external atoms is interleaved with ordinary answer set search. In this way, it is ensured that all answer sets computed for a given HEX program comply with the formal semantics based on oracle functions (which abstract external sources). More details on the evaluation of HEX programs can be found in Section 4.

In Section 3.1 we provide methodology specifically for using external atoms and distinguish typical kinds of external sources. They can be classified as

1. outsourcing of computation,
2. outsourcing of information, or
3. combination thereof.

A primary use case of HEX is the direct usage as a formalism for modeling user applications. Section 3.2 describes several application scenarios with examples.

In each of these scenarios, all types of external sources can be used.

Basic Methodology. HEX is an extension of ASP, therefore all modeling techniques from ASP may also be used in HEX programs. One of the most important examples is the *guess and check paradigm*, where default negation or disjunctive rules are used to generate a superset of the intended solutions (*guessing part*), and constraints are used to eliminate spurious candidates (*checking part*). For instance, if we assume that facts over predicates *node* and *edge* define a graph, then the well-known graph 3-colorability problem can be solved by guessing all possible colorings of the nodes of a graph using the disjunctive rule

$$g\colon\ color(red, X) \vee color(green, X) \vee color(blue, X) \leftarrow node(X), \qquad (5)$$

and eliminating all colorings which assign the same color to adjacent nodes using the constraint

$$c\colon \quad \leftarrow \mathit{color}(C, X), \mathit{color}(C, Y), \mathit{edge}(X, Y). \tag{6}$$

However, unlike in ASP, HEX programs allow for using external atoms in addition. They can occur both in the guessing and in the checking part. In the former case, they may be used to import individuals over which guessing is performed. For instance, one may replace the atom $\mathit{node}(X)$ in the body of rule (5) by $\&\mathit{node}[](X)$ to import the nodes of the graph. In the latter case, external atoms can be used in the body of constraints to check given conditions. For instance, rule c may be replaced by

$$c'\colon \quad \leftarrow \mathbf{not}\ \&\mathit{check}[\mathit{color}, \mathit{edge}](), \tag{7}$$

where $\&\mathit{check}[\mathit{color}, \mathit{edge}]()$ is true if *color* is a valid 3-coloring wrt. *edge* and false otherwise. Here, the external atom $\&\mathit{check}[\mathit{color}, \mathit{edge}]()$ implements a Boolean check, such that no output terms are required. This type of usage is common when external atoms are utilized for external checks.

The *saturation technique* [37] is an advanced modeling technique for solving problems up to Σ_2^P-completeness, by exploiting the subset-minimality of answer sets for checking whether a property holds *for all* guesses in a search space [39]. A typical example is the check if a graph is *not* 3-colorable, i.e., all possible colorings are invalid. Also here, the checking part may employ external atoms.

For more details about ASP modeling techniques we refer to [39, 53].

3.1 Methodology for Using External Atoms

In general, one can roughly distinguish between two main usages of external sources that we call *computation outsourcing* and *information outsourcing*, respectively, and combinations thereof. We stress that this distinction concerns the usage in applications, as both usages are based on the same language constructs. For each of them we will describe some typical use cases that serve as usage patterns for external atoms when writing HEX programs.

Computation Outsourcing means to send the definition of a subproblem to an external source and retrieve its result. The input to the external source uses predicate extensions and constants to define the problem at hand and the output terms are used to retrieve the result, which can in simple cases also be a Boolean decision.

On-demand constraints are of the form $\leftarrow \&\mathit{forbidden}[p_1, \ldots, p_n]()$. They eliminate certain extensions of predicates $p_1, \ldots, p_n$ and are a special case of computation outsourcing, see also the 3-colorability example above. The external evaluation of such a constraint can return reasons for conflicts to the reasoner in order to restrict the search space and avoid reconstruction of the same conflict [30]. This technique avoids explicitly grounding a set of ordinary ASP constraints representing the forbidden combinations and by this, reduces the size of the ground program. On-demand constraints have been used for efficient planning in robotics where external atoms verify the feasibility of a 3D motion [49, 63].

Computations that cannot (easily) be expressed by rules. Outsourcing computations also allows for including algorithms which cannot (easily or efficiently) be expressed by rules. As a concrete example, an artificial intelligence agent for the skills and tactics game *AngryBirds* needs to perform physics simulations [21]. This requires floating point computations which cannot be done by rules in a practical way (this would either come at the costs of very limited precision or a blow-up of the grounding). Therefore, the physics simulations are integrated with game playing rules as external atoms in a HEX program.

Complexity lifting. This is another kind of computation outsourcing that allows for realizing computations with a complexity higher than the complexity of ordinary ASP programs. The external atom serves then as an 'oracle' for deciding subprograms. While for the purpose of complexity analysis of the formalism, it is often assumed that external atoms can be evaluated in polynomial time[8] [50], as long as external sources are decidable, there is no practical reason for limiting their complexity. External sources can also be other ASP or HEX programs, which allows for encoding other formalisms of higher complexity in HEX programs, e.g., *abstract argumentation frameworks* [27].

Information Outsourcing refers, in contrast to computational outsourcing, to external sources which import information, while reasoning itself is done in the logic program.

A typical example can be found in Web resources which provide information for import, e.g., *RDF triple stores* [68] or *geographic data* [82]. More advanced use cases are *multi-context systems*, which are systems of knowledge-bases (*contexts*) that are abstracted to acceptable belief sets (roughly speaking, sets of atoms) and interlinked by *bridge rules* that range across knowledge bases [12] (see also Section 6.1); access to individual contexts has been provided through external atoms [9]. Also sensor data, as often used when planning and executing actions in an environment, is a form of information outsourcing (cf. ACTHEX [6]).

Combinations. It is also possible to outsource computation and information at the same time. A typical example are logic programs with access to Description Logic knowledge bases (DL KB), called *DL-programs* [41]. A DL KB not only stores information, but also provides reasoning services. This allows for interleaving reasoning within the DL KB and the logic program with information that flows across the external atom API in both directions.

3.2 Concrete Application Scenarios

The HEX language can be directly used for modeling a problem at hand and computing its solutions. Note that the problem instance formally consists both of the HEX program and the external sources, but external sources may be reused for different applications if suitable.

The typical procedure when modeling an end user application starts with identifying and realizing the required external sources, followed by writing a

[8] Under this assumption, deciding the existence of an answer set of a propositional HEX program is Σ_2^P-complete.

HEX program which makes use of these external sources. The two steps may be repeated in order to refine the encoding, i.e., while writing the HEX program, the need for further or modified external sources may arise. In some cases, external atoms of other applications can be reused. Some existing plugins are generic and useful for different applications, e.g., string manipulation functions and an interface to RDF triple stores.

We next give concrete application scenarios including HEX example code.

Semantic Web Applications In the context of the Semantic Web, HEX was applied to connect SPARQL and RDF querying with logic programming rules [87]. Moreover, HEX was used for archaeological research in order to combine geographical and cultural knowledge from various ontologies [82], and for adapting user interfaces targeted at elderly and disabled people by combining ontologies about user profiles with rules about potential user interface styles [100].

The following example uses the FOAF (Friend-of-a-friend) RDF schema to return all pairs of nicknames that know each other, as stored in a FOAF RDF datasource such as can be obtained from `www.livejournal.com`.

$$explore(\text{"http://}\langle Nick\rangle\text{.livejournal.com/data/foaf"}). \tag{8}$$

$$triple(S, P, O) \leftarrow \&rdf[What](S, P, O),\ explore(What). \tag{9}$$

$$\begin{aligned} knows(Nick_1, Nick_2) \leftarrow\ & triple(Id_1, \text{"http://xmlns.com/foaf/0.1/knows"}, Id_2), \\ & triple(Id_1, \text{"http://xmlns.com/foaf/0.1/nick"}, Nick_1),\ Nick_1 < Nick_2, \\ & triple(Id_2, \text{"http://xmlns.com/foaf/0.1/nick"}, Nick_2). \end{aligned} \tag{10}$$

$$knows(A, C) \leftarrow knows(A, B),\ knows(B, C). \tag{11}$$

We start with a fact (8) that represents FOAF-URLs of users that we want to explore, where we substitute the nickname for $\langle Nick\rangle$. Rule (9) uses the external atom *&rdf* to retrieve all RDF-triples from the URL instantiated as input argument *What*, i.e., all URLs that we specified in predicate *explore*. The external atom *&rdf* is true for all RDF-triples found in the resource, therefore they are represented in the predicate *triple*. Rule (10) uses the FOAF relations 'knows' and 'nick' to build all pairs of nicknames of people that know each other, and define the predicate *knows* as result. Finally, we obtain the transitive closure of *knows* using rule (11). As a result, we represent all pairs of nicknames who know each other directly or indirectly.

In the above example, the set of URLs to retrieve was given explicitly in the predicate *explore*.

In the following example, a FOAF RDF-graph is explored implicitly by following URLs retrieved via RDF.

$$explore\_to(What, 3) \leftarrow explore(What). \tag{12}$$

$$triple\_at(S, P, O, D) \leftarrow \&rdf[Uri](S, P, O),\ explore\_to(Uri, D),\ D > 1. \tag{13}$$

$$\begin{aligned} explore\_to(U, D_2) \leftarrow\ & D_2 = D_1 - 1, \\ & triple\_at(Id, \text{"http://www.w3.org/2000/01/rdf-schema\#seeAlso"}, U, D_1), \\ & triple\_at(Id, \text{"http://xmlns.com/foaf/0.1/nick"}, Nick, D_1). \end{aligned} \tag{14}$$

$$found(Nick) \leftarrow triple\_at(S, \text{"http://xmlns.com/foaf/0.1/nick"}, Nick, D). \tag{15}$$

To avoid excessive exploration, we limit following URLs in RDF up to a fixed depth. Resources of interest are again assumed to be given as facts of the predicate *explore*. Rule (12) defines *explore_to* for these resources of interest with a fixed exploration depth of 3. In (13) we retrieve RDF triples for resources where the exploration depth is above zero and represent triples together with their exploration depth. To follow links, in (14) we define *explore_to* also for all RDF links that are associated with nicknames in the RDF graph. This indirection decreases exploration depth by one. Finally (15) defines predicate *found* to represent all nicknames found during exploration, independent from the depth.

The RDF examples are available in the repository of the dlvhex manual.[9]

AngryHEX. The annual *AIBirds Competition*[10] is a competition for AI agents based on the popular *Angry Birds*[11] game, which is about using a slingshot to shoot birds of different types at pigs placed on a scene in order to destroy them. The pigs are usually protected by obstacles of different types. The game uses a realistic physics simulation, including gravity and statics. In the competition, agents are given the positions and dimensions of the objects in the scene and need to return the angle and velocity for shooting the next bird.

The *AngryHEX* agent [20] is implemented on top of HEX programs. The basic strategy is to maximize the estimated damage to obstacles and pigs for all possible targets. Plain ASP is ill-suited for this application as the computation involves physics simulation and floating point numbers. Therefore, a HEX program was used to realize the basic strategy including the optimal selection of the target, while low-level numeric computations have been outsourced. The agent participated in the competition since 2012 and ranked second in 2015.

A very simplified example of the tactics layer of AngryHex, which is evaluated for each shot, is shown in Figure 3.2 Intuitively, (16) uses external atom *&shootable* to determine which objects O in the scene can be hit by shooting with trajectory Tr, velocity V, and bird type B, given that the slingshot (which ejects the bird) is located at coordinates Sx, Sy and has width Sw and height Sh, and given that *bb* represents bounding boxes of all objects in

[9] `https://github.com/hexhex/manual/RW2017/rdf/`

[10] `https://aibirds.org`

[11] `https://www.angrybirds.com`

$$
\begin{aligned}
\mathit{shootable}(O, \mathit{Type}, \mathit{Tr}) \leftarrow\ & \&\mathit{shootable}[O, \mathit{Tr}, V, \mathit{Sx}, \mathit{Sy}, \mathit{Sw}, \mathit{Sh}, B, \mathit{bb}](O), \\
& \mathit{birdType}(B),\ \mathit{velocity}(V),\ \mathit{objectType}(O, \mathit{Type}), \\
& \mathit{slingshot}(\mathit{Sx}, \mathit{Sy}, \mathit{Sw}, \mathit{Sh}),\ \mathit{trajectory}(\mathit{Tr}). && (16)\\
\mathit{tgt}(O, \mathit{Tr}) \vee \mathit{ntgt}(O, \mathit{Tr}) \leftarrow\ & \mathit{shootable}(O, \mathit{Type}, \mathit{Tr}). && (17)\\
\leftarrow\ & \mathit{target}(X, \_),\ \mathit{target}(Y, \_), X \neq Y. && (18)\\
\leftarrow\ & \mathit{target}(\_, T_1),\ \mathit{target}(\_, T_2),\ T_1 \neq T_2. && (19)\\
\mathit{target\_ex} \leftarrow\ & \mathit{target}(\_, \_). && (20)\\
\leftarrow\ & \mathbf{not}\ \mathit{target\_ex}. && (21)\\
\mathit{directDmg}(O, P, E) \leftarrow\ & \mathit{target}(O, \mathit{Tr}),\ \mathit{objectType}(O, T),\ \mathit{birdType}(\mathit{Bird}), \\
& \mathit{dmgProbability}(\mathit{Bird}, T, P), \\
& \mathit{energyLoss}(\mathit{Bird}, T, E). && (22)\\
\mathit{exDirectDmg}(O) \leftarrow\ & \mathit{directDmg}(O, \_, \_). && (23)\\
\mathit{nexDirectDmg}(O) \leftarrow\ & \mathbf{not}\ \mathit{exDirectDmg}(O),\ \mathit{objectType}(O, \_). && (24)\\
\mathit{goodObject}(O) \leftarrow\ & \mathit{objectType}(O, \mathit{pig}). && (25)\\
\mathit{goodObject}(O) \leftarrow\ & \mathit{objectType}(O, \mathit{tnt}). && (26)\\
\leftsquigarrow\ & \mathit{nexDirectDmg}(O), \mathit{goodObject}(O). \quad [1@4, O, \mathit{nexDirectDmg}] && (27)\\
\leftsquigarrow\ & \mathit{nexDirectDmg}(O). \quad [1@1, O, \mathit{nexDirectDmg}] && (28)
\end{aligned}
$$

Fig. 1. AngryHex tactics layer (somplified)

the scene. The vision module of the AngryHex client represents the scene in facts of form $\mathit{birdType}(\mathit{Type})$, $\mathit{objectType}(O, \mathit{Type})$, $\mathit{slingshot}(\mathit{Sx}, \mathit{Sy}, \mathit{Sw}, \mathit{Sh})$, and $\mathit{bb}(O, \mathit{Type}, X, Y, \mathit{Width}, \mathit{Height}, \mathit{Angle})$ where O is a unique object ID. Moreover, possible velocities (a set of integers) and trajectories (either *low* or *high*) are present as facts. External atom *&shootable* extracts the extension of the *bb* argument, builds a 2D representation of the world in the Box2D library,[12] and simulates the shot specified in arguments $O, \ldots, B$. If the shot hits the object, the atom is true for that object.

Rule (17) guesses whether a shootable object shall be the target of the next shot, and (18)–(21) ensure that a single target is chosen. Rule (22) represents direct damage to objects that are hit by the shot, using background knowledge about damage probability and energy loss (disadvantage) of the bird type with respect to the object type. Presence and absence of direct damage is represented in (23)–(24).

Objects that are of type *pig* or *tnt* (explosive blocks) are defined as 'good' objects to hit in (25)–(26), and weak constraint (27) incurs a cost of 1 with priority level 4 for each good object that does not obtain direct damage. Moreover weak constraint (28) incurs a cost of 1 for each object that does not obtain direct damage, however with a lower priority (1) than for good objects.

Recall from (2) that weak constraints are of form $\leftsquigarrow \mathit{Body}.\ [C@L, \ldots]$ and add cost C at level L to answer sets that satisfy Body. Answer sets with lowest cost are considered optimal and minimizing cost on higher levels has priority.

[12] `http://box2d.org/`

The full encoding of AngryHex uses several more external atoms, for example $\&next[O, Tr, Sx, Sy, Sw, Sh, bb](Idx, O')$ is true for a set of pairs $\langle Idx, O' \rangle$ that represent the sequence of objects that a bird shot at object O with parameters $Tr, \ldots, Sh$ would pass through: O' is the Idx'th object hit in the trajectory. Another external atom is $\&firstbelow[O, bb](O')$, which yiedls true for pairs of objects O, O' such that O would hit O' before hitting any other object when falling down. These and further external atoms are used to select the target and trajectory that will inflict the most useful direct and indirect damage to all objects of the scene. The AngryHex project is publicly available.[13]

Route planning. While many commercial and free route planning applications exist (Google Maps is currently perhaps the most popular), the supported query types are usually limited. In contrast, an implementation in HEX programs allows for an easy addition of side constraints and thus tailoring to very specific settings. As a concrete use-case, [32] considered tours with multiple stops (e.g. at shops, a pharmacy, kindergarden, etc) using an external source that supports only point-to-point queries. Side constraints may include restrictions on the order of stops, the tour length, or opening hours at the stops.

Related to route planning is a trip planning scenario. When planning a holiday trip with multiple stops, the order of the stops is often irrelevant, but one wants to spend a certain number of days at each location. However, due to shifts of the dates, the overall price often differs significantly with different sequences. In addition to the sequence of the locations, also other considerations affect the price. E.g. instead of a multi-stop flight through all locations, one may book a return flight to one of them plus local flights from there to the others; sometimes special offers for two-way-tickets make this more attractive. A logic program can automatically generate flight plans according to the constraints and enquire their ticket prices by an external atom that internally uses an online flight booking service. An additional weak constraint can select the cheapest.

Our case study in Section 5 provides details for route planning with HEX.

Description Logic Programs. *Description logics (DLs)* provide a logical formalism for ontologies that are well-suited for the Semantic Web [64] or in medical applications [65]. Ontologies represent classes of objects, referred to as *concepts*, and the relations between objects, called *roles*. Concepts and roles correspond to unary and binary predicates in first-order logic, respectively. A *description logic knowledge base* consists of a *Tbox* (*the terminology*) that defines concepts and roles and represents relations between them, and an *Abox* (*assertions*), that contains specific information on membership of individuals in concepts resp. of pairs of individuals in roles.

Example 24. Suppose *PhDStudent*, *Student* and *Professor* are concepts and *isAssistantOf* is a role. The Tbox may contain the *concept inclusion axiom* $PhDStudent \sqsubseteq Student$, which states that the class of PhD students is a subclass of all students. The Abox contains concept membership assertions like

[13] `https://github.com/DeMaCS-UNICAL/Angry-HEX`

Professor(*smith*) and *PhDStudent*(*johnson*), representing that *smith* is a professor and *johnson* a PhD student. An assertion *isAssistantOf*(*johnson*, *smith*) states that *johnson* is an assistant of professor *smith*. □

Typical reasoning tasks over description logic knowledge bases include concept and role retrieval, i.e., listing all individuals or pairs of individuals which are members of a given concept or role, respectively. In the example above one may ask for all members of *Student* and expects as answer *johnson* as he is a *PhDStudent* and thus, by the terminological knowledge, also a *Student*.

Combining ontologies and answer set programming is especially valuable as existing domain knowledge can be accessed from logic programs. To this end, *DL-programs* have been developed by [40,41] which have been implemented on top of HEX programs with dedicated external atoms; where the external source features external atoms for concept and role queries. Prior to query evaluation, concepts and/or roles are enriched by the contents designated unary resp. binray predicates that occur in the ASP program. This allows for advanced reasoning tasks such as terminological default reasoning or closed world reasoning on description logic knowledge bases [24].

As description logics are monotonic, default reasoning can only be realized by the (cyclic) interaction of rules and the DL knowledge base. To this end, appropriate encodings and an implementation were developed [24]. DL-programs have, e.g., been applied in complaint management for e-government [101].

Our case study in Section 5 contains a code example and a walkthrough for integrating DL reasoning with logic programming using the HEX formalism.

4 The DLVHEX System

In this section we present the DLVHEX system[14] for evaluating HEX programs. The system is implemented in C++ and available as open-source software for all major platforms (Linux, OS X, Windows). Pre-compiled binaries are also provided. External sources are implemented using a plugin interface, which is currently available for C++ and Python.

At the beginning of the DLVHEX project, the system focused on applications for the Semantic Web. Early versions of the system were based on dlv[15] and extended it with higher-order and external atoms. Higher-order atoms allow for using variables in place of a predicate symbol, such as in the rule $C(X) \leftarrow \mathit{subClassOf}(D, C), D(X)$ to model a general subclass relation; while they are still supported, they were less emphasized in later versions as they can be compiled away. External atoms were introduced for accessing arbitrary external sources and are a generalization of DL-atoms, which allow only for interfacing a description logic reasoner.

The first evaluation algorithms used dlv as a blackbox backend for single-shot evaluation of ordinary ASP programs. In a nutshell, the traditional HEX-algorithm translates the HEX-program into an ordinary ASP program which

[14] `http://www.kr.tuwien.ac.at/research/systems/dlvhex`
[15] `http://www.dlvsystem.com`

guesses the values of external atoms (disregarding the actual semantics), evaluates this ASP program using the backend, and performs for each answer set a post-check to ensure that the guesses were correct and that minimality wrt. the FLP-reduct is given. As this approach did not scale to real applications, the evaluation algorithms were improved over time, which required a tighter integration with the backend (such as separate access to the grounding and the solving component of the backend, a callback interface, etc). In context of these improvements, the default backend was replaced by GRINGO and CLASP from the Potassco suite[16]; the original system name DLVHEX was kept and should now be read as *Datalog with disjunctions, higher-order and external atoms*. However, our interface allows for the integration of further solver backends. For instance, in order to make the system self-contained and for testing purposes, we further provide as another alternative an internal grounder and solver, which do not rely on any third-party components. Also dlv is still supported as an alternative to GRINGO and CLASP (used with the traditional algorithms).

We will first discuss the basic evaluation approach and the system architecture, before we switch to the user perspective and point out system features which distinguish DLVHEX from ordinary ASP solvers and also from previous versions. However, since this paper focuses on the usage of HEX-programs rather than evaluation algorithms, we refer to [89] for details.

4.1 Evaluation Approach and System Architecture

In practice, external sources are evaluated wrt. truth assignments computed by the employed ASP solver. Hereby, the information that can be gained from external evaluations depends on the semantic properties of external sources and the extent of the solver assignment at the point of evaluation. Because a solver only assigns truth values to a subset of the Herbrand base during model search, we explicitly represent truth valuations of ground atoms by means of assignments $\mathbf{A}$. An *assignment* $\mathbf{A}$ is a consistent set of literals of form $\mathbf{T}a$ or $\mathbf{F}a$, where a is an atom which is said to be *true* in $\mathbf{A}$ if $\mathbf{T}a \in \mathbf{A}$, *false* if $\mathbf{F}a \in \mathbf{A}$, and *undefined* otherwise. We say that $\mathbf{A}$ is *complete* over a program P if for all atoms a in $grnd(P)$ either $\mathbf{T}a \in \mathbf{A}$ or $\mathbf{F}a \in \mathbf{A}$ holds. The interpretation I corresponding to a complete assignment $\mathbf{A}$ is $I = \{a \mid \mathbf{T}a \in \mathbf{A}\}$.

Traditionally, ground HEX programs have been evaluated by replacing each external atom $\&e[\vec{p}](\vec{c})$ by an ordinary atom $e_{\&e[\vec{p}]}(\vec{c})$ and introducing a rule $e_{\&e[\vec{p}]}(\vec{c}) \vee ne_{\&e[\vec{p}]}(\vec{c}) \leftarrow$ to guess its truth value; the resulting program is evaluated by an ordinary ASP solver (such as GRINGO and CLASP) to produce model candidates. Since the ordinary ASP solver is not aware of the actual semantics of external atoms, each candidate $\mathbf{A}$ is subsequently checked by testing (i) if the external atom guesses are correct, i.e., if $\mathbf{A} \models e_{\&e[\vec{p}]}(\vec{c})$ iff $\mathbf{A} \models \&e[\vec{p}](\vec{c})$ for all external atoms $\&e[\vec{p}](\vec{c})$, and (ii) if assignment $\mathbf{A}$ is a *subset-minimal* model of $f\Pi^{\mathbf{A}}$. If both conditions are satisfied, an answer set has been found. However, this approach did not scale well because there are exponentially many independent guesses in the number of external atoms in the ground program.

[16] `https://potassco.org`

To overcome the problem, novel evaluation algorithms based on *conflict-driven* techniques have been introduced [30]. As in ordinary ASP solving, the input program is translated to a set of *nogoods*, i.e., a set of literals which must not be true at the same time. Given this representation, techniques from SAT solving are applied to find an assignment which satisfies all nogoods [57]. Notably, as the encoding as a set of nogoods is of exponential size due to *loop nogoods* which avoid cyclic justifications of atoms, those parts are generated only on-the-fly. Moreover, additional nogoods are learned from conflict situations, i.e., violated nogoods which cause the solver to backtrack; this is called *conflict-driven nogood learning*.

The extension of this algorithm towards the integration of external sources into the learning component works as follows. Whenever an external atom $\&e[\vec{p}](\vec{c})$ is evaluated under an assignment $\mathbf{A}$ in the checking part (i), the actual truth value under the assignment becomes evident. Then, regardless of whether the guessed value was correct or not, one can add a nogood which represents that $e_{\&e[\vec{p}](\vec{c})}$ must be true under $\mathbf{A}$ if $\mathbf{A} \models \&e[\vec{p}](\vec{c})$ or that $e_{\&e[\vec{p}](\vec{c})}$ must be false under $\mathbf{A}$ if $\mathbf{A} \not\models \&e[\vec{p}](\vec{c})$. If the guess was incorrect, the newly learned nogood will trigger backtracking, if the guess was correct, the learned nogood will prevent future wrong guesses.

Example 25. Suppose $\&diff[p,q](X)$ computes the set difference between the extensions of predicates p and q and that it is evaluated under $\mathbf{A} = \{\mathbf{T}p(a), \mathbf{T}p(b), \mathbf{F}q(a), \mathbf{T}q(b)\}$ with Herbrand universe $\mathcal{C} = \{a, b\}$. Then one can add the nogood $\{\mathbf{T}p(a), \mathbf{T}p(b), \mathbf{F}q(a), \mathbf{T}q(b), \mathbf{F}e_{\&diff[p,q]}(a)\}$ in order to learn that $\mathbf{A} \models e_{\&diff[p,q]}(a)$, i.e., whenever $p(a), p(b), q(b)$ are true and $q(a)$ is false, then $\&diff[p,q](a)$ must not be false. Conversely, one can learn that $\mathbf{A} \not\models \&diff[p,q](b)$ by adding nogood $\{\mathbf{T}p(a), \mathbf{T}p(b), \mathbf{F}q(a), \mathbf{T}q(b), \mathbf{T}e_{\&diff[p,q]}(b)\}$.

Experimental results show a significant, up to exponential speedup [35]. This is explained by the exclusion of up to exponentially many guesses by the learned nogoods.

The system architecture is shown in Figure 2. The arcs model both control and data flow within the system. The evaluation of a HEX program works as follows. First, the input program is read from the file system or from standard input and passed to an *evaluation framework* ①, which may partition the input program depending on the chosen evaluation heuristics. This results in a number of acyclically interconnected *evaluation units*, which can be evaluated independently and interplay only by their input and output interpretations. While this interplay of the units is managed by the evaluation framework, the individual units are handled by *model generators* of different kinds depending on the different program classes. Each instance of a model generator takes care of a single evaluation unit, receives *input interpretations* from the framework (which are either output by predecessor units or come from the input facts for leaf units), and sends output interpretations back to the framework ②, which manages the integration of these interpretations to final answer sets.

Internally, the model generators make use of a *grounder* and a *solver* for ordinary ASP programs. The architecture of our system is flexible and supports multiple concrete backends which can be plugged in. Currently it supports

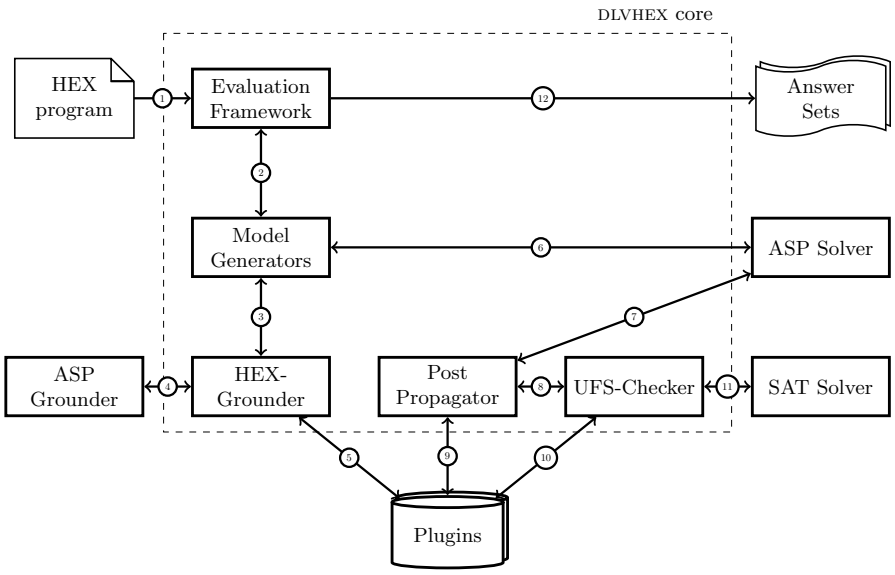

Fig. 2. Architecture of DLVHEX

GRINGO and CLASP, dlv, and an (unoptimized) internal grounder and solver, which serve mainly as a fallback option and for testing purposes. The reasoner backends GRINGO and CLASP are statically linked to our system, thus no interprocess communication is necessary. The model generator within the DLVHEX core sends a non-ground program to the HEX-grounder, and receives a ground program ③. The HEX-grounder in turn uses an (intelligent) ordinary ASP grounder (e.g. GRINGO, dlv's grounder, etc) as submodule ④ and accesses external sources to handle value invention, i.e., values returned by external sources that do not occur in the input program ⑤. The ground-program is then sent to the solver and answer sets of the ground program (i.e. candidate compatible sets) are returned ⑥. Note that the grounder and the solver are separated and communicate only through the model generator, which is in contrast to previous implementations of DLVHEX where the external grounder and solver were used as a single unit (i.e., the non-ground program was sent and the answer sets were retrieved). Separating the two units became necessary because the DLVHEX core needs access to the ground-program in order to obtain important structural information (e.g. cyclicity) for optimization purposes.

The solver backend makes callbacks to the *post propagator* in the DLVHEX core once a model has been found or after deterministic propagation has been finished. During the callback, a complete or partial model is sent from the solver backend to the post propagator, and learned nogoods are sent back to the external solver ⑦. In case of CLASP as backend, we exploit its SMT interface, which was previously used for the special case of constraint answer set solving. The post propagator performs checks to eliminate spurious answer set candidates, which requires calls to the *plugins*, which implement the external

sources. The input list is sent to the external source and the truth values and possibly user-defined learnt nogoods are returned to the post propagator ⑨. Moreover, the post propagator also sends the (complete or partial) model to the *unfounded set checker (UFS checker)*. UFS checking is one possible realization of minimality checking wrt. the reduct. While foundedness (cf. Section 2) means that each true atom is supported by some rule, this additional step is necessary to exclude self-justified atoms due to cyclic dependencies. While the ordinary ASP solver already performs such a check, it does not know the semantics of external sources and thus cannot detect all unfounded sets, which makes an additional check necessary. For this, the UFS checker employs a SAT solver ⑪, which can either be CLASP or the internal solver. More precisely, a specific SAT instance depending on the current answer set candidate and the semantics of the external atoms wrt. this candidate is constructed, such that the models of this instance correspond to unfounded sets. In order to consider the semantics of external atoms during UFS detection for constructing the SAT instance, it needs to call the external sources ⑩. The UFS checker possibly returns nogoods learned from unfounded sets to the post propagator ⑧. The post propagator sends all learned nogoods back to the ASP solver. This makes sure that eventually only valid answer sets arrive at the model generator ⑥.

Finally, after the evaluation framework has built the final answer sets from the output interpretations of the individual evaluation units, they are output to the user ⑫.

For more details we refer to [89].

4.2 Using the DLVHEX System

The system is provided as a command-line tool called `dlvhex2` which expects as only mandatory parameter the filename of the HEX program to evaluate (or `--` to read from standard input). Plugins are loaded from a global plugin directory where they need to be installed before. Thus, the simplest possible call is of form `dlvhex2 prog.hex` where `prog.hex` refers to a program.

However, the system provides numerous command-line options to customize the reasoning process. They include technical options such as the possibility to load plugins from custom locations (e.g. `--plugindir=$HOME/myplugin`), options for customizing the output such as to project answer sets to certain predicates (e.g. `--filter=p`) or restrictions of the maximum number of answer sets to compute (e.g. `-n=7`), and options for tuning the reasoning algorithms; the latter may be used to select heuristics and reasoning techniques based on the problem to be solved. For an exhaustive overview of the usage of the system and its command-line options, we refer to its manual [46]. The system also provides online help, which can be retrieved by calling `dlvhex2 -h`.

In the following we focus on recently added features which distinguish the DLVHEX system from other similar systems and from earlier versions.

While previous releases were mainly prototypes for empirically evaluating algorithms and research results, recent releases also aim at practical applicability of the system for implementing real applications. To this end, important system features have been added to improve the overall user's convenience by

simplifying its usage, to speed up the evaluation, and in order to reduce syntactic restrictions. The enhancements can be organized in two main groups: (i) **usability and system features**, including a novel convenient programming interface for providers of external sources and the integration of support for popular ASP extensions and interoperability, and (ii) enhancements based on **exploiting external source properties** towards *scalability boosts* and increased *language flexibility* based on *liberal safety*, which is a safety criterion that is less restrictive than previous notions of safety. We describe these features in the following.

4.3 Usability and System Features

In this section we present recent work on the system side to improve the user's convenience. We start with general remarks on the DLVHEX software and its dissemination. DLVHEX was previously only available in source format (released under GNU LGPL) and only for Linux platforms. This deployment method turned out to be inconvenient for ASP programmers who want to use the system as is without custom modifications, We thus now provide pre-built binaries for all major platforms (Linux-based, OS X and Windows) in addition. We further created an online demo of the system under `http://www.kr.tuwien.ac.at/research/systems/dlvhex/demo.php` which allows for evaluating HEX programs directly in the browser (the user may specify both the logic program and custom Python-implemented external atoms in two input fields). The demo comes with a small set of examples to demonstrate the main features of the KR formalism. We further provide a manual to support new users of the system [46].

Next, the following two subsections give an overview of the new Python programming interface and interoperability of the system.

Python Programming Interface With earlier versions of the system, users who wanted to integrate custom external sources had to write plugins in C++. While this was natural as the reasoner itself is implemented in C++, it was cumbersome and introduced development overhead even for experienced developers. This is because multiple configuration, source and header files need to be created even when realizing only a small and simple plugin. Also the compilation and linking overhead during development and debugging was considered inconvenient.

As a user-friendly alternative, DLVHEX 2.5.0 introduces a plugin API for Python-implemented external sources. A plugin consists of a single file (unless the user explicitly wants to use multiple files), which imports a dedicated `dlvhex` package and specifies a single method for each external atom. Thanks to higher-level features of Python and modern packages, this usually results in much shorter and simpler code than with C++-implemented plugins. A central `register` method exports the available external atoms and (optionally) their properties to DLVHEX.

Example 26. The following snippet implements $\&diff[p,q](X)$ for computing the values X which are in the extension of p but not in that of q.

```
import dlvhex

def diff(p,q):
  for x in dlvhex.getTrueInputAtoms():    # for all true input atoms
    if x.tuple()[0] == p:                 # is it of form p(c)?
      if dlvhex.isFalse(dlvhex.storeAtom(# is q(c) false?
            (q, x.tuple()[1]))):
        dlvhex.output((x.tuple()[1], )); # then c is in the output

def register():
  dlvhex.addAtom("diff", (dlvhex.PREDICATE, dlvhex.PREDICATE),
    1, prop)
```

The following example illustrates the usage of an external atom in a HEX program, for which the corresponding Python plugin is created subsequently.

Example 27. Consider the program

$$\Pi = \left\{ \begin{array}{l} r_1 \colon start(s). \\ r_2 \colon scc(X) \leftarrow start(X). \quad r_3 \colon scc(Y) \leftarrow scc(X), \&edge[X](Y). \end{array} \right\}$$

where r_1 selects a node s from an externally defined (finite) graph, and r_2 and r_3 recursively compute the strongly connected component of s. To this end, the external atom $\&edge[X](Y)$ is used, which is true if Y is directly reachable from X (and false otherwise).

The implementation of $\&edge[X](Y)$ may look as follows:

```
def edge(x):
  graph=((1,2),(1,3),(2,3))      # simplified example implementation
  for edge in graph:             # search for outgoing edges
    if edge[0]==x.intValue():    # of node x
      dlvhex.output((edge[1],)) # output edge target
```

On the command-line, the call

```
dlvhex2 --python-plugin=plugin.py prog.hex
```

loads the external atoms defined in `plugin.py` and then evaluates HEX program `prog.hex`.

In the system, the Python programming interface is realized as a wrapper of the generic C++ interface as shown in Figure 3, where arcs model both control and data flow. That is, the Python interface uses only the C++ interface but does not communicate with the core reasoning components otherwise. This turns the Python interface in fact into a special C++ plugin. The performance gap between C++ and Python plugins is normally negligible (the update of the Python data structures is in the worst case linear in the number of input atoms), unless the plugin is itself computationally expensive. Wrappers for other languages such as Java or C# can be added similarly and can also be implemented externally, i.e., they do not necessarily need to be part of the DLVHEX solver.

For a complete API description we refer to the system website.

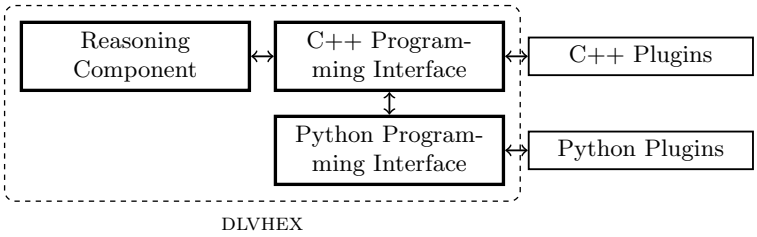

Fig. 3. Architecture of the Python Programming Interface

ASP-Core-2 Standard, Extensions and Interoperability In the course of the organization of the fourth ASP competition, the input language of ASP systems was standardized in the *ASP-Core-2 input language format* [19].[17] The DLVHEX system in its current version supports all features defined in the standard, including function symbols, choice rules, conditional literals, aggregates, and weak constraints. The supported language is therefore a strict superset of the standard.

The system further supports input and output in CSV format to improve interoperability with other systems such as Unix commands or spreadsheet applications. That is, facts may be read from the lines of a CSV file, where the different values are mapped to the arguments of a predicate. After the computation, the extension of a specified predicate may be written in CSV format to allow a seamless further processing by other applications. For instance, consider `salary.csv`:

```
joe,smith,2000
sue,johnson,2200
```

It can be read as facts $emp(1, joe, smith, 2000)$ and $emp(2, sue, johnson, 2200)$ (where the first element is the original line number if relevant) using the DLVHEX command-line option `--csvinput=emp,salary.csv`. Conversely, results can be output in CSV format.

4.4 Exploiting External Source Properties

External sources were seen as black boxes in earlier versions of DLVHEX. It was assumed that the system does not have any information about them, except that there is an oracle function which decides satisfaction of an external atom under a complete assignment. As a consequence, the room for optimizations in the algorithms was limited because the value of an external atom under one assignment did not allow for drawing any conclusions about its behavior under other assignments.

However, in many practical applications the provider of an external source and/or the HEX programmer have additional knowledge about the behavior of the source, for instance, that the source is monotonic, functional, has a limited

[17] `https://www.mat.unical.it/aspcomp2013/files/ASP-CORE-2.01c.pdf`

domain, returns only elements which are smaller than the input (according to some ordering), etc. Knowing such properties allows for implementing more specialized algorithms which are tailored to the particular external sources used in a program. We therefore identified a set of *properties* that external sources might have, and allow the user to specify the ones which are fulfilled by a concrete external source. Note that specifications by the user are assumed to be correct and cannot be further checked by the system, either due to high computational costs or undecidability of some properties.

Example 28. Suppose $\&tail[X](Y)$ is true whenever Y is the string which results from string X if the first character is dropped. Then the output is always smaller than the input wrt. string length.

The system exploits these properties automatically, mainly for two purposes: in the *learning algorithms for scalability enhancements* and in the *grounding component for more flexibility of the language* due to reduced syntactic limitations. In addition, there are several other system components which exploit the properties to further speed up the evaluation, such as skipping various checks if their result is definite due to known behavior of external sources, partitioning a reasoning task into smaller independent tasks, avoiding unnecessary evaluations of external atoms, and drawing deterministic conclusions rather than guessing.

However, as this section presents the system from a user's perspective, we focus on *which* properties can be specified, *how* the user can do that, and give a rough idea of how the system makes use of this information, but we refrain from discussing the involved algorithms in detail. This is in line with the goal of these properties: the user can benefit from the advantages when specifying them, but without the need to care about how the system is going to exploit this information. Instead, the user can generally expect that the more information is available to the system, the more efficient evaluation will be; if the added information does not yield a speedup, it does at least no harm.[18] Some of the properties, such as monotonicity, do even lead to a drop of complexity from Σ_2^P to NP for answer set existence checking over ground disjunction-free programs, provided that external sources are polynomial [51].

Furthermore, properties also serve as *assertions*: if the reasoner observes a behavior of external sources which contradicts the declared properties, appropriate error messages are printed. However, a systematic check of asserted properties is not performed because of high computational costs or even undecidability of some properties.

Specifying Properties The specification of properties is supported in two ways. The first option is to declare them as part of the external source im-

[18] The only property related to potential performance decrease is provision of a *three-valued semantics* as additional calls of the external source are sometimes counter-productive [44]. However, even then the property itself does not harm since it is only exploited by certain (non-default) evaluation heuristics selected via command-line options.

plementation via the *external source interface*. The second option is to specify them as part of the HEX program using so-called *property tags*.

Specification via the External Source Interface. Properties are mostly specified via the (C++ or Python) programming interface for external sources. To this end, the procedural code which implements external atoms calls specific *setter methods* provided by the programming interface to inform the system that the source has certain properties.

Example 29. The implementation of $\&md5[X](Y)$ which computes for a string X its MD5 hash value Y might call `prop.setFunctionality(true)` to let DLVHEX know that for each X there is exactly one Y. This allows the system, for instance, to conclude that $\&md5[x](y_2)$ is false without evaluating the external source, if it has already found a value $y_1 \neq y_2$ such that $\&md5[x](y_1)$ is true.

If a property is declared in this way, the source is meant to *always* provide a certain behavior, independent of its usage in a HEX program, like in case of the computation of a hash value. Another example is $\&diff[p,q](X)$ from Example 26, which computes all values X in the extension of p but not in that of q wrt. assignment $\mathbf{A}$ (formally, these are all values x s.t. $f_{\&diff}(\mathbf{A}, p, q, x) = \mathbf{T}$). This external atom is always monotone/antimonotone in the first/second parameter, which can be specified by calling `prop.addMonotonicInputPredicate(0)` and `prop.addAntimonotonicInputPredicate(1)`.

Example 30. Reconsider the external atom $\&diff[p,q](X)$ from Example 26. It is monotonic in p and antimonotonic in q. We adopt the implementation of the external source as follows in order to inform the reasoner about the properties, which typically leads to efficiency improvements.

```
1  import dlvhex
2
3  def diff(p,q):
4    for x in dlvhex.getTrueInputAtoms():   # for all true input atoms
5      if x.tuple()[0] == p:                # is it of form p(c)?
6        if dlvhex.isFalse(dlvhex.storeAtom(# is q(c) false?
7          (q, x.tuple()[1]))):
8          dlvhex.output((x.tuple()[1], )); # then c is in the output
9
10 def register():
11   prop = dlvhex.ExtSourceProperties()   # inform dlvhex about
12   prop.addMonotonicInputPredicate(0)    # monotonicity/antimon.
13   prop.addAntimonotonicInputPredicate(1)# in first/second parameter
14   dlvhex.addAtom("diff", (dlvhex.PREDICATE, dlvhex.PREDICATE),
15     1, prop)
```

Specification via Property Tags. However, it might also be the case that only a specific usage of an external source in a concrete program has a property. Then the implementer of the external source cannot declare it yet; instead, only the implementer of the HEX program has sufficient knowledge and can declare the property as part of an external atom in the program.

Example 31. Suppose $\&greaterThan[p, 10]()$ checks if the sum of integer values c s.t. $p(c)$ is true is greater than 10. It is not monotone in general if negative

integers are allowed, but it is monotone if a program uses only positive integers. While the provider of the external source cannot assert the property, the user of the external source in a concrete program, who knows the context, can.

To this end, the HEX language and implementation were extended such that external atoms can be followed by *property tags* of form $\langle$*list of properties*$\rangle$, where the list of properties is comma-separated. Each property is a whitespace-separated list of constants, consisting of a *property type* (first element in the list), and a number of *property parameters* (remaining elements in the list), whose number depends on the property type and may also have default values. For example, $\&diff[p,q](X)\langle monotonic\ p, antimonotonic\ q\rangle$ specifies two properties which declare that the external atom is monotonic in p and antimonotonic in q wrt. their extension in the input assignment. Here, the first property *monotonic* p uses the property type *monotonic* and the property parameter p, while the second property *antimonotonic* q uses the property type *antimonotonic* and the property parameter q. Another example is the external atom $\&greaterThan[p,10]()\langle monotonic\rangle$, which declares that the external source is monotonic in all parameters (because it is monotonic in p and it is trivially monotonic in constant input parameters because they are independent of the input assignment); the property type is *monotonic* and no property parameters are explicitly specified, which indicates by default that the source is monotonic in all inputs. Properties declared by tags are understood to hold *in addition* to those declared via the external source interface (stating conflicting properties is not possible with the currently available ones).

Supported properties. The following list gives an overview about the currently available properties and how to specify them if the property tag language is used (but all of them can be specified both via the external source interface or in property tags). Each property is explained with an example in order to show the property type and the expected property parameters.

- **Functionality**: $\&add[X,Y](Z)\langle functional\rangle$
  The external atom adds integers X and Y and is true for their sum Z. The source provides exactly one output value for a given input. There are no property parameters.

- **Monotonicity in a parameter**: $\&diff[p,q](X)\langle monotonic\ p\rangle$
  The external atom computes the difference of the extensions of p and q. The source is monotonic in predicate parameter p (i.e., if the extension of p increases, the output does not shrink), as indicated by the property parameter.

- **Global monotonicity**: $\&union[p,q](X)\langle monotonic\rangle$
  The source computes the set union of the extensions of p and q. It is monotonic in all parameters (indicated by the default value of the missing property parameter). Another example are queries over DL ontologies and RDF queries as mentioned in Section 3.2.

– **Antimonotonicity in a parameter**: $\&diff[p,q](X)\langle antimonotonic\ q\rangle$
The source is antimonotonic in predicate parameter q (i.e., if the extension of q shrinks, the output does not shrink).

– **Global antimonotonicity**: $\&complement[p](X)\langle antimonotonic\rangle$
The source computes the complement of the extension of p wrt. a fixed domain. It is antimonotonic in all parameters.

– **Linearity on atoms**: $\&union[p,q](X)\langle atomlevellinear\rangle$
We have domain independence on the level of atoms, i.e., the source can be separately evaluated for each input atom s.t. the final result is the union of the results of all evaluations. For instance, the evaluation under assignment $\mathbf{A} = \{\mathbf{T}p(a), \mathbf{T}p(b), \mathbf{T}q(c)\}$, which yields $\{a, b, c\}$, can be split up into three evaluations under $\mathbf{A}_1 = \{\mathbf{T}p(a)\}$, $\mathbf{A}_2 = \{\mathbf{T}p(b)\}$ and $\mathbf{A}_3 = \{\mathbf{T}q(c)\}$, which yield $\{a\}$, $\{b\}$ and $\{c\}$, respectively, and their union the result of the evaluation under $\mathbf{A}$. There are no property parameters.

– **Linearity on tuples**: $\&diff[p,q](X)\langle tuplelevellinear\rangle$
We have domain independence on the level of tuples in the extensions of predicate input parameters, i.e., the source can be separately evaluated for each pair of atoms $p(\vec{c})$ and $q(\vec{c})$ for all vectors of terms $\vec{c}$ s.t. the final result is the union of the results of all evaluations. For instance, the evaluation under $\mathbf{A} = \{\mathbf{T}p(a), \mathbf{T}p(b), \mathbf{F}q(a), \mathbf{T}q(b)\}$, which yields $\{a\}$, can be split up into two evaluations under $\mathbf{A}_1 = \{\mathbf{T}p(a), \mathbf{F}q(a)\}$ and $\mathbf{A}_2 = \{\mathbf{T}p(b), \mathbf{T}q(b)\}$, which yield $\{a\}$ and $\emptyset$, respectively, and their union in the result of the evaluation under $\mathbf{A}$. However, it would not be correct to split $\mathbf{A}_2$ further up into $\mathbf{A}_{2.1} = \{\mathbf{T}p(b)\}$ and $\mathbf{A}_{2.2} = \{\mathbf{T}q(b)\}$ as they would yield the results $\{b\}$ and $\emptyset$, which would put b into the final result, which differs from the evaluation under $\mathbf{A}$. There are no property parameters.

– **Finite domain**: $\&edges[graph.dot](X,Y)\langle finitedomain\ 0, finitedomain\ 1\rangle$
Imports the edges of a predefined graph. Both output values can have only finitely many different values. To this end, we specify two properties with type *finitedomain* with property parameters that identify the output terms X and Y by index (0 and 1, respectively).
In the route planning application mentioned in Section 3.2, and shown in more detail below, accesses to external maps fulfill this property since real-world maps are always finite.

– **Finite domain wrt. the input**: $\&diff[p,q](X)\langle relativefinitedomain\ 0\ 0\rangle$
Only constants which already appear in the 0-th input (indicated by the first property parameter 0; points in this case to the predicate p) may occur as first output term (indicated by the second property parameter 0). Informally, the difference between sets represented by predicates p and q can only contain elements which appear in the set represented by p.

– **Finite fiber**: $\&sqrt[X](Z)\langle finitefiber\rangle$
The source computes the square root of X. Each element in the output is only produced by finitely many different inputs (in this case, in fact, only by a single input value). There are no property parameters.

– **Well-ordering wrt. string lengths**: $\&tail[X](Z)\langle wellorderingstrlen\ 0\ 0\rangle$
The source drops the first character of string X and returns the result in Z. The 0-th output (indicated by the second property parameter 0) is no longer than the longest string in the 0-th input (indicated by the first property parameter 0).

– **General well-ordering**: $\&decrement[X](Z)\langle wellordering\ 0\ 0\rangle$
The external atom decrements a given integer. There is an ordering of all constants such that the 0-th output (second parameter) is no greater than the 0-th input (first parameter) wrt. this ordering.

– **Three-valued semantics**: $\&g[\vec{X}](\vec{Y})\langle providespartialanswer\rangle$
The external source can be evaluated under partial assignments, i.e., it can handle assignments which do not define all atoms, but may evaluate to *undefined* (**U**) in this case (can be used with any external source if implemented).

Note that properties are only useful if they are exploited by at least one solving technique or algorithm implemented in the reasoner. It is therefore *not* intended that typical users introduce custom properties, but only tag external atoms with existing ones from the above list. However, for advanced users who contribute to or customize the reasoner itself, the framework supports easy extension of the parser and data structures. Exploiting such a new property in the algorithms might be more sophisticated depending on the particular property and the envisaged goal.

Scalability Boost Recall that the current evaluation algorithm for HEX-programs employs a conflict-driven approach, which learns nogoods from external sources to exclude incorrect guesses. This basic approach can be further improved by keeping the learned nogoods small by exploiting external source properties or external source evaluation under partial assignments.

Exploiting external source properties. In Example 25, atoms $p(a)$ and $q(a)$ in the assignment are in fact irrelevant when deciding whether $\&diff[p,q](b)$ is true because constants a and b are independent (similarly for $p(b)$ and $q(b)$ when deciding $\&diff[p,q](a)$). If this information is available to the system, it can be exploited to shrink nogoods to the relevant part such that the search space is pruned more effectively.

One way to gain the required information is to make use of external source properties. In particular, the independence of a and b in the previous example can be derived from the property 'linearity on tuples'. Then we can reduce the nogood $\{\mathbf{T}p(a), \mathbf{T}p(b), \mathbf{F}q(a), \mathbf{T}q(b), \mathbf{F}e_{\&diff[p,q]}(a)\}$ to $\{\mathbf{T}p(a), \mathbf{F}q(a), \mathbf{F}e_{\&diff[p,q]}(a)\}$ and nogood $\{\mathbf{T}p(a), \mathbf{T}p(b), \mathbf{F}q(a), \mathbf{T}q(b), \mathbf{T}e_{\&diff[p,q]}(b)\}$ to the smaller nogood $\{\mathbf{T}p(b), \mathbf{T}q(b), \mathbf{T}e_{\&diff[p,q]}(b)\}$. If monotonicity in p is known in addition, then nogood $\{\mathbf{T}p(b), \mathbf{T}q(b), \mathbf{T}e_{\&diff[p,q]}(b)\}$ can be further simplified to $\{\mathbf{T}q(b), \mathbf{T}e_{\&diff[p,q]}(b)\}$ by dropping $\mathbf{T}q(a)$ because $\&diff[p,q](b)$ will remain false even if $q(a)$ becomes false.

Exploiting three-valued oracle functions. Alternatively or in addition to external source properties, also three-valued oracle functions can be exploited for shrinking learned nogoods to the essential part [44]. If the truth value is already known and will not change when the assignment becomes more complete, then the set of yet unassigned atoms is irrelevant for the output of the external source. This is exploited for nogood minimization as follows. Whenever a nogood is learned, the system iteratively tries to remove one of the input atoms and evaluate again in order to check if the truth value is still defined. If so, the according atom is not necessary and can be removed from the nogood.

For instance, a proper three-valued oracle function in the previous example allows for reducing the nogood $\{\mathbf{T}p(a), \mathbf{T}p(b), \mathbf{F}q(a), \mathbf{T}q(b), \mathbf{T}e_{\&diff[p,q]}(b)\}$ from above to $\{\mathbf{T}q(b), \mathbf{T}e_{\&diff[p,q]}(b)\}$, because whenever $\mathbf{T}q(b)$ is in the assignment, it is already definite that $\&diff[p,q](b)$ is false.

Discussion and Extensions. Whether to exploit external source properties, three-valued oracle functions, or both, largely depends on the use case. Depending on the type of external source to be realized, the implementation of a three-valued oracle function might be more challenging than that of a Boolean one (implementing an algorithm which decides over partial assignments is in general more difficult than if all information is known). However, it allows for exploiting application-specific knowledge in an optimal way [44]. In contrast, tagging external sources with properties from a list is easy and can still lead to good efficiency.

Language Flexibility based on Liberal Safety As already discussed, external atoms may introduce constants which do not appear in the program (*value invention*). Obviously, this can in general lead to programs Π where no finite subset of the full, possibly infinite grounding $grnd(\Pi)$ of the program has the same answer sets as Π. Since this inhibits grounding in general, it is crucial to identify classes of programs for which the existence of such a finite grounding is guaranteed; we call this property *finite groundability*. Traditionally, *strong safety* was used, which basically forbids value invention by recursive external atoms (i.e., external atoms whose input possibly depends on its own output wrt. the predicate dependency graph, for a formal definition cf. [43]). If only non-recursive external atoms introduce new values, termination is guaranteed. However, it turns out that this is only a sufficient but not a necessary criterion, i.e., strong safety is overly restrictive.

Example 32. The program Π from Example 27 is *not* strongly safe because $\&edge[X](Y)$ is recursive (output Y may be input to the same external atom by another application of r_3) but may introduce values for Y which do not appear in Π. However, if one knows that the graph is finite, one can conclude that the recursive addition of new values will end at some point.

In the example, the criterion may be circumvented by importing the full domain a priori and adding *domain predicates*, i.e., adding $node(Y)$ to the body of r_3 and another rule $node(X) \leftarrow \&node[](X)$ to import all nodes. Then $\&edge[X](Y)$ does no longer invent values because all possible values for Y are

determined in a non-recursive fashion using $\&node[](X)$. However, this comes at the price of importing the whole graph although only a small set of nodes might be in the strongly connected component of s.

Therefore, new safety criteria have been introduced which allow for exploiting both syntactic and semantic conditions to derive finite groundability, where the latter are based on external source properties. For instance, if it is known that the input to an external atom can have only finitely many different values, then (due to the restrictions for oracle functions introduced in Section 2.4) also the set of possible output values will be finite. Furthermore, if an external atom is acyclic, then the set of relevant output values will also be finite. Such considerations have been formalized by the notion of *liberally safe* HEX programs, which are guaranteed to have a finite grounding that can be computed using a novel algorithm [34]. Formally, the notion is based on *attribute positions* of external atoms; however, since the technical details are more elaborated, we give only an intuitive overview here.

Example 33. Let $\&tail[X](Y)$ drop the first character of string X and return it as Y. Then Y is no longer than X and – even if used recursively – it is guaranteed that it can generate only finitely many strings because there are only finitely many strings with a length up to the one of X.

In addition to the declaration of predefined properties, the generic framework is also extensible in such a way that custom knowledge about external sources can be exploited. To this end, providers may implement additional safety criteria, which are integrated into the safety check. The safety check itself is fast (at most quadratic in the size of the non-ground program).

The system combines the available information, given by syntactic conditions, specified semantic properties, and safety plugins in order to check safety of the program. This does not only allow for writing programs with fewer syntactic restrictions, but the implementation of some applications may become possible in the first place. For instance, in *route planning applications*, importing the whole map material a priori is practically impossible due to the large amount of data, while a selective import using liberal safety makes the application possible [34], as is the case for the route planning application described in the next section.

In case a program is not safe, the system prints hints such as the rule and the variable for which finiteness during instantiation could not be proven. This information is intended to guide the user when providing more information in order to make the program safe, e.g., by adding properties which constrain the values of this variable further. Alternatively, a command-line option allows to disable the safety check altogether, in which case there is no guarantee that the reasoner terminates.

5 Case Study

In this section, a case study is presented, where we describe the practical treatment of a realistic problem by employing DLVHEX, following the methodology

introduced in Section 3. This section also serves as a tutorial covering the basic usage of DLVHEX, as well as some advanced features. For this purpose, we develop a HEX-encoding step by step, which is more elaborated than the example programs considered in the previous section, and discuss possible efficiency improvements that can be achieved by exploiting facilities provided by the DLVHEX-system. More details on implementing HEX applications can be found in the DLVHEX user guide [46]. The code snippets presented in this section are fragments of the complete implementation for this case study available at `https://github.com/hexhex/manual/tree/master/RW2017/`.

The problem of our case study is from the *route planning* domain, which has already been considered for HEX before, e.g. in [34,46,47,90]. Suppose one wants to plan a trip through the city of Vienna, where a number of places should be visited on the way. For planning the trip, we rely on data about metro, tram and bus stations, which can be obtained from `data.wien.gv.at`. It contains tuples of the form (l, l', c, t), where l and l' are locations in Vienna, for example 'Karlsplatz' or 'Wien Mitte', t is a means of transport that connects both locations, e.g. 'Bus 65A' or 'Metro U4', and c is an integer representing the associated costs, e.g. the amount of time required to travel from l to l' by using t. For our implementation, the data is contained in a file which is structured as shown in Figure 4.

```
1 Museumsquartier              Karlsplatz       1 U2
2 Floetzersteigbruecke         Matschgasse      3 51A
3 WaehringerStrasseVolksoper   Kutschkergasse   2 40
4 ...
```

Fig. 4. vienna_transport.graph

For instance, line 1 states that 'Karlsplatz' can be reached from 'Museumsquartier' taking 'U2' at cost 1.

5.1 Sequence Generation

Given a subset of all possible locations contained in the data set, the general goal is to compute a sequence of locations that satisfies a number of criteria, which we are going to introduce in several stages. We start developing our encoding by generating all possible sequences in which a number of destinations could be visited, by means of the HEX program shown in Figure 5 (which will be extended in the sequel of this section). This corresponds to the *guessing part* of the encoding as described in the basic methodology in Section 3.

The first rule generates a guess for each combination of locations that should be visited and possible position in the resulting sequence. Here, the locations that should be visited are assumed to be all locations contained in the extension of the predicate `destination`, and `C` is the number of such locations obtained by using the `#count`-aggregate. The constraints in lines 4 and 5 state that each location should only appear once in the sequence, and that two locations cannot

```
1  sequence(I,L) v nsequence(I,L) :- destination(L),
2          #int(I), #int(C), C = #count{N : destination(N)}, I < C.
3
4  :- sequence(I1,L), sequence(I2,L), I1 != I2.
5  :- sequence(I,L1), sequence(I,L2), L1 != L2.
6
7  inSequence(L) :- sequence(I,L)}.
8  :- destination(L), not inSequence(L).
9
10 haveLocation(I) :- sequence(I,L).
11 :- sequence(I,L), I1 < I, #int(I1), not haveLocation(I1).
```

Fig. 5. route_planning.hex

be visited at the same time, respectively. The rules in lines 7 to 11 ensure that every destination appears in the sequence and that there are no gaps in the sequence.

If we add two destinations via the facts `destination("Stephansplatz")` and `destination("Karlsplatz")`, and execute the program by making the command-line call:

```
$ dlvhex2 route_planning.hex --filter=sequence --maxint=10
```

the DLVHEX-system returns the following two answer sets:

```
{sequence(1,"Stephansplatz"),sequence(0,"Karlsplatz")}
{sequence(0,"Stephansplatz"),sequence(1,"Karlsplatz")}
```

By using the command line option `--filter`, we can limit the output to a specific predicate, and `--maxint` sets an upper limit for the integers that occur in the grounding of the program. The latter value needs to be chosen large enough depending on the program. The returned answer sets correspond to all possible sequences in which the given destinations can be visited.

5.2 Trip Planning

Next, we want to exploit the information we have in our data set about connections between locations (via metro, tram or bus) in order to retrieve which means of transport we can use regarding a particular visit sequence. In addition, based on the associated costs for each trip via a certain means of transport, we are interested in the fastest connections between destinations in the sequence. For this reason, the next step in the development of our encoding consists in creating an external source that computes the shortest path between two locations in Vienna, by employing a dedicated algorithm. The plugin should retrieve the fastest connection together with the required costs and the traffic lines that need to be taken. At this, the corresponding plugin method needs access to our data set file.

Our goal is to implement an external source that can be interfaced by means of an external atom of the following form:

```
&route[File,Location1,Location2](Station1,Station2,Costs,Line)
```

Given the name of a file containing the transport data ('vienna_transport.graph' in our case) and two location names, the external source should yield all tuples representing direct connections between stations that need to be visited in order to travel from `Location1` to `Location2` with minimal costs, together with the costs for each connection and the transport line used. Thus, the shortest path from one location to another needs to be computed externally by the corresponding plugin. This corresponds to *computation outsourcing* as described in Section 3.

Here, we utilize the *Python* interface of DLVHEX to implement the plugin, which allows faster prototyping than the alternative *C++* interface, on which the *Python* interface is based. By outsourcing the computation of the optimal connection between two locations, we can access an off-the-shelf implementation contained in a *Python* library for this task. The plugin implementation is realized as shown in Figure 6.

```
1  import dlvhex
2  import networkx as nx

   ...

6  def route(graph, start, end):
7    G = nx.read_edgelist(graph.value()[1:-1], nodetype=str,
8      data=[('weight', float),('label', str)],
9      create_using=nx.MultiDiGraph())
10   shortestPath = nx.shortest_path(G, source=start.value()[1:-1],
11     target=end.value()[1:-1], weight='weight')
12
13   for i in range(0, len(shortestPath)-1):
14     costs = 10
15
16     for edge in G.edges(data=True):
17       if edge[0] == shortestPath[i] and
18         edge[1] == shortestPath[i+1] and
19         edge[2]['weight'] < costs:
20         costs = edge[2]['weight']
21         transport = edge[2]['label']
22
23     dlvhex.output(('"' + shortestPath[i] + '"',
24       '"' + shortestPath[i+1] + '"',
25       int(costs), '"' + transport + '"'))

   ...

56 def register():
57   prop = dlvhex.ExtSourceProperties()
58   prop.addFiniteOutputDomain(0)
59   prop.addFiniteOutputDomain(1)
60   prop.addFiniteOutputDomain(2)
61   prop.addFiniteOutputDomain(3)
62   dlvhex.addAtom("route", (dlvhex.CONSTANT, dlvhex.CONSTANT,
63     dlvhex.CONSTANT), 4, prop)
```

Fig. 6. route_plugin.py

First, we need to import the *Python* library `dlvhex`, and the `networkx` package for performing graph computations. The implementation of the external source mirrors the input-output structure of external atoms, in that a

plugin is constituted by a *Python* method with arguments corresponding to the input parameters of the external atom; and output tuples of an external atom are added via the interface method `dlvhex.output()`, representing the results of the plugin method. In this respect, it is essential that the plugin method implements a stateless behavior where the same set of output tuples is returned for a specific input each time the method is called, as the semantics of HEX and the DLVHEX-algorithm rely on this property.

Inside the plugin method starting at line 6, we first import the transport network using the file name provided in the call of the external source. The respective input constant can be retrieved by calling the method `value()` on the first argument of the plugin method. As the transport network does not change between calls of the external source, the graph could additionally be cached in the implementation, so that it would not need to be reloaded each time the source is evaluated. However, we omit the caching here to keep the code listing succinct. Afterwards, we compute and store the shortest path between the locations provided as input constants by means of the library function `nx.shortest_path`, in line 10. Finally, in lines 13 to 25, we build the output tuples representing separate connections on the way from the start to the end location, together with the traffic line taken in each step and the associated costs; and we return them via the method `dlvhex.output()`.

For DLVHEX to be able to call the plugin method for evaluating the truth value of an external atom, we need to register all plugins in a designated method called `register` (line 56). This is done via the method `dlvhex.addAtom` in line 62, which takes the method name corresponding to the external atom name, a tuple defining the input parameter types, the output arity, and a properties-object as arguments. For the external atom at hand, all input parameters are declared to be constants. If the evaluation of an external atom depends on the extension of some input predicate in the given interpretation, the type `dlvhex.PREDICATE` is used instead (as will be demonstrated below). A properties-object is obtained via the method `dlvhex.ExtSourceProperties()` and stored in the variable `prop` in line 57. It can be used to declare the external source properties described in Section 4. Here, we just define that each element of the output tuple can take only finitely many different values since our transport network is finite.

Now, we can extend our encoding by further rules that utilize the external atom named `&route` as shown in Figure 7.

We extend the file *route_planning.hex* by a rule that retrieves all connections we have to take regarding a given visit sequence from the external source, in line 13. Note that, besides the locations in the extension of the predicate `destination` which we add to the program, the encoding does not contain any other locations. Thus, these need to be introduced by *value invention* by the external atom, restricted to the relevant stations. Moreover, we want to obtain the connections we are taking during the trip in sequential order. For this, we reference the overall length of the computed trip via the predicate `pathLength`, in line 17. In lines 19 to 26, we aggregate the connections between stations in form of a new sequence containing the whole trip in `tripTmp`, where we take connections between two destinations in line 21, and transitions between

```
11 ...
12
13 connection(L1,L2,X,Y,C,T) :- sequence(N,L1), sequence(Next,L2),
14         Next = N + 1,
15         &route["vienna_transport.graph",L1,L2](X,Y,C,T).
16
17 pathLength(L) :- L = #count{L1,L2,X,Y : connection(L1,L2,X,Y,C,T)}.
18
19 tripTmp(0, X, L2, X, Y, C, T) :- sequence(0, X),
20         connection(X, L2, X, Y, C, T).
21 tripTmp(S, L1, L2, Y, Z, C2, T2) :- tripTmp(P, L1, L2, X, Y, C, T),
22         connection(L1, L2, Y, Z, C2, T2), S = P + 1, #int(S),
23         pathLength(L), S <= L.
24 tripTmp(S, Y, L3, Y, Z, C2, T2) :- tripTmp(P, L1, Y, X, Y, C, T),
25         connection(Y, L3, Y, Z, C2, T2), S = P + 1, #int(S),
26         pathLength(L), S <= L.
27
28 trip(S, X, Y, C, T) :- tripTmp(S, L1, L2, X, Y, C, T).
```

Fig. 7. route_planning.hex - second part

destinations from the initial sequence in line 24. For this, we use the variables `L1` and `L2` to associate sub-paths with trips between destinations, which we project away in line 28 to obtain the relevant trip information.

Now, assume we indicate a starting position by means of the fact `sequence(0,"Volkstheater")`, and add the facts `destination("Taubstummengasse")`, `destination("Stephansplatz")` and `destination("Volkstheater")` to the encoding. If DLVHEX is called by

```
$ dlvhex2 route_planning.hex --python-plugin=route_plugin.py
  --maxint=10 --filter=trip
```

the following possible trips are returned:

```
{trip(0,"Volkstheater","Museumsquartier",1,"U2"),
 trip(1,"Museumsquartier","Karlsplatz",1,"U2"),
 trip(2,"Karlsplatz","Taubstummengasse",1,"U1"),
 trip(3,"Taubstummengasse","Karlsplatz",1,"U1"),
 trip(4,"Karlsplatz","Stephansplatz",1,"U1")}

{trip(0,"Volkstheater","Herrengasse",1,"U3"),
 trip(1,"Herrengasse","Stephansplatz",1,"U3"),
 trip(2,"Stephansplatz","Karlsplatz",1,"U1"),
 trip(3,"Karlsplatz","Taubstummengasse",1,"U1")}
```

When calling DLVHEX with a program containing an external atom for which the corresponding plugin is implemented in a *Python* file, the path to the file needs to be provided via the option `--python-plugin`.

Consequently, there are two different options we can choose from, where the second trip is shorter. In order to only obtain the shortest trip, we could make use of weak constraints as introduced in Section 2, by adding the following to our encoding: `:∼ trip(S, X, Y, C, T). [C@1,S,X,Y,C,T]`. As a consequence, by minimizing overall costs, DLVHEX only returns the answer set where the sum of costs of `trip`-atoms in the answer set is minimal. Computing shortest trips is related to the traveling salesperson problem.

5.3 Cyclic Dependencies

Next, suppose we want to refine our encoding further by also taking the requirement into account that if the whole trip is longer than a certain value, we want to include a destination that has a restaurant for having lunch in our trip. For this, we can introduce another external atom that checks whether we need a restaurant, by making use of the fact that it is easy to combine several external sources in a program with DLVHEX. We create the *Python* implementation for an external atom of the form `&needRestaurant[trip,Limit]()`. It should simply evaluate to true if the sum of the costs of the connections contained in the true extension of `trip` (relative to the current solver assignment) exceeds the constant value `Limit`. In contrast to `&route`, this external atom does not provide any output values, which is often the case when external atoms are used for checks or constraints in a HEX program. We extend our plugin file as shown in Figure 8.

```
25  ...
26
27  def needRestaurant(trip,limit):
28    tripLength = 0
29
30    for x in dlvhex.getInputAtoms():
31      if x.tuple()[0] == trip and x.isTrue():
32        tripLength += int(x.tuple()[4].value())
33
34    if tripLength > int(limit.value()):
35      dlvhex.output(())

    ...

56  def register():

    ...

65    prop = dlvhex.ExtSourceProperties()
66    prop.addMonotonicInputPredicate(0)
67    dlvhex.addAtom("needRestaurant",
68                   (dlvhex.PREDICATE, dlvhex.CONSTANT), 0, prop)
```

Fig. 8. route_plugin.py - second part

In the plugin method, we iterate over all input atoms and filter those that have the predicate name which has been passed to the external source, and which are true in the current solver assignment, in lines 30 and 31. Then, we add up all according costs by accessing the fourth argument of the respective atom. Finally, we define that the external atom evaluates to true if the provided limit is exceeded, by returning an empty output tuple, in line 35. If the output method of a Boolean external atom is not called at all, DLVHEX interprets this as an evaluation to false.

Here, we can declare that the external atom behaves monotonically on the first input parameter because once the costs associated to some trip exceed the limit, the external atom cannot be false when further connections are added to the trip. Defining such properties often has a large impact on the efficiency

of the solving process. For instance, if we call the plugin method for a trip containing only one connection, the costs of which already exceed the limit, the truth value of the external atom is implied for any other trip as soon as it contains that connection, due to monotonicity of the external source. However, this cannot be detected by DLVHEX if the corresponding property declaration is missing. Also, in contrast to the first plugin, we now declare the external atom to have a predicate input parameter, which causes DLVHEX to pass the complete extension of the predicate occurring in the ground HEX program (and, in the standard configuration, to postpone the external evaluation until the truth values of all its instances are decided during solving).

The additional external atom is used in the extension of our HEX-encoding shown in Figure 9 in order to decide whether a location having a restaurant needs to be included in the trip.

```
27 ...
28
29 needRestaurant v notNeedRestaurant.
30 needRestaurant :- &needRestaurant[trip, 3]().
31 notNeedRestaurant :- not &needRestaurant[trip, 3]().
32
33 chooseRestaurant(R,L) v nchooseRestaurant(R,L) :- needRestaurant,
34         restaurant(R,L).
35 :- needRestaurant, chooseRestaurant(R1,L1),
36         chooseRestaurant(R2,L2), R1 != R2.
37 chosen :- needRestaurant, chooseRestaurant(R,L).
38 :- needRestaurant, not chosen.
39
40 destination(L) :- needRestaurant, chooseRestaurant(R,L).
```

Fig. 9. route_planning.hex - third part

Here, we apply the external atom `&needRestaurant` to decide if we need a restaurant or not, in lines 29 to 31. Then, we choose exactly one location that has a restaurant from all locations that are declared to have a restaurant by the predicate `restaurant`, in lines 33 to 38. Finally, we add the chosen restaurant location to our destinations.

We test the extended HEX program with the same destinations and starting location as before, and state that the location 'Museumsquartier' has a restaurant by adding the fact `restaurant("Museumsquartier")`. We use the following command:

```
$ dlvhex2 route_planning.hex --python-plugin=route_plugin.py
  --maxint=10 --filter=trip --aggregate-mode=ext
```

where the additional option `--aggregate-mode=ext` activates the internal aggregates implementation of DLVHEX. This is necessary whenever there is a cycle in a HEX program that contains aggregates as well as external atoms. Overall, the call yields six answer sets as both of the two trips from before have costs greater than 3 and an additional restaurant location needs to be added to the respective sequences. The shortest trip from before is not viable anymore since

it does not include a restaurant location, but a detour to 'Museumsquartier' can be inserted, so that the following answer set is a solution regarding our new encoding:

```
{trip(0,"Volkstheater","Herrengasse",1,"U3"),
 trip(1,"Herrengasse","Stephansplatz",1,"U3"),
 trip(2,"Stephansplatz","Karlsplatz",1,"U1"),
 trip(3,"Karlsplatz","Museumsquartier",1,"U2"),
 trip(4,"Museumsquartier","Karlsplatz",1,"U2"),
 trip(5,"Karlsplatz","Taubstummengasse",1,"U1")}
```

Note that now the information retrieved from the external atom named `&route` influences the input of the same atom since the extension of the predicate `destination`, and in turn of the predicate `sequence`, depends on the costs for the retrieved connections. Such loops over external atoms potentially make the grounding of a HEX program infinite, even when only finitely many values are introduced by each separate call of an external source. In general, this can be avoided by imposing the *strong safety* condition [34], which, informally speaking, forbids cyclic dependencies over external atoms that introduce new values.

However, the strong safety condition is overly restrictive, and we observe that our encoding can be handled without problems by DLVHEX, even though value invention is employed. This is because the *liberal safety* condition is used by DLVHEX by default, which has been introduced in Section 4 and ensures that programs that are not strongly safe still have a finite grounding. This is the case for our program as the traffic network is finite. Using the command line option `--strongsafety` would yield a warning. To make our program strongly safe, we could add a domain predicate to the rule in line 12 of the encoding containing all possible connections in the transport network. However, this is infeasible in our case due to the large size of the network, of which only a small fraction is relevant for planning the trip.

5.4 Partial Evaluation

During solving, DLVHEX incrementally extends the set of truth assignments for ground atoms such as `trip(0,"Volkstheater","Herrengasse",1,"U3")` and `destination("Museumsquartier")`. The external source for the atom `&needRestaurant[trip,3]()` is invoked as soon as all ground atoms with predicate name `trip` have a truth value because the truth value of the external atom could still change before the complete true/false extension of `trip` is known.

When calling the plugin method `needRestaurant(trip,limit)`, DLVHEX provides information concerning all ground instances of atoms with predicate name `trip` via the interface method `getInputAtoms()`, as demonstrated in Figure 8. Their respective truth values can be queried in *Python* by means of the methods `isTrue()` and `isFalse()`. Based on the truth values of atoms in its input extension, the plugin declares the corresponding output tuples. Accordingly, all ground external atoms that instantiate one of these output

tuples need to evaluate to true under the given assignment, and the remaining ground instances are assumed to be false. The according input-output relations obtained from an external call are then added as nogoods to the solver, so that the correct truth values for the respective ground external atoms are implied.

For instance, when the method `needRestaurant(trip,limit)` is called under an assignment that assigns true to six ground atoms with predicate `trip`, each associated with a cost of 1, and false to all other `trip`-atoms, a nogood is generated that implies that `&needRestaurant[trip,3]()` is true whenever the respective six atoms are true in an assignment. However, note that the truth value of `&needRestaurant[trip,3]()` is already fixed as soon as just four `trip`-atoms have been assigned the value true (assuming costs of 1 for each connection), even though all other atoms might still be unassigned. On the other side, if the plugin method in Figure 8 would be called by DLVHEX under an assignment only containing three true `trip`-atoms (while all others are not assigned), `&needRestaurant[trip,3]()` would be inferred to be false whenever the three `trip`-atoms are true, which does not hold in general. Hence, external sources cannot directly be called under partial assignments without taking care of the latter issue.

For this reason, DLVHEX enables partial evaluation of external atoms by providing the additional output method `outputUnknown()` for declaring that the correctness of some output tuple cannot be determined without information about further truth assignments. This corresponds to the three-valued oracle functions from Section 4. We exploit this feature in a variant of the method from Figure 8, as shown in Figure 10.

An external source is only called under partial input assignments by DLVHEX if the property `setProvidesPartialAnswer(True)` is set. In this case, it is the responsibility of the source developer to make sure that all outputs that may potentially be derived when the assignment is extended are declared via the method `outputUnknown()`. The additional implementation effort for allowing partial evaluations often pays off, since partial external sources allow DLVHEX to evaluate external atoms earlier during search. This leads to an earlier detection of wrong guesses and smaller (thus, more general) input-output nogoods, resulting in efficiency improvements. Moreover, external sources allowing partial evaluations can also be used by DLVHEX for minimizing nogoods to find the "essential" part of an input assignment on which a given output depends, as described in Section 4.

Regarding our `needRestaurant`-plugin, in addition to counting the costs of `trip`-atoms that are true in the given assignment, we now also need to keep track of the maximal costs that may result from extending the assignment. For this purpose, in lines 35 to 37, we count the costs for all atoms that are true or not assigned, i.e. those which are not known to be false. Furthermore we state that the truth value of the external atom is not known if the limit has not been exceeded, but may be exceeded during future evaluation steps, in line 42.

To enable partial evaluation when starting DLVHEX, the command line option `--eaevalheuristics=always` needs to be set, so that all external sources allowing partial evaluations are queried whenever the solver assignment is extended. If the option `--eaevalheuristics=periodic` is used, DLVHEX waits

```
25  ...
26
27  def needRestaurant(trip,limit):
28    tripLength = 0
29    maxTripLength = 0
30
31    for x in dlvhex.getInputAtoms():
32      if x.tuple()[0] == trip and x.isTrue():
33        tripLength += int(x.tuple()[4].value())
34
35    for x in dlvhex.getInputAtoms():
36      if x.tuple()[0] == trip and not x.isFalse():
37        maxTripLength += int(x.tuple()[4].value())
38
39    if tripLength > int(limit.value()):
40      dlvhex.output(())
41    elif maxTripLength > int(limit.value()):
42      dlvhex.outputUnknown(())

    ...

56  def register():

    ...

65    prop = dlvhex.ExtSourceProperties()
66    prop.addMonotonicInputPredicate(0)
67    prop.setProvidesPartialAnswer(True)
68    dlvhex.addAtom("needRestaurant", (dlvhex.PREDICATE,
69      dlvhex.CONSTANT), 0, prop)
```

Fig. 10. route_plugin.py - partial evaluation

ten iterations before a source is evaluated again, mitigating potential runtime overheads when the computation inside the external source requires more runtime. Nogood minimization is activated with the option `--ngminimization=always`.

5.5 Interfacing a Description Logic Reasoner

While the two external sources discussed so far in this section are tailored to our specific problem, existing plugins often accomplish more generic tasks. For instance, *Semantic Web* technologies can be leveraged in HEX programs by interfacing a *Description Logic* (*DL*) reasoner for taxonomical reasoning, as introduced in Section 3. Moreover, it is often a useful strategy to implement new plugins that are created for a specific purpose in a generic manner, so that they can be easily reused in other HEX programs. In the last part of our case study, we provide examples for both of these use cases.

Before, we simply declared one location as a restaurant location by adding the corresponding fact to our encoding. Now suppose we have a DL ontology available, containing information about restaurants in Vienna and their corresponding locations, which we want to use for inferring suitable lunch locations. To illustrate this, we use a small sample ontology formalized in *RDF* syntax in the file 'lunch.owl'. The definitions in our ontology file correspond to the axioms and assertions shown in Figure 11.

$$
\begin{aligned}
BeerGarden &\sqsubseteq Restaurant \\
BeerGarden &\sqsubseteq \neg IndoorRestaurant \\
IndoorRestaurant &\sqsubseteq Restaurant \\
IndoorRestaurant &\sqsubseteq \neg BeerGarden \\
IndoorRestaurant &\sqsubseteq \neg WurstStand \\
Restaurant &\sqsubseteq \exists closeTo.Location \\
WurstStand &\sqsubseteq Restaurant \\
WurstStand &\sqsubseteq \neg IndoorRestaurant
\end{aligned}
\qquad
\begin{aligned}
&Location(Karlsplatz) \\
&Location(Museumsquartier) \\
&Location(Praterstern) \\
&BeerGarden(bg1) \\
&closeTo(bg1, Praterstern) \\
&IndoorRestaurant(ir1) \\
&closeTo(ir1, Museumsquartier) \\
&WurstStand(ws1) \\
&closeTo(ws1, Karlsplatz)
\end{aligned}
$$

Fig. 11. DL-Lite axioms and assertions defined in lunch.owl

In the ontology, for instance the concepts `BeerGarden` and `IndoorRestaurant` are disjoint, every `Restaurant` is close to some `Location`, and `bg1` is a `BeerGarden` close to the `Location Praterstern`. For reasoning with ontologies expressed in lightweight DLs [22], the *DL-Lite* plugin for DLVHEX has been developed which is publicly available at `https://github.com/hexhex/dlliteplugin`. The plugin can be installed and used off the shelf by an end-user of DLVHEX, without the need for understanding the details of its implementation or for additional configurations. The plugin provides external sources for several external atoms that can be used for role and concept queries, as well as consistency checking, and uses a dedicated DL reasoner in the back-end.

For example, by adding the rule

```
restaurant(R,L) :- &rDL["lunch.owl",cp,cm,rp,rm,"closeTo"](R,L).
```

we can retrieve all restaurants with their close-by locations. The external atom for retrieving the extensions of roles has the name `&rDL`, while `&cDL` is used for concept queries. At this, the name of a file containing the ontology encoding and a role name need to be provided as arguments to the external atom. The input predicates `cp`, `cm`, `rp`, `rm` can be used to declare additions to the extensions of concepts and roles as well as to their complements, which are performed before the according query is executed. Consequently, a bidirectional information exchange between the HEX program and the DL reasoner is possible, but we are not exploiting this feature here.[19]

5.6 Accessing Remote Data

As a final refinement of our HEX-encoding, we consider a situation where we need to dynamically integrate some remote data into the evaluation of a HEX program. This represents another common use case for DLVHEX, since remote data that is subject to changes cannot be incorporated into an encoding during construction time, even though no external reasoning is required in this case.

[19] For more details, refer to the documentation of the *DL-Lite* plugin at `http://www.kr.tuwien.ac.at/research/systems/dlvhex/dlliteplugin.html`.

Hence, this usage of external sources constitutes a typical case of information outsourcing.

Nowadays, many web services provide access to their data resources via APIs. These often return data in the now ubiquitous *JSON* format, which expresses data objects by means of (nested) key-value pairs. For example, weather data can be retrieved from `http://openweathermap.org/`, and the following represents part of the data which is returned as JSON string when a request for the current weather in a given location is sent:

```
{"weather":[{"id":803,"main":"Clouds","description":"clouds",
            "icon":"04d"}], ...}.
```

Our aim is to exploit this data in our program and to decide based on the current weather if we should have lunch outside or inside. At the same time, the external source we create for querying the JSON data should be generic, so that it can be used for querying arbitrary JSON providers that are reachable via an URL. Here, the external atom should enable accessing a specified data field of a JSON object. For instance, in order to retrieve the string representing the current weather inside the JSON object from above, we need to access the array stored under the key 'weather', and retrieve the value for the key 'main' of the object contained in this array. A corresponding plugin method can be realized as shown in Figure 12.

```
2   ...
3   import urllib as ul
4   import json

    ...

44  def getJSON(url,fields):
45    jsonurl = ul.urlopen(url.value()[1:-1])
46    data = json.loads(jsonurl.read())
47
48    for field in fields:
49      if field.value()[1:-1].isdigit():
50        data = data[int(field.value()[1:-1])]
51      else:
52        data = data[field.value()[1:-1]]
53
54    dlvhex.output(('"' + str(data) + '"', ))
55
56  def register():

    ...

71    prop = dlvhex.ExtSourceProperties()
72    prop.setFunctional(True)
73    dlvhex.addAtom("getJSON", (dlvhex.CONSTANT, dlvhex.TUPLE ), 1,
74      prop)
```

Fig. 12. route_plugin.py - third part

For retrieving data from a URL and parsing JSON strings, we import the libraries `urllib` and `json`, respectively. The plugin method is provided with

a URL and information about the keys that need to be used to obtain the data chunk that should be returned. At this, the input parameter `fields` is declared to be of the type `dlvhex.TUPLE`, allowing an arbitrary number of input constants to be provided. Here, these constants represent the sequence of keys that need to be used to access the data field containing the respective value of interest. After loading the JSON data from the provided URL in lines 45 and 46, we iterate through the keys of the input tuple provided as second argument, following the path to the target value, in lines 48 to 52. If an entry constitutes an array, an integer needs to be used as key to access the respective entry, in line 50. Otherwise, we can simply use the respective string as key. Once the complete sequence has been processed, the target value is declared as output value of the external source. Note that here we can declare functionality of the external atom as only one value is returned for a given input.

Now, we can utilize the DL-Lite plugin and our new JSON plugin in combination to choose a lunch location depending on the current weather. For this purpose, we extend our HEX-encoding by the rules shown in Figure 13.

```
39 ...
40
41 weather(X) :- &getJSON["http://api.openweathermap.org/data/2.5/
42          weather?q=Vienna&apikey=APIKEY","weather","0","main"](X).
43
44 restaurant(R,L) :- &rDL["lunch.owl",cp,cm,rp,rm,"closeTo"](R,L),
45          &cDL["lunch.owl",cp,cm,rp,rm,"IndoorRestaurant"](R),
46          weather("Rain").
47
48 restaurant(R,L) :- &rDL["lunch.owl",cp,cm,rp,rm,"closeTo"](R,L),
49          &cDL["lunch.owl",cp,cm,rp,rm,"-IndoorRestaurant"](R),
50          not weather("Rain").
```

Fig. 13. route_planning.hex - fourth part

In the first rule in line 41, we retrieve the current weather from the *Open Weather Map* service. For this to work, the string `APIKEY` needs to be replaced by a valid API key, which can be obtained from `http://openweathermap.org/`. All of the constants we provide after the URL are contained in the tuple `fields` when the external source is called. They denote that we first want to lookup the array that is mapped to the key `"weather"`. Then, we select the element with index 0 in the retrieved array and obtain the value associated with the key `"main"`, which is a string describing the current weather. Finally, we retrieve all indoor restaurants from the ontology if it is raining. Otherwise, we pose a query for all restaurants that can be derived to be not an indoor restaurant, in line 48. In our case, these are all restaurants that can be derived to be a beer garden or a wurst stand, due to the disjointness axioms in Figure 11.

5.7 Summary of the Case Study

We started by generating all permutations of a set of locations in Vienna, representing different visit sequences, and ended up with an elaborate encoding

for planning trips through the city of Vienna, satisfying a number of heterogeneous constraints. The final encoding makes use of four different types of external atoms, which are used for computation outsourcing (computing shortest paths by means of a dedicated algorithm), information outsourcing (retrieving remote data from an URL), combined information and computation outsourcing (concept and role queries to an external ontology), and external checks. We discussed the implementation of three plugin methods in detail, and we demonstrated how the corresponding external atoms can be used in combination with already available plugin implementations for DLVHEX, such as the DL-Lite plugin. For this usage, it is important to adhere to the conditions imposed by the formal semantics of oracle functions, by ensuring a stateless behavior of external sources, and by declaring all potential outputs under partial evaluations. The result is a working HEX implementation, which is available at `https://github.com/hexhex/manual/tree/master/RW2017/` and can be executed by using the DLVHEX-system.

6 Further HEX Usage and Related Work

In Sections 3 and 5, we have already considered concrete application scenarios where external atoms are used in a problem encoding. The specific external atoms considered were mostly tailored to the given problem and similar plugins can be developed by a user on demand. However, there are also other types of usage scenarios for HEX, where either new language features are implemented based on the HEX formalism, or other formalisms are translated into HEX programs. In this section, we give an overview over further applications falling into these classes, and consider related work.

6.1 Further HEX Use Scenarios and HEX Extensions

Some advanced HEX applications call for additional language features, which cannot be realized easily in pure HEX programs. However, often such extensions can be realized by compiling them to pure HEX programs. HEX programs can also be used as a backend for realizing formalisms that do not resemble HEX, by using an appropriate translation.

HEX$^{\exists}$ programs. As already mentioned earlier, an important feature of HEX programs is that they are capable of value invention, i.e., that new constants are introduced into a program. This can be used to realize existential quantification in the head of rules in a formalism called HEX$^{\exists}$ [33]. The approach is related to Datalog$^{\pm}$ [17], which also allows existential quantification in heads, but HEX$^{\exists}$ offers *domain-specific existential quantification* such that the structure of introduced values can be controlled via external atoms.

For example the following rule, intuitively 'every employee has an office',

$$r_1\colon \qquad \exists X\colon \mathit{office}(Y,X) \leftarrow \mathit{employee}(Y).$$

is not interpretable in standard ASP due to the existential quantification in the head. The rule, which is also called an *existential rule*, can be rewritten into the following HEX$^{\exists}$ rule:

$$r_1': \qquad \mathit{office}(Y, X) \leftarrow \mathit{employee}(Y), \&\mathit{exists}^{1,1}[r_1, Y](X).$$

The external atom introduces novel constant terms into the program, based on the rule identifier (here r_1) and universally quantified variables in the rule body (here Y). Superscripts '$1, 1$' on the external atom indicate that we need to invent one value from values of one universally quantified variable.

HEX programs with function symbols. Uninterpreted function symbols, for example $do(a, s)$ to represent the follow up of a situation s after executing an action a, can be realized in HEX using external atoms. To this end, composition and decomposition of function terms with external atoms are simulated as in [18].

As a simple example, the program

$$\begin{aligned} q(f(X)) &\leftarrow p(X). \\ r(Y) &\leftarrow q(f(Y)). \end{aligned}$$

would be rewritten into the HEX program

$$\begin{aligned} q(A) &\leftarrow p(X), \&\mathit{comp}_1[f, X](A). \\ r(Y) &\leftarrow q(B), \&\mathit{decomp}_1[B](f, Y). \end{aligned}$$

where the external atom $\&\mathit{comp}_1[f, x](a)$ is true for a constant a that represents the function term $f(x)$. Moreover, $\&\mathit{decomp}_1[b](f, y)$ analyzes and decomposes the term b: if b contains a representation of a function term of form $f(y)$, then the external atom is true for output term y. While being formally defined on ground terms x, y, a, and b, the program contains variables X, Y, A, and B, respectively.

This way function terms can be emulated via HEX programs, moreover we obtain an increased control over issues like maximum nesting depth of terms in the external atoms.

ACTHEX: HEX programs with action atoms. ACTHEX [6,52] is an extension of HEX: an ACTHEX-program is repeatedly executed in an *environment*, can obtain (sense) information from the environment using external atoms, and can declaratively schedule actions to be executed in the environment using *action atoms* in the head of rules. The environment is an abstraction of the world outside the logic program. External atoms are generalized such that the environment may influence their truth values.

Example 34 (simplified from [52]). The following ACTHEX-program controls a robot capable of executing a parameterized action *#robot*, where an external

&sensor predicate enables to access sensor data.

$$\begin{aligned} \#robot[clean, kitchen]\{c, 2\} &\leftarrow night \\ \#robot[clean, bedroom]\{c, 2\} &\leftarrow day \\ \#robot[goto, charger]\{b, 1\} &\leftarrow \&sensor[bat](low) \\ night \vee day &\leftarrow \end{aligned}$$

Informally, in the night the kitchen should be cleaned, and during daytime the bedroom; if the battery is low, the robot needs to go to the charger. The option b (brave) makes charging mandatory, while other actions with option c (cautious) are only executed if they occur in every answer set. By the disjunctive fact, this is not the case. Precedence 1 of the charging action makes the robot recharge (if needed) before any cleaning. □

Constraint HEX programs. *Constraint Answer Set Programming (CASP)* (see e.g. [71,81]) combines ASP with constraint programming [3]. A well-known implementation is the clingcon system [86], which integrates GRINGO, CLASP and the constraint solver GECODE. Constraints can be encoded in plain ASP using builtin predicates, but this quickly produces groundings of unmanageable size; hence, a genuine support of constraints in ASP is reasonable, which can hide instances of constraint variables in the constraint solver. Dedicated CASP do not allow to integrate background theories other than constraints, which motivated an integration of CASP with HEX programs to *constraint HEX programs* [93]. Constraint HEX programs are strictly more general than CASP programs, as in addition to constraint atoms also external atoms can be used. Informally, a constraint HEX program may contain besides ordinary and external atoms also *constraint atoms*. The latter are comparisons of arithmetic expressions such as $x + y < 10$, where x and y are constraint variables which range over a certain domain. Different from ASP variables, constraint variables are global, i.e., each occurrence in a program is bound to the same value; thus, the atoms $x < 10$ and $x > 20$ can never be jointly true, even if they occur in different rules. For evaluating constraint HEX programs, constraint atoms are rewritten to auxiliary atoms in rule heads and bodies, e.g., $con(x, +, y, <, 10)$ for the above expression. Additionally, a constraint

$$\leftarrow \textbf{not } \&check[con, sum]()$$

eliminates answer sets where the extension of predicate *con* contains an inconsistent set of conditions over the constraint variable theory.

HEX programs with nested program calls. Notably, DLVHEX can be used to 'call' HEX programs from other HEX programs (called *host programs*). Specifically, one can process the collection of answer sets of a different program and can for instance reason on top of it. To this end, dedicated external atoms for evaluating subprograms and inspecting their answer sets are available [45, 91].

When a subprogram call (corresponding to the evaluation of a special external atom) is encountered in the host program, the external atom internally creates another instance of DLVHEX to evaluate the subprogram. The result is

then stored in an *answer cache* and gets a unique *handle* that can be later used to reference the result and access its components (e.g., predicate names, literals, arguments) via other external atoms. The subprogram can either be *directly embedded* in the host program, or *stored in a separate file*. In the latter case, code reuse is easy and libraries for solving re-occurring subproblems in ASP applications, e.g., graph problems or combinatorial optimization problems, can be built, where code updates are automatically reflected in the call program.

The MELD belief merging system deals with merging *collections of belief sets* [88, 91], which are roughly sets of classical ground literals. A merging strategy is defined by tree-shaped *merging plans*, whose leaves are the collections of belief sets to be merged, and whose inner nodes are *merging operators* (provided by the user). The structure is akin to syntax trees of terms. The automatic evaluation of tree-shaped merging plans is based on nested HEX programs; it proceeds bottom-up, where every step requires inspection of the subresults, i.e., accessing the answer sets of subprograms. In fact, the need for such processing has led to develop nested HEX program.

Interactive ASP. Interactive applications based on ASP can be realized with the Answer Set Application Programming (ASAP) framework [95] where incoming events (e.g., keyboard inputs) are processed by ASP and the application state is managed using state variables (as in planning, where these are called fluents). An ASAP program is rewritten to a HEX program where each evaluation obtains fluent values and event information via HEX external atoms. This is a *hybrid* HEX use scenario: an ASAP program is rewritten into a HEX program, transforming fluent atoms into regular atoms and adding rules containing external atoms. At the same time an ASAP program can use arbitrary external atoms, e.g., for string processing. This use scenario combines computation outsourcing (string processing) with information outsourcing (events and fluents) and moreover applies HEX for interfacing with the real world.

Multi-context systems. Heterogeneous nonmonotonic multi-context systems (MCSs) [12] are a formalism for interlinking multiple knowledge based systems called *contexts*. This formalism is on an evolution line of multi-context systems that goes back to seminal work of John McCarthy [80] and which has been further developed by the Trento school [15, 61, 62]. The MCS formalism abstracts from the knowledge representation language and models context semantics in terms of accepted *belief sets*. The latter are abstractly modeled as naked sets whose elements (i.e., the beliefs) need not bear logical structure. The contexts are interlinked by so called *bridge rules* which add formulas to the knowledge base of a context depending on the presence and/or absence of beliefs from the belief sets of other contexts. The MCS formalism is suitable for modeling many Semantic Web scenarios where distributed knowledge repositories interact, e.g., [10, 11, 79], and MCSs have been adapted for the Ambient Intelligence domain [8] and for modularly combining nonmonotonic rules bases in the MWeb approach [2].

The semantics of an MCS is given in terms of *equilibria*, which are global states that consist of acceptable belief sets for each context, such that all bridge rules are satisfied. Equilibria computation has been realized in a tool based on a HEX program [9], in which external atoms *outsource contextual reasoning*

and check whether a context accepts a certain belief set. This application hides HEX within a tool that realizes MCS semantics and inconsistency analysis [36].

6.2 Related Work

Because there are many scenarios where it is more natural, and often more efficient, to outsource some information or computation in the context of declarative problem solving, a number of approaches have been developed for this purpose, realizing different degrees of integration.

Motivated by the need for integration of data in commercial relational databases, extensions of dlv have been developed that allow to access external data. The dlv^{DB} system [98, 99] offers via an ODBC interface access to dispersed relational databases, where both direct (remote) execution of possibly recursive queries on databases and main memory execution (after loading the databases) are supported. The ontodlv system [92], allows the user to retrieve information from OWL ontologies, which can be utilized in a genuine ontology representation language that extends ASP with features such as classes, inheritance, relations and axioms.

DLV-EX programs [18] represent an early generic integration approach, which enables bidirectional communication with an external source, and allows the introduction of new terms by value invention into an answer set program. However, the interaction is more restricted than in the case of HEX since only terms can be used as inputs to external sources and thus, e.g., nonmonotonic aggregates cannot be expressed in this formalism.

The CLINGO system also provides a mechanism for importing the extension of user-defined predicates [55] similar to DLV-EX, but they are different from external atoms in HEX in that their evaluation is not interleaved with the solving process. For this, GRINGO supports custom functions (implemented in the scripting languages Lua or Python) which are evaluated during the program grounding and thus compiled away prior to the solving step. They are intended to be used as customizable built-in atoms, but no cyclic dependencies are possible.

Recently, CLINGO 5 has been released [54], which provides generic interfaces for integrating theory solving into ASP. A main difference between ASP modulo theories solving in CLINGO 5 and the HEX-framework consists in the fact that unfounded support over theory atoms is allowed by the semantics defined for CLINGO, which would violate the minimality criterion w.r.t. the FLP-reduct in HEX. This can be illustrated by the following example.

Example 35. Consider the program $\Pi = \{p \leftarrow \&id[p]()\}$, where $\&id[p]()$ is true iff p is true. Then Π has the answer set $\mathbf{A}_1 = \emptyset$; but $\mathbf{A}_2 = \{p\}$ is not an answer set because the support for p is not founded and thus, it is not a minimal model of the FLP-reduct.

Consequently, a more sophisticated minimality check has to be applied in DLVHEX, lifting the computational complexity of the formalism.[20]

[20] Deciding the existence of an answer set of a ground HEX program in the presence of nonmonotonic external atoms that are decidable in polynomial time is Σ_2^p-complete already for Horn programs [35].

Moreover, even though the CLINGO system moves into a similar direction as DLVHEX by facilitating the integration of external reasoners, the perspectives taken by the two systems are different, and their roles can be viewed as somewhat orthogonal.

While theory atoms are interrelated via an external theory in CLINGO, where the consistency of their truth assignments is usually checked during theory propagation, the truth value of external atoms in HEX depends on the evaluation of ordinary atoms representing their input. Thus, the focus of the HEX-approach is more on input-output relations over external atoms, which are easy to understand from a user's perspective and can be used to call external sources in an API-like fashion.

As a result, external atoms have a number of distinguishing features, which are tailored to their specific role in the HEX-framework. For instance, external source properties as described in Section 4 constitute a user-friendly high-level interface for steering the external evaluation process, which has to be implemented manually for each theory in CLINGO's propagation methods.

The input-output structure of external atoms facilitates the introduction of constants by value invention relying on the liberal safety condition for HEX programs, which is of special interest for applications in the area of the Semantic Web. There is no comparable mechanism for value invention in CLINGO 5 as new values cannot be imported based on the respective answer set, and theory solving is performed w.r.t. the pre-grounded program.

On the other side, CLINGO 5 is well-suited for system development and powerful solver building, by providing a comprehensive and rich infrastructure at the low technical levels for integrating theory reasoning into CLINGO, which is accessible through an interface. This novel interface will be exploited in future versions of DLVHEX, which benefits a lot from the CLINGO advances.

Besides CLINGO, the WASP solver was recently extended with support for general external Python propagators [25]. Furthermore there are extensions of ASP towards the integration of specific external sources. Examples are constraint ASP as an integration of ASP with constraint programming as realized e.g. in clingcon [86] lc2casp [16], ezcsp [4], and EZSMT [97]. The latter is like mingo [76] an SMT-based solver for constraint ASP; other formalisms that extend ASP with SMT are dingo [66], which uses difference logic, and ASPMT [69]. For an overview of systems that combine ASP with constraint solving and other theories, we refer to [72].

Similar to SMT [85], where usually only specific theories are considered, the mentioned approaches rely on a tailored integration of an external solver and hence, can easily leverage the propagation capabilities of the respective solver. The aim of the HEX formalism differs in that its goal is to enable a broad range of users to implement custom external sources and to harness efficient solving techniques for HEX programs. Moreover, clingcon and approaches in SMT usually only consider monotonic external theories, which facilitates the integration of their evaluation into the respective solving algorithm. In contrast, HEX allows for the integration of arbitrary external sources through a general interface and their flexible combination; the other use cases correspond to special cases thereof.

7 Conclusion

Arriving at the end, we give a summary and discuss ongoing work. Pointers to further resources regarding DLVHEX and ASP can be found in the appendix.

Summary. The HEX formalism extends ASP with access to external sources through an API-style interface, which has been implemented in the DLVHEX-system and has been fruitfully deployed to various applications. In this paper, we introduced the formalism and focused on its application to practical problems in KR and Semantic Web.

To this end, we first introduced HEX programs as a generalization of answer set programs, which constitute logic programs interpreted under the stable model semantics. In ASP, a problem can be solved by modeling it as an answer set program using variables, grounding the program to obtain a variable-free program, and using an answer set solver to compute (possibly multiple) answer sets of the program. Then, each answer set corresponds to a solution of the initial problem. By extending ASP with external atoms, HEX enables a bidirectional interaction between an answer set program and external sources of computation.

For modeling problems utilizing external atoms, we first presented the general methodology as a strict generalization of the common methodology for designing an ASP encoding. In particular, the prominent guess and check paradigm can be seamlessly combined with external sources, both in the guessing and in the checking part. Also other ASP techniques, such as saturation, can be used with external sources. We then discussed two typical types of external sources for computation outsourcing and for information outsourcing, respectively, and for combinations thereof. We further demonstrated the usage of external sources in existing applications in the areas of the Semantic Web and planning.

Subsequently, we presented the DLVHEX system, which implements a feature-rich solver for HEX programs. There, we described the architecture of DLVHEX and its practical usage. In DLVHEX, external sources that are used by the system for the evaluation of external atoms can be implemented via a user-friendly interface in C++ and Python. We also discussed how properties of external sources can be exploited for solving and demonstrated the configuration of DLVHEX for use cases with specific properties, e.g. with the need for introducing new values.

Finally, we integrated insights from previous sections by showcasing the development of a HEX encoding for a realistic application scenario, by following the methodology for designing HEX programs and using different features of the DLVHEX system. In addition, we reviewed related formalisms and applications where HEX is used for implementing language extensions and backends for other formalisms, in the last section.

Outlook. Current developments regarding the HEX formalism and DLVHEX comprise the design and implementation of new solving techniques for improving the efficiency of the formalism in general, as well as for specific classes of programs. At the same time, the goal is to relieve the user from the burden to configure the system manually for each type of problem, while still

profiting from performance gains. For instance, different static heuristics for partial evaluation of external atoms and minimization of nogoods have been introduced [44], and future work in this direction concerns the development of dynamic evaluation heuristics that adjust the frequency of external evaluations based on the amount of information gained from previous external calls.

Moreover, recently a lazy grounding solver has been integrated into the DLVHEX system, which exhibits promising results for classes of programs where the grounding bottleneck of ASP is an issue. This issue is even more challenging to tackle within the framework of HEX due to the need for grounding external atoms, which are largely black boxes from the viewpoint of the solver. Lazy grounding avoids an exponential blowup of the grounding by interleaving grounding and solving, whereby rules are grounded on-the-fly depending on the satisfaction of their bodies.

Furthermore, since HEX has already been applied to many different problems from the area of KR, another focus of ongoing work is on exploring new application areas for HEX to combine approaches that are different in nature, for solving concrete problems. For instance, external atoms could be utilized to integrate probabilistic methods into ASP for tackling problems from the area of *Statistical Relational Learning* [60], where complex relational as well as uncertain information is required simultaneously.

Acknowledgments. We appreciate the review feedback and are thankful for the detailed suggestions in it to improve the presentation of the material in this article. We also thank Roland Kaminski and Torsten Schaub for comments regarding CLINGO.

A Further Resources

The following list contains some further links to web resources regarding practical aspects of ASP, HEX programs and the DLVHEX system, and summarizes those already given in the paper.

- All executable examples from this paper are available under:
  ⇒ `https://github.com/hexhex/manual/tree/master/RW2017/`
- Slides of a tutorial considering the topic "ASP for the Semantic Web" and many executable ASP/HEX-examples related to Semantic Web applications can be found at:
  ⇒ `http://asptut.gibbi.com/`
- A tutorial paper providing a gentle introduction to ASP and an overview over programming techniques (also considering Semantic Web applications) is available at:
  ⇒ `http://www.kr.tuwien.ac.at/staff/tkren/pub/2009/rw2009-asp.pdf`
- The main website of DLVHEX contains all relevant information about the system and existing plugins, and many further references to the relevant literature and related work:
  ⇒ `http://www.kr.tuwien.ac.at/research/systems/dlvhex/`

– An online demo of the DLVHEX system can be found at:
  ⇒ `http://www.kr.tuwien.ac.at/research/systems/dlvhex/demo.php`
– The easiest way to use DLVHEX is by downloading the pre-built binaries, which are available for Linus, OS X and Windows under:
  ⇒ `http://www.kr.tuwien.ac.at/research/systems/dlvhex/downloadb.html`
– The source code of DLVHEX and corresponding plugins is available on Github, which is also the best place for filing bug reports:
  ⇒ `https://github.com/hexhex/`
– The website of the *Potsdam Answer Set Solving Collection* (*Potassco*) is the main portal for CLINGO and related systems and tools, and contains a lot of additional information on them:
  ⇒ `https://potassco.org/`
– Material of the Potsdam ASP course can be found under:
  ⇒ `https://potassco.org/teaching/`
– The source code of CLINGO, clingcon, and related systems is publicly available at:
  ⇒ `https://github.com/potassco/`
– *ASPIDE* is an integrated development environment for ASP with a wide range of features facilitating the implementation of answer set programs:
  ⇒ `http://www.mat.unical.it/ricca/aspide`
– Slides of a tutorial covering ASPIDE and the development of answer set programs can be found at:
  ⇒ `https://www.mat.unical.it/ricca/downloads/rr2013-tutorial.pdf`
– A special issue of the *AI Magazine* has been dedicated to ASP, covering many different perspectives on the topic:
  ⇒ `http://aaai.org/ojs/index.php/aimagazine/issue/view/215/`

References

1. Mario Alviano, Carmine Dodaro, Wolfgang Faber, Nicola Leone, and Francesco Ricca. WASP: A native ASP solver based on constraint learning. In Pedro Cabalar and Tran Cao Son, editors, *Logic Programming and Nonmonotonic Reasoning, 12th International Conference, LPNMR 2013, Corunna, Spain, September 15-19, 2013. Proceedings*, volume 8148 of *Lecture Notes in Computer Science*, pages 54–66. Springer, 2013.
2. Anastasia Analyti, Grigoris Antoniou, and Carlos Viegas Damásio. MWeb: A principled framework for modular web rule bases and its semantics. *ACM Trans. Comput. Log.*, 12(2):17, 2011.
3. Krzysztof Apt. *Principles of Constraint Programming.* Cambridge University Press, New York, NY, USA, 2003.
4. Marcello Balduccini. Representing constraint satisfaction problems in answer set programming. In *Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP) at ICLP*, 2009.
5. Chitta Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving.* Cambridge Univ. Press, 2003.
6. Selen Basol, Ozan Erdem, Michael Fink, and Giovambattista Ianni. HEX programs with action atoms. In *Technical Communications of the International Conference on Logic Programming (ICLP)*, pages 24–33, 2010.

7. R. Ben-Eliyahu and R. Dechter. Propositional Semantics for Disjunctive Logic Programs. *Annals of Mathematics and Artificial Intelligence*, 12:53–87, 1994.
8. Antonis Bikakis and Grigoris Antoniou. Defeasible contextual reasoning with arguments in ambient intelligence. *IEEE Transactions on Knowledge and Data Engineering*, 22(11):1492–1506, 2010.
9. Markus Bögl, Thomas Eiter, Michael Fink, and Peter Schüller. The MCS-IE system for explaining inconsistency in multi-context systems. In *European Conference on Logics in Artificial Intelligence (JELIA)*, pages 356–359, 2010.
10. P Bouquet, F Giunchiglia, F van Harmelen, L Serafini, and H Stuckenschmidt. Contextualizing Ontologies. *Web Semantics: Science, Services and Agents on the World Wide Web*, 1(4):325–343, 2004.
11. Loris Bozzato and Luciano Serafini. Materialization Calculus for Contexts in the Semantic Web. In Thomas Eiter, Birte Glimm, Yevgeny Kazakov, and Markus Krötzsch, editors, *DL2013*, volume 1014 of *CEUR-WP*, pages 552 – 572. CEUR-WS.org, 2013.
12. Gerd Brewka and Thomas Eiter. Equilibria in heterogeneous nonmonotonic multi-context systems. In *AAAI Conference on Artificial Intelligence*, pages 385–390. AAAI Press, 2007.
13. Gerd Brewka, Thomas Eiter, and Miroslaw Truszczyński, editors. *AI Magazine: special issue on Answer Set Programming*. AAAI Press, 2016. Volume 37, number 3. Editorial pp. 5-6.
14. Gerhard Brewka, Thomas Eiter, and Miroslaw Truszczynski. Answer set programming at a glance. *Commun. ACM*, 54(12):92–103, 2011.
15. Gerhard Brewka, Floris Roelofsen, and Luciano Serafini. Contextual default reasoning. In Manuela M. Veloso, editor, *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007*, pages 268–273, 2007.
16. Pedro Cabalar, Roland Kaminski, Max Ostrowski, and Torsten Schaub. An ASP semantics for default reasoning with constraints. In Subbarao Kambhampati, editor, *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016*, pages 1015–1021. IJCAI/AAAI Press, 2016.
17. Andrea Calì, Georg Gottlob, and Andreas Pieris. Towards more expressive ontology languages: The query answering problem. *Artif. Intell.*, 193:87–128, 2012.
18. Francesco Calimeri, Susanna Cozza, and Giovambattista Ianni. External sources of knowledge and value invention in logic programming. *Annals of Mathematics and Artificial Intelligenc*, 50(3–4):333–361, 2007.
19. Francesco Calimeri, Wolfgang Faber, Martin Gebser, Giovambattista Ianni, Thomas Krennwallner Roland Kaminski, Nicola Leone, Francesco Ricca, and Torsten Schaub. ASP-Core-2 Input Language Format, 2013.
20. Francesco Calimeri, Michael Fink, Stefano Germano, Andreas Humenberger, Giovambattista Ianni, Christoph Redl, Daria Stepanova, Andrea Tucci, and Anton Wimmer. Angry-HEX: an artificial player for angry birds based on declarative knowledge bases. *IEEE Transactions on Computational Intelligence and AI in Games*, 8(2):128–139, 2016.
21. Francesco Calimeri, Michael Fink, Stefano Germano, Giovambattista Ianni, Christoph Redl, and Anton Wimmer. AngryHEX: an artificial player for angry birds based on declarative knowledge bases. In *National Workshop and Prize on Popularize Artificial Intelligence*, pages 29–35, 2013.
22. Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, and Riccardo Rosati. Tractable reasoning and efficient query answering in description logics: The *DL-Lite* family. *J. Autom. Reasoning*, 39(3):385–429, 2007.

23. Evgeny Dantsin, Thomas Eiter, Georg Gottlob, and Andrei Voronkov. Complexity and Expressive Power of Logic Programming. *ACM Computing Surveys*, 33(3):374–425, 2001.
24. Minh Dao-Tran, Thomas Eiter, and Thomas Krennwallner. Realizing default logic over description logic knowledge bases. In *European Conference on Symbolic and Quantitative Approaches to Reasoning with Uncertainty*, pages 602–613, 2009.
25. Carmine Dodaro, Francesco Ricca, and Peter Schüller. External propagators in WASP: Preliminary report. In Stefano Bistarelli, Andrea Formisano, and Marco Maratea, editors, *International Workshop on Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion (RCRA)*, volume 1745 of *CEUR Workshop Proceedings*, pages 1–9. CEUR-WS.org, November 2016.
26. Włodzimierz Drabent, Thomas Eiter, Giovambattista Ianni, Thomas Krennwallner, Thomas Lukasiewicz, and Jan Małuszyński. Hybrid reasoning with rules and ontologies. In Francois Bry and Jan Małuszyński, editors, *Semantic Techniques for the Web: The REWERSE perspective*, number 5500 in LNCS, chapter 1, pages 1–49. Springer, 2009.
27. Phan Minh Dung. On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games. *Artificial Intelligence*, 77(2):321–357, 1995.
28. T. Eiter and G. Gottlob. On the computational cost of disjunctive logic programming: Propositional case. *Annals of Mathematics and Artificial Intelligenc*, 15(3/4):289–323, 1995.
29. 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. doi:10.1017/S1471068415000113, `http://arxiv.org/abs/1507.01451`.
30. Thomas Eiter, Michael Fink, Thomas Krennwallner, and Christoph Redl. Conflict-driven ASP solving with external sources. *Theory and Practice of Logic Programming*, 12(4-5):659–679, 2012.
31. Thomas Eiter, Michael Fink, Thomas Krennwallner, and Christoph Redl. Liberal Safety Criteria for HEX-Programs. In Marie des Jardins and Michael Littman, editors, *AAAI Conference on Artificial Intelligence (AAAI)*. AAAI Press, 2013.
32. Thomas Eiter, Michael Fink, Thomas Krennwallner, and Christoph Redl. Domain expansion for ASP-programs with external sources. Technical Report INFSYS RR-1843-14-02, Institut für Informationssysteme, Technische Universität Wien, A-1040 Vienna, Austria, September 2014.
33. Thomas Eiter, Michael Fink, Thomas Krennwallner, and Christoph Redl. HEX-programs with existential quantification. In *International Conference on Applications of Declarative Programming and Knowledge Management (INAP)*, 2014.
34. Thomas Eiter, Michael Fink, Thomas Krennwallner, and Christoph Redl. Domain expansion for ASP-programs with external sources. *Artif. Intell.*, 233:84–121, 2016.
35. Thomas Eiter, Michael Fink, Thomas Krennwallner, Christoph Redl, and Peter Schüller. Efficient HEX-program evaluation based on unfounded sets. *Journal of Artificial Intelligence Research*, 49:269–321, 2014.
36. Thomas Eiter, Michael Fink, Peter Schüller, and Antonius Weinzierl. Finding Explanations of Inconsistency in Multi-Context Systems. *Artificial Intelligence*, 216:233–274, November 2014.

37. Thomas Eiter and Georg Gottlob. On the computational cost of disjunctive logic programming: Propositional case. *Ann. Math. Artif. Intell.*, 15(3-4):289–323, 1995.
38. Thomas Eiter, Georg Gottlob, and Helmuth Veith. Generalized quantifiers in logic programs. In *Generalized Quantifiers and Computation: 9th European Summer School in Logic, Language, and Information, ESSLLI 1997 Workshop, Aix-en-Provence, France, August 1997*, volume 1754 of *LNCS*, pages 72–98. Springer, 1997.
39. Thomas Eiter, Giovambattista Ianni, and Thomas Krennwallner. Answer set programming: A primer. In *Reasoning Web Summer School*, pages 40–110, 2009.
40. Thomas Eiter, Giovambattista Ianni, Thomas Krennwallner, and Roman Schindlauer. Exploiting conjunctive queries in description logic programs. *Annals of Mathematics and Artificial Intelligenc*, 53(1–4):115–152, 2008.
41. Thomas Eiter, Giovambattista Ianni, Thomas Lukasiewicz, Roman Schindlauer, and Hans Tompits. Combining answer set programming with description logics for the semantic web. *Artificial Intelligence*, 172(12-13):1495–1539, 2008.
42. Thomas Eiter, Giovambattista Ianni, Roman Schindlauer, and Hans Tompits. A Uniform Integration of Higher-Order Reasoning and External Evaluations in Answer-Set Programming. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 90–96. Professional Book Center, 2005.
43. Thomas Eiter, Giovambattista Ianni, Roman Schindlauer, and Hans Tompits. Effective integration of declarative rules with external evaluations for semantic-web reasoning. In *European Semantic Web Conference*, pages 273–287, 2006.
44. Thomas Eiter, Tobias Kaminski, Christoph Redl, and Antonius Weinzierl. Exploiting partial assignments for efficient evaluation of answer set programs with external source access. In *IJCAI*, pages 1058–1065. IJCAI/AAAI Press, 2016.
45. Thomas Eiter, Thomas Krennwallner, and Christoph Redl. HEX-Programs with Nested Program Calls. In *Applications of Declarative Programming and Knowledge Management (INAP 2011)*, pages 1–10. Springer, 2013.
46. Thomas Eiter, Mustafa Mehuljic, Christoph Redl, and Peter Schüller. User guide: dlvhex 2.x. Technical Report INFSYS RR-1843-15-05, Vienna University of Technology, Institute for Information Systems, 2015.
47. Thomas Eiter, Christoph Redl, and Peter Schüller. Problem solving using the HEX family. In Christoph Beierle, Gerhard Brewka, and Matthias Thimm, editors, *Computational Models of Rationality - Essays dedicated to Gabriele Kern-Isberner on the occasion of her 60th birthday, Tributes*, pages 150–174. College Publications, January 2016.
48. Esra Erdem, Michael Gelfond, and Nicola Leone. Applications of answer set programming. *AI Magazine*, 37(3):53–68, 2016.
49. Esra Erdem, Volkan Patoglu, and Peter Schüller. A Systematic Analysis of Levels of Integration between Low-Level Reasoning and Task Planning. *AI Communications*, 29(2):319–349, 2016.
50. Wolfgang Faber, Nicola Leone, and Gerald Pfeifer. Recursive aggregates in disjunctive logic programs: Semantics and complexity. In *European Conference on Logics in Artificial Intelligence (JELIA)*, pages 200–212. Springer, 2004.
51. Wolfgang Faber, Nicola Leone, and Gerald Pfeifer. Semantics and complexity of recursive aggregates in answer set programming. *Artificial Intelligence*, 175(1):278–298, 2011.
52. Michael Fink, Stefano Germano, Giovambattista Ianni, Christoph Redl, and Peter Schüller. ActHEX: implementing HEX programs with action atoms. In Pedro Cabalar and TranCao Son, editors, *International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, pages 317–322. Springer, 2013.

53. 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.
54. Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, Torsten Schaub, and Philipp Wanko. Theory solving made easy with clingo 5. In *ICLP (Technical Communications)*, volume 52 of *OASICS*, pages 2:1–2:15. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.
55. Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. Clingo = ASP + control: Preliminary report. *CoRR*, abs/1405.3694, 2014.
56. Martin Gebser, Benjamin Kaufmann, Roland Kaminski, Max Ostrowski, Torsten Schaub, and Marius Thomas Schneider. Potassco: The Potsdam Answer Set Solving Collection. *AI Commun.*, 24(2):107–124, 2011.
57. Martin Gebser, Benjamin Kaufmann, and Torsten Schaub. Conflict-driven answer set solving: From theory to practice. *Artificial Intelligence*, 187–188:52–89, 2012.
58. M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programming. In R. Kowalski and K. Bowen, editors, *Logic Programming: Proceedings of the 5th International Conference and Symposium*, pages 1070–1080. MIT Press, 1988.
59. M. Gelfond and V. Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *Next Generation Computing*, 9(3–4):365–386, 1991.
60. Lise Getoor. *Introduction to statistical relational learning*. MIT press, 2007.
61. Chiara Ghidini and Fausto Giunchiglia. Local models semantics, or contextual reasoning=locality+compatibility. *Artif. Intell.*, 127(2):221–259, 2001.
62. Fausto Giunchiglia and Luciano Serafini. Multilanguage hierarchical logics or: How we can do without modal logics. *Artif. Intell.*, 65(1):29–70, 1994.
63. Giray Havur, Guchan Ozbilgin, Esra Erdem, and Volkan Patoglu. Geometric Rearrangement of Multiple Movable Objects on Cluttered Surfaces: A Hybrid Reasoning Approach. In *International Conference on Robotics and Automation (ICRA)*, pages 445–452, 2014.
64. J. Heflin and H. Munoz-Avila. Lcw-based agent planning for the semantic web. In A. Pease, editor, *Ontologies and the Semantic Web*, number WS-02-11 in AAAI Technical Report, pages 63–70, Menlo Park, CA, 2002. AAAI Press.
65. Robert Hoehndorf, Frank Loebe, Janet Kelso, and Heinrich Herre. Representing default knowledge in biomedical ontologies: Application to the integration of anatomy and phenotype ontologies. *BMC Bioinformatics*, 8(1):377, 2007.
66. Tomi Janhunen, Guohua Liu, and Ilkka Niemelä. Tight integration of non-ground answer set programming and satisfiability modulo theories. In Pedro Cabalar, David Mitchell, David Pearce, and Eugenia Ternovska, editors, *Informal Proceedings of the 1st Workshop on Grounding and Transformations for Theories with Variables (GTTV'11), LPNMR, Vancouver, BC, Canada May 16th, 2011*, pages 1–14, 2013. Online available at `http://www.dc.fi.udc.es/GTTV11/GTTV-Proc.pdf`.
67. Roland Kaminski, Torsten Schaub, and Philipp Wanko. A tutorial on hybrid answer set solving with clingo. In *Reasoning Web Summer School*, 2017. To appear.
68. O. Lassila and R.R. Swick. Resource description framework (RDF) model and syntax specification, 1999. `www.w3.org/TR/1999/REC-rdf-syntax-19990222`.
69. Joohyung Lee and Yunsong Meng. Answer set programming modulo theories and reasoning about continuous changes. In Francesca Rossi, editor, *IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013*, pages 990–996. IJCAI/AAAI, 2013.

70. N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The DLV System for Knowledge Representation and Reasoning. *ACM Transactions on Computational Logic (TOCL)*, 7(3):499–562, July 2006.
71. Yuliya Lierler. Relating constraint answer set programming languages and algorithms. *Artificial Intelligence*, 207:1–22, February 2014.
72. Yuliya Lierler, Marco Maratea, and Francesco Ricca. Systems, engineering environments, and competitions. *AI Magazine*, 37(3):45–52, 2016.
73. Vladimir Lifschitz. Answer Set Programming and Plan Generation. *Artificial Intelligence*, 138:39–54, 2002.
74. Vladimir Lifschitz. Thirteen definitions of a stable model. In Andreas Blass, Nachum Dershowitz, and Wolfgang Reisig, editors, *Fields of Logic and Computation, Essays Dedicated to Yuri Gurevich on the Occasion of His 70th Birthday*, volume 6300 of *Lecture Notes in Computer Science*, pages 488–503. Springer, 2010.
75. Fangzhen Lin and Yuting Zhao. ASSAT: computing answer sets of a logic program by SAT solvers. *Artificial Intelligence*, 157(1–2):115–137, 2004.
76. Guohua Liu, Tomi Janhunen, and Ilkka Niemelä. Answer set programming via mixed integer programming. In Gerhard Brewka, Thomas Eiter, and Sheila A. McIlraith, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Thirteenth International Conference, KR 2012, Rome, Italy, June 10-14, 2012*. AAAI Press, 2012.
77. Victor W. Marek and Mirosław Truszczyński. Stable Models and an Alternative Logic Programming Paradigm. In *The Logic Programming Paradigm – A 25-Year Perspective*, pages 375–398. Springer, 1999.
78. W. Marek and M. Truszczyński. Autoepistemic logic. *Journal of the ACM*, 38(3):588–619, 1991.
79. Wolfgang May, José Júlio Alferes, and Ricardo Amador. Active rules in the semantic web: Dealing with language heterogeneity. In *International Workshop on Rules and Rule Markup Languages for the Semantic Web*, pages 30–44, 2005.
80. John McCarthy. Notes on formalizing context. In Ruzena Bajcsy, editor, *Proceedings of the 13th International Joint Conference on Artificial Intelligence. Chambéry, France, August 28 - September 3, 1993*, pages 555–562. Morgan Kaufmann, 1993.
81. Veena S Mellarkod, Michael Gelfond, and Yuanlin Zhang. Integrating Answer Set Programming and Constraint Logic Programming. *Annals of Mathematics and Artificial Intelligenc*, 53(1-4):251–287, 2008.
82. Alessandro Mosca and Diego Bernini. Ontology-driven geographic information system and dlvhex reasoning for material culture analysis. In *Italian Workshop RiCeRcA at ICLP*, 2008.
83. Boris Motik and Riccardo Rosati. Reconciling description logics and rules. *J. ACM*, 57(5):30:1–30:62, 2010.
84. Ilkka Niemelä. Logic programming with stable model semantics as constraint programming paradigm. *Annals of Mathematics and Artificial Intelligenc*, 25(3–4):241–273, 1999.
85. Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *J. ACM*, 53(6):937–977, 2006.
86. Max Ostrowski and Torsten Schaub. ASP modulo CSP: the clingcon system. *Theory and Practice of Logic Programming (TPLP)*, 12(4-5):485–503, 2012.
87. Axel Polleres. From SPARQL to rules (and back). In *International Conference on World Wide Web (WWW)*, pages 787–796. ACM, 2007.

88. Christoph Redl. Development of a belief merging framewerk for dlvhex. Master's thesis, Vienna University of Technology, A-1040 Vienna, Karlsplatz 13, 2010.
89. Christoph Redl. *Answer Set Programming with External Sources: Algorithms and Efficient Evaluation*. PhD thesis, Vienna University of Technology, 2014.
90. Christoph Redl. The dlvhex system for knowledge representation: recent advances (system description). *TPLP*, 16(5-6):866–883, 2016.
91. Christoph Redl, Thomas Eiter, and Thomas Krennwallner. Declarative belief set merging using merging plans. In *International Symposium on Practical Aspects of Declarative Languages (PADL)*, pages 99–114. Springer, 2011.
92. Francesco Ricca, Lorenzo Gallucci, Roman Schindlauer, Tina Dell'Armi, Giovanni Grasso, and Nicola Leone. OntoDLV: An ASP-based system for enterprise ontologies. *J. Log. Comput.*, 19(4):643–670, 2009.
93. Alessandro De Rosis, Thomas Eiter, Christoph Redl, and Francesco Ricca. Constraint answer set programming based on HEX-programs. In *Eighth Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP 2015), August 31, 2015, Cork, Ireland*, August 2015. Accepted for publication.
94. Roman Schindlauer. *Answer Set Programming for the Semantic Web*. PhD thesis, Vienna University of Technology, Vienna, Austria, 2006.
95. Peter Schüller and Antonius Weinzierl. Answer Set Application Programming: a Case Study on Tetris. In Marina De Vos, Thomas Eiter, Yuliya Lierler, and Francesca Toni, editors, *International Conference on Logic Programming (ICLP), Technical Communications*, volume 1433. CEUR-WS.org, 2015.
96. Patrik Simons, Ilkka Niemelä, and Timo Soininen. Extending and implementing the stable model semantics. *Artif. Intell.*, 138(1-2):181–234, 2002.
97. Benjamin Susman and Yuliya Lierler. SMT-based constraint answer set solver EZSMT (system description). In Manuel Carro, Andy King, Neda Saeedloei, and Marina De Vos, editors, *Technical Communications of the 32nd International Conference on Logic Programming, ICLP 2016 TCs, October 16-21, 2016, New York City, USA*, volume 52 of *OASICS*, pages 1:1–1:15. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.
98. Giorgio Terracina, Erika De Francesco, Claudio Panetta, and Nicola Leone. Enhancing a DLP system for advanced database applications. In Diego Calvanese and Georg Lausen, editors, *Web Reasoning and Rule Systems, Second International Conference, RR 2008, Karlsruhe, Germany, October 31-November 1, 2008. Proceedings*, volume 5341 of *Lecture Notes in Computer Science*, pages 119–134. Springer, 2008.
99. Giorgio Terracina, Nicola Leone, Vincenzino Lio, and Claudio Panetta. Experimenting with recursive queries in database and logic programming systems. *TPLP*, 8(2):129–165, 2008.
100. Jesia Zakraoui and Wolfgang L. Zagler. A method for generating CSS to improve web accessibility for old users. In *International Conference on Computers Helping People with Special Needs (ICCHP)*, pages 329–336, 2012.
101. Hande Zirtiloğlu and Pinar Yolum. Ranking semantic information for e-government: complaints management. In *International Workshop on Ontology-supported business intelligence (OBI)*. ACM, 2008.