The Hexlite Solver

Lightweight and Efficient Evaluation of HEX Programs

Peter Schüller 

Technische Universität Wien
Institut für Logic and Computation
Knowledge-based Systems Group
ps@kr.tuwien.ac.at

Abstract. HEXLITE is a lightweight solver for the HEX formalism which integrates Answer Set Programming (ASP) with external computations. The main goal of HEXLITE is efficiency and simplicity, both in implementation as well as in installation of the system. We define the Pragmatic HEX Fragment which permits to partition external computations into two kinds: those that can be evaluated during the program instantiation phase, and those that need to be evaluated during the answer set search phase. HEXLITE is written in PYTHON and suitable for evaluating this fragment with external computations that are realized in PYTHON. Most performance-critical tasks are delegated to the PYTHON module of CLINGO. We demonstrate that the Pragmatic HEX Fragment is sufficient for many use cases and that it permits HEXLITE to have superior performance compared to the DLVHEX system in relevant application scenarios.

1 Introduction

The HEX formalism [7] facilitates the combination of logic programming and external computations in other programming paradigms and facilitates the integration of logical reasoning with diverse other reasoning methods such as motion planning [18], description logics [12], or sub-symbolic reasoning [15].

Different from externals in GRINGO and the PYTHON interface of CLINGO, the HEX formalism provides uniform and generic syntax and semantics for external computations that influence (a) the instantiation of the program (by performing value invention), and (b) the solving process (by computing truth values relative to interpretations).

Computing HEX semantics requires the evaluation of external computations both during grounding and during search, in an interleaved fashion. The main solver implementation for the HEX formalism is the DLVHEX[1] system [11]. While DLVHEX implements the full HEX language, it performs a lot of analysis and preprocessing to be able to deal with all eventualities of combinations of external computations, which makes it unnecessarily slow in several relevant application scenarios.

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

In order to obtain performance when evaluating HEX programs, we here present a new HEX solver that is lightweight and efficient, at the cost of handling only a fragment of HEX. (This fragment is sufficient for many applications.)

In this work, we make the following contributions.

• We define the Pragmatic HEX Fragment (PHF) that permits to separate external computations into those that can be evaluated completely during instantiation and those that can be evaluated completely during search. We show several application scenarios where the PHF is sufficient.
• We describe and provide the HEXLITE solver that provides lightweight and efficient evaluation machinery for the PHF. The solver is implemented in PYTHON and uses the CLINGO PYTHON API as a backend for ASP grounding and search. HEXLITE rewrites both classes of external atoms in different ways before passing the rules to CLINGO for evaluation. As a main benefit of this architecture, HEXLITE supports the full ASP input language of CLINGO, including weak constraints, choice rules, aggregates, expansion terms, and builtin arithmetics.
• We experimentally compare the HEXLITE solver with DLVHEX on two application scenarios that gave rise to the development of HEXLITE: cost-based abduction [28] and RDF processing [16]. Our experiments show that HEXLITE performs better than DLVHEX in these applications. As HEXLITE uses the same PYTHON API as DLVHEX, we use the same plugin with both solvers, which makes the comparison very realistic.

HEXLITE can be installed via CONDA or PIP and is available as open source.[2]

2 Preliminaries

We give syntax and semantics of the HEX formalism [7, 14] which generalizes logic programs under answer set semantics [22] with external computations.

2.1 HEX Syntax

Let $\mathcal{C}$, $\mathcal{X}$, and $\mathcal{G}$ be mutually disjoint sets whose elements are called *constant names*, *variable names*, and *external predicate names*, respectively. Usually, elements from $\mathcal{X}$ and $\mathcal{C}$ are denoted with first letter in upper case and lower case, respectively; while elements from $\mathcal{G}$ are prefixed with '&'. Elements from $\mathcal{C} \cup \mathcal{X}$ are called *terms*. An (ordinary) *atom* is a tuple $p(Y_1, \ldots, Y_n)$ where $p \in \mathcal{C}$ is a predicate name and $Y_1, \ldots, Y_n$ are terms and $n \geq 0$ is the *arity* of the atom. The atom is *ground* if all its terms are constants. An *external atom* is of the form $\&g[Y_1, \ldots, Y_n](X_1, \ldots, X_m)$, where $Y_1, \ldots, Y_n$ and $X_1, \ldots, X_m$ are two lists of terms, called *input* and *output* lists, respectively, and $\&g \in \mathcal{G}$ is an external predicate name. We assume that input and output lists have fixed lengths $in(\&g) = n$ and $out(\&g) = m$.

A *rule* r is of the form $\alpha_1 \vee \cdots \vee \alpha_k \leftarrow \beta_1, \ldots, \beta_n, not\ \beta_{n+1}, \ldots, not\ \beta_m$ with $m, k \geq 0$ where all α_i are atoms and all β_j are either atoms or external atoms. We

[2] `https://github.com/hexhex/hexlite`

let $H(r) = \{\alpha_1, \ldots, \alpha_k\}$ and $B(r) = B^+(r) \cup B^-(r)$, where $B^+(r) = \{\beta_1, \ldots, \beta_n\}$ and $B^-(r) = \{\beta_{n+1}, \ldots, \beta_m\}$. A rule r is a *constraint*, if $H(r) = \emptyset$ and $B(r) \neq \emptyset$; a *fact*, if $B(r) = \emptyset$ and $H(r) \neq \emptyset$; and *nondisjunctive*, if $|H(r)| \leq 1$. We call r *ordinary*, if it contains only ordinary atoms.

A *HEX program* is a finite set P of rules. We call a program P *ordinary* (resp., *nondisjunctive*), if all its rules are ordinary (resp., nondisjunctive). We denote by $cons(P)$ the set of constant symbols occurring in a program P. Note, that we here assume that programs have no *higher-order* atoms (i.e., atoms of form $Y_0(Y_1, \ldots, Y_n)$ where $Y_0 \in \mathcal{X}$) because HEX-programs with higher-order atoms can easily be rewritten to HEX-programs without higher-order atoms [7].

A comprehensive introduction to HEX is given in [16].

2.2 Semantics

Given a rule r, the grounding $grnd(r)$ of r is obtained by replacing all variables with constants from $\mathcal{C}$. Given a HEX-program P, the *Herbrand base* HB_P of P is the set of all possible ground versions of atoms and external atoms occurring in P obtained by replacing variables with constants from $\mathcal{C}$. The grounding $grnd(P)$ of P is given by $grnd(P) = \bigcup_{r \in P} grnd(r)$. Importantly, the set of constants $\mathcal{C}$ that is 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 the semantics of the program.

We associate an $(n+m+1)$-ary Boolean function with every external predicate name $\&g \in \mathcal{G}$: $f_{\&g}$ assigns each tuple $(I, y_1, \ldots, y_n, x_1, \ldots, x_m)$ either 0 or 1, where $n = in(\&g)$, $m = out(\&g)$, $I \subseteq HB_P$, and $x_i, y_j \in \mathcal{C}$. This definition of external atom semantics is very general, and does not lend itself easily to computation. In HEX solver implementations (such as DLVHEX and HEXLITE), we use *extensional semantics* [7,14], defined as follows. External predicates have a type signature $t_1, \ldots, t_n$ such that input term t_i, $1 \leq i \leq n$, is either interpreted as a constant ($t_i = cons$) or as a name of a predicate with arity t_i ($t_i \in \mathbb{N}$). The external computation is restricted to depend only on the constant values of y_i whenever $t_i = cons$; and the extension of predicate y_i of arity t_i in I if $t_i \in \mathbb{N}$. The external computation is formalized as an *extensional evaluation function* $F_{\&g}$ that computes a set of ground output tuples $(x_1, \ldots, x_m)$: $(x_1, \ldots, x_m) \in F_{\&g}(I, y_1, \ldots, y_n)$ iff $f_{\&g}(I, y_1, \ldots, y_n, x_1, \ldots, x_m) = 1$.

An interpretation $I \subseteq HB_P$ is a *model* of an atom a, denoted $I \models a$, if a is an ordinary atom and $a \in I$. I is a model of a ground external atom $a = \&g[y_1, \ldots, y_n](x_1, \ldots, x_m)$ if $(x_1, \ldots, x_m) \in F_{\&g}(I, y_1, \ldots, y_n)$. Given a ground rule r, $I \models H(r)$ if $I \models a$ for some $a \in H(r)$; $I \models B(r)$ if $I \models a$ for all $a \in B^+(r)$ and $I \not\models a$ for all $a \in B^-(r)$; and $I \models r$ if $I \models H(r)$ whenever $I \models B(r)$. Given a HEX-program P, $I \models P$ if $I \models r$ for all $r \in grnd(P)$; the *FLP-reduct* of P with respect to $I \subseteq HB_P$, denoted fP^I, is the set of all $r \in grnd(P)$ such that $I \models B(r)$; $I \subseteq HB_P$ is an *answer set* of P if I is a minimal model of fP^I, and we denote by $\mathcal{AS}(P)$ the set of all answer sets of P.

3 The Pragmatic HEX Fragment (PHF)

We next define a fragment of HEX that permits to separate external computations into two classes: grounding-relevant and solving-relevant.

Definition 1. *A* HEX*-program P is* in the Pragmatic HEX Fragment (PHF) *iff each external atom of form $\&g[Y_1,\ldots,Y_n](X_1,\ldots,X_m)$ with type signature $t_1,\ldots,t_n$ satisfies one of the following conditions:*
(G) $t_i = cons$ for all i, $1 \leq i \leq n$; or
(S) $m = 0$ and there is at least one type t_i, $1 \leq i \leq n$, such that $t_i \in \mathbb{N}$.

Type (G). External atoms that satisfy condition (G) can be *evaluated during instantiation* of program P because their computation does not depend on I. These external computations can perform value invention: they can produce constants in the output tuple $X_1,\ldots,X_m$ that do not exist in P.

Example 1. We can use an external atom of form $\&rdf[U](S,P,O)$ of type (G) to accesses a RDF [24] triple stores [16]. The function $F_{\&rdf}(I, U)$ returns all tuples (S,P,O) that are obtained from the RDF graph accessible at URI U. Intuitively, this external computation imports the RDF graph into the HEX program and makes all its constants accessible to HEX. The computation does not depend on I and returns arbitrary strings (value invention).

Type (S). External atoms that satisfy condition (S) have an empty output tuple and can therefore not produce any output apart from their own truth value. This makes it possible to instantiate rules that contain such external atoms, without performing the associated external computation; the external computation needs to be performed only *during the answer set search* phase, when the interpretation I is available.

Example 2. We can use external atoms of form $\&transitive[p]()$ of type (S) to verify whether an extension is transitive in the interpretation [28]. The function $F_{\&transitive}(I,p)$ returns the empty tuple if the extension of p in I is a transitive relation. Otherwise, it returns no tuple.

External atoms of type (S) have the possibility to create nogoods that relate the truth value of the ground external atom with parts of I that are relevant for computing that truth value. This feature, which also exists in DLVHEX, can be used for increasing evaluation performance by guiding the solver towards answer set candidates that are compatible with external computation results.

3.1 Properties

Clearly, the two classes (G) and (S) of external atoms are mutually exclusive. Those external atoms of type (G) that have no output terms (i.e., $m = 0$) could be evaluated during the solving process because their evaluation is not necessary for instantiating the ground program. However, we found it useful to evaluate

(and therefore eliminate) as many external atoms as possible already during program instantiation.

Moreover, there are external atoms that fall neither into class (G) nor into class (S), for example an external atom $\&sum[pred](X)$ that has one predicate input *pred* of arity 1 and realizes a summation aggregate with extensional evaluation function $F_{\&sum}(I, pred) := \{(X)\}$ where $X = \Sigma\{x \mid p(x) \in I\}$.

External atoms of type (G) can be evaluated (and eliminated) during instantiation of the program as shown in the following proposition.

Proposition 1. *Given a ground* HEX*-program P in PHF, an equivalent program P' can be produced by (1) omitting rules that contain an external atom a of type (G) where (i) a is in a positive literal and with $\emptyset \not\models a$; or (ii) a is in a negative literal and $\emptyset \models a$; and (2) omitting all other external atoms of type (G).*

Proof (sketch). A ground external atom a of type (G) has only constant input, therefore $I \models a$ is independent from the value of I. Hence, rules where (1) applies can never obtain a satisfied body due to a while rules where (2) applies can never obtain a non-satisfied body due to a.

External atoms of type (S) can be handled the same way as in DLVHEX [7,11], therefore we here do not provide formal results about them.[3]

External atoms outside the PHF fragment can be processed with Liberal Safety [9]: it permits automatic verification of finite instantiation of a HEX program in the presence of cyclic dependencies among external atoms that perform value invention, depend on the answer set candidate, and have certain (semantic) properties. Opposed to the methodology of Liberal Safety, HEXLITE delegates finiteness of instantiation to the programmer (as also done, e.g., by GRINGO).

3.2 Amenable Application Scenarios

External computations of type (S) can pass nogoods to the solver that describe how their truth value depends on the interpretation.

Constraint Answer Set Programming (CASP) [25] has been realized in HEX [27] using one external atom *&check* of type (S). Application scenarios in [27] use external atoms of type (G) for SQL querying: $\&sql[Query](AnswerTuple)$. High-level *planning for robotics* has been interleaved with low-level motion planning using HEX [18], where external atoms for motion planning are either of type (S) or of type (G) and there is no value invention. HEX-programs with *existential quantification* (HEX$^{\exists}$) [8] as well as HEX-programs with function symbols use only external atoms of type (G) to perform tasks related to Skolemization, similar to what is presented in Section 5.1. The MCS-IE system for *explaining inconsistency in Multi-Context Systems* [4] is implemented in HEX and uses only external atoms of type (S). Two further application scenarios are *abduction* and *RDF processing*, shown in detail in Sections 5.1 and 5.2.

[3] DLVHEX replaces them with an ordinary replacement atom, guesses truth of repalcement atoms with extra rules, and accepts only answer set candidates I where guessed truth values correspond with external computations wrt. I, see Section 4.

4 Hexlite Solver Design and Architecture

Principles. The design of HEXLITE followed several guiding principles.

- Delegation: delegate as much as possible to the backend solver (currently CLINGO).
- Separation: deal with external atoms either during grounding or during solving in order to avoid multiple grounding passes.
- Programmer Responsibility: delegate the responsibility for finite instantiation to the programmer (as in GRINGO).

Delegation reduces computational overhead and duplication of code that already exists in the ASP backend. As the main consequence of this principle, HEXLITE performs no safety check and no syntax check, not even thorough parsing of the input. Instead, a shallow representation of the input program is created. This representation is sufficient for rewriting rules and external atoms for subsequent evaluation. As a consequence of Delegation, unsafe variables are detected only within GRINGO because safety checking is not required for HEXLITE rewriting. A second consequence of Delegation is, that HEXLITE has no internal representation of the current answer set candidate; instead, the CLASP PYTHON API is used to directly access the (partial) model within CLASP. Moreover, optimization is handled transparently within CLASP.

Separation is the basis for defining the PHF and contributes to the small implementation of HEXLITE, because evaluation follows the common structure of ASP solvers with external atoms that are relevant either for grounding or for solving. (Currently, HEXLITE is implemented using 3,300 lines of PYTHON code, which includes the shallow parser, FLP checker, and code comments.) In particular, HEXLITE always uses only a single Evaluation Unit [7].

Programmer Responsibility is a principle from GRINGO: the burden of ensuring a finite instantiation is put on the programmer and not verified by preprocessing. This is the opposite of the philosophy followed in the DLVHEX solver where various safety notions such as Domain Expansion Safety and Liberal Domain Expansion Safety [10] are defined and also checked by the solver, depending on properties of external computations. The upside of not checking this in the solver is decreased preprocessing effort, the downside is that HEXLITE (just like GRINGO) will not complain about the program '$p(0) \leftarrow .\ p(s(X)) \leftarrow p(X)$.' but start to instantiate it and exhaust the available memory due to its infinite instantiation. In HEX programs, external computations can cause infinite instantiation, but the programmer of external computations might take specific measures to prevent infinite instantiation (as shown in Section 5.1). Therefore, delegating responsibility to the user instead of performing costly verifications can be an advantage.

Architecture. Figure 1 shows the architecture of HEXLITE. The input program P is analyzed with a *shallow parser*, followed by the rewriter module that rewrites only those parts of rules that contain external atoms. The *rewriter* accesses external computations for obtaining their type and creates the following GRINGO-compatible rules for each rule $r \in P$.

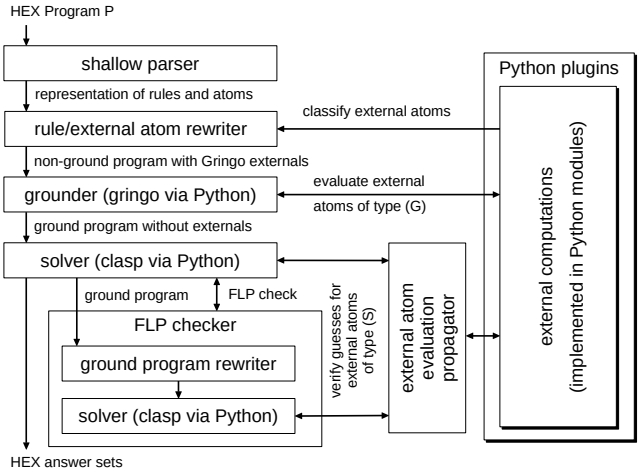

Fig. 1. Architecture of the HEXLITE system and its interaction with the CLINGO Python library and plugins for external computations.

(i) a rule r' where each external atom a of form $\&g[Y_1,\ldots,Y_n](X_1,\ldots,X_m)$
   (a) is replaced by a GRINGO external of form $@g(Y_1,...,Y_n)=(X_1,...,X_m)$ if a is of type (G);
   (b) is replaced by a *replacement atom* of form $e_{\&g}(Y_1,\ldots,Y_n)$ if a is of type (S); and[4]

(ii) for each replacement atom created in (b) above a rule r'' of form

$$e_{\&g}(Y_1,\ldots,Y_n) \vee \overline{e}_{\&g}(Y_1,\ldots,Y_n) \leftarrow B'$$

where $B' = \{\beta \in B(r) \mid \beta \text{ shares variables with } Y_1,\ldots,Y_n\}$.

(DLVHEX uses a similar rewriting, see [13, 17].) After rewriting, GRINGO is used for instantiating the rewritten program with a PYTHON context that evalutes external computations of form $@g(Y_1,\ldots,Y_n) = (X_1,\ldots,X_m)$ and replaces them by the corresponding truth values. The resulting ordinary ground program is passed to a CLASP instance that has been prepared with (i) a custom PYTHON propagator, (ii) a ground program observer that collects the ground program for usage in the FLP checker, and (iii) an `on_model` callback (not shown in the figure) which calls the FLP checker and, after a successful check, outputs the answer set without auxiliary elements.

The custom PYTHON propagator compares truth values of replacement atoms with the results of their corresponding external computations. For failed checks, nogoods are created to prevent future failed checks. Moreover, user-defined nogoods can be provided by plugins to further guide the search (see also the description of external atoms in Sections 5.1 and 5.2). The FLP checker is necessary

[4] In that case $m = 0$ so the replacement atom does not include $X_1,\ldots,X_m$.

for HEX programs with positive loops over external atoms and prevents answer sets with self-founded truth values, see [19].[5]

HEXLITE uses the same PYTHON API as DLVHEX, therefore migrating plugins from one solver to the other is easy. None of the benchmarks we used in evaluations has positive loops over external atoms of type (S), therefore we deactivated the FLP checker for all experiments (both for DLVHEX and for HEXLITE).

5 Experimental Evaluation

For experimental evaluation of HEXLITE, we use with two application domains: variants of cost-based abduction [28] and the RDF plugin [16].

We choose these domains because they both contain both types of external atoms, and because they were part of the inspiration for developing HEXLITE.

5.1 Cost-based Abduction Benchmark

Cost-based abduction consists of 50 instances over the ACCEL natural language story understanding benchmark [26]. The full benchmark is available online.[6,7] ACCEL uses two external predicates: *&invent* for flexible Skolemization, and *&transitive* as described in Example 2 for ensuring transitivity of a guessed relation over many elements.

External predicate *&invent* is used for ensuring finite instantiation in the presence of existential variables in rule heads.[8] Given a rule R with an existential variable B in the head and universal variables $A_1, \ldots, A_k$ in the body, instead of replacing B with Skolem term $s(A_1, \ldots, A_k)$ we add $\&invent[R, c_B, A_1, \ldots, A_k](B)$ to the rule body and define $F_{\&invent}$ such that a finite set of values for B is invented, independent from cycles in the program. For our evaluation, we used two variants of this benchmark:

SK/P[1] *variant.* Value invention is blocked if at least one parent term is invented: $F_{\&invent^{P^1}}(I, R, V, A_1, \ldots, A_k)$ returns the tuple $(\, s(R, V, A_1, \ldots, A_k)\,)$ if no A_i, $1 \leq i \leq k$ is of form $s(\cdots)$, otherwise it returns no tuple.

SK/G[1] *variant.* Value invention is blocked if at least one parent term was invented by the same ground external predicate: $F_{\&invent^{G^1}}(I, R, V, A_1, \ldots, A_k)$ returns the tuple $(\, s(R, V, A_1, \ldots, A_k)\,)$ if no A_i, $1 \leq i \leq k$ has a sub-term of form $s(R, V, \cdots)$, otherwise it returns no tuple.

[5] The FLP check implemented in HEXLITE is described in Proposition 1 in [7]. The FLP check can be deactivated if it is not required (this is another example of Programmer Responsibility). The custom PYTHON propagator is re-used in the FLP checker.

[6] `https://bitbucket.org/knowlp/asp-fo-abduction`

[7] To permit a fairer comparison, we used only objective functions CARD and COH (DLVHEX is incompatible with WA) and we removed all facts of the form `comment(...).` which served only an informational purpose (DLVHEX is significantly slower if these facts are included).

[8] This generalizes the termination mechanism for reasoning as it was implemented in the original ACCEL reasoner [26].

Intuitively, SK/P[1] invents only values that have non-invented *parents*, and SK/G[1] invents only a single *generation* of values in each rule. For details, examples, and proofs of the finite instantiation property, we refer to Sec. 4.1 in [28].

External predicate *&transitive* appears in a single constraint of form $\leftarrow$ *not* $\&transitive[eq]()$. Each time this external computation evaluates to false, it creates nogoods that provide the reason for intransitivity (i.e., a triple of literals $p(A,B)$, $p(B,C)$, $\neg p(A,C)$) to the solver to assist the search for a transitive relation. For more details see Section 4.2 in [28].

Importantly, $F_{\&invent}$ does not use I and therefore external atoms of form $\&invent[\cdots](\cdots)$ are of type (G). Moreover, external predicate *&transitive* produces no values in output tuples (i.e., $m = 0$) and it has type $t_1 = 2$ (it uses the extension of binary predicate p) therefore it is of type (S).

5.2 RDF Benchmark

The RDF plugin realizes the RDF triple external atom as described in Example 1. We here extend the original application [16] with a second external atom of type (S). We experiment with the COLINDA [29] knowledge graph which contains 150,000 triples about conferences.[9] We perform the following three reasoning problems.

Import. We simply import all triples using the program

$$explore(\text{"}\ldots/\text{colinda.rdf"}) \leftarrow .$$
$$triple\_at(S,P,O) \leftarrow \&rdf[What](S,P,O), explore(What).$$

which yields a single answer set with around 150,000 atoms.

Vegas. We are interested in names of all conferences in Las Vegas. The program

$$explore(\text{"}\ldots/\text{colinda.rdf"}) \leftarrow . \quad location(\text{"http://sws.geonames.org/3635260/"}) \leftarrow .$$
$$conference(Conf) \leftarrow explore(G), location(Loc),$$
$$\&rdf[G](Conf, \text{"http://swrc.ontoware.org/ontology\#location"}, Loc).$$
$$title(Title) \leftarrow explore(G), conference(Conf),$$
$$\&rdf[G](Conf, \text{"http://swrc.ontoware.org/ontology\#eventTitle"}, Title).$$

yields a single answer set with 330 atoms, 164 of them containing the titles conferences in COLINDA that took place in Las Vegas (encoded as ID 3635260).

Marathon. This is an optimization problem where we are interested in making a conference marathon for visiting the maximum possible number of cities in

[9] Dataset retrieved from `https://old.datahub.io/dataset/colinda` .

France over two weeks. This is encoded in the following HEX program.

$$
\begin{aligned}
&\mathit{explore}(\text{"}\ldots/\mathsf{colinda.rdf}\text{"}) \leftarrow . \quad \mathit{country}(\text{"}\mathsf{France}\text{"}) \leftarrow .\\
&\mathit{location}(\mathit{Loc}) \leftarrow \mathit{explore}(G), \mathit{country}(C),\\
&\qquad \&\mathit{rdf}[G](\mathit{Loc}, \text{"}\mathsf{http://www.geonames.org/ontology\#countryName}\text{"}, C).\\
&\mathit{conference}(\mathit{Conf}, \mathit{Loc}) \leftarrow \mathit{explore}(G), \mathit{location}(\mathit{Loc}),\\
&\qquad \&\mathit{rdf}[G](\mathit{Conf}, \text{"}\mathsf{http://swrc.ontoware.org/ontology\#location}\text{"}, \mathit{Loc}).\\
&\mathit{in}(\mathit{Conf}) \vee \mathit{out}(\mathit{Conf}) \leftarrow \mathit{conference}(\mathit{Conf}, \mathit{Loc}).\\
&\mathit{covered}(\mathit{Loc}) \leftarrow \mathit{conference}(\mathit{Conf}, \mathit{Loc}), \mathit{in}(\mathit{Conf}).\\
&\leftsquigarrow \mathit{location}(\mathit{Loc}), \mathit{not}\ \ \mathit{covered}(\mathit{Loc}). \quad [1, \mathit{Loc}]\\
&\mathit{date}(\mathit{Date}) \leftarrow \mathit{explore}(G), \mathit{in}(\mathit{Conf}),\\
&\qquad \&\mathit{rdf}[G](\mathit{Conf}, \text{"}\mathsf{http://swrc.ontoware.org/ontology\#startDate}\text{"}, \mathit{Date}).\\
&\leftarrow \mathit{not}\ \ \&\mathit{dates\_span\_days}[\mathit{date}, 14].
\end{aligned}
$$

This encoding represents locations in France in *location*, conferences and their locations (if in France) in *conference*, and performs a guess (*in*) selecting relevant conferences. Those conferences that are selected define which location is covered. We maximize coverage by incurring a cost of 1 for each location that is not covered by means of the weak constraint ($\leftsquigarrow$). Furthermore, we extract dates of covered conferences in *date* and use an external atom to indicate whether the dates lie within a 14 day period.

Function $F_{\&dates\_span\_days}(I, p, d)$ is defined to return an empty tuple iff all dates X with $p(X) \in I$ are within d days of one another. The external computation of *dates_span_days* guides the search by providing nogoods for all pairs of dates $p(X) \in I$ that are not within d days of each other.

Evaluating this program yields an answer set with 29 locations and 404 conferences in France, and an optimal selection of 6 conferences (in 6 distinct cities) which start between 12^{th} and 23^{rd} of March 2012.

5.3 Experimental Setup

We performed experiments on a computer with an Intel(R) i5-3450 CPU with 4 cores and 16 GB RAM running Linux. For the Abduction Benchmark we limited memory consumption to 5 GB and execution time to 300 s (5 min). For the RDF Benchmark we limited execution time to 1800 s (30 min). We never executed more than 2 runs in parallel and we used non-parallel computation mode for solver settings. To make the comparison fair, we used the same PYTHON plugin for DLVHEX and HEXLITE (the plugin API of HEXLITE is compatible with the one of DLVHEX, including the API for learning nogoods in external computations). We used the latest version of DLVHEX[10] and the latest version of HEXLITE.[11]

[10] We used git hash 5a1ee06d from `git@github.com:hexhex/core.git` because the stable version 2.5.0 performed significantly worse.

[11] Git hash d0e7896eb from `git@github.com:hexhex/hexlite.git`.

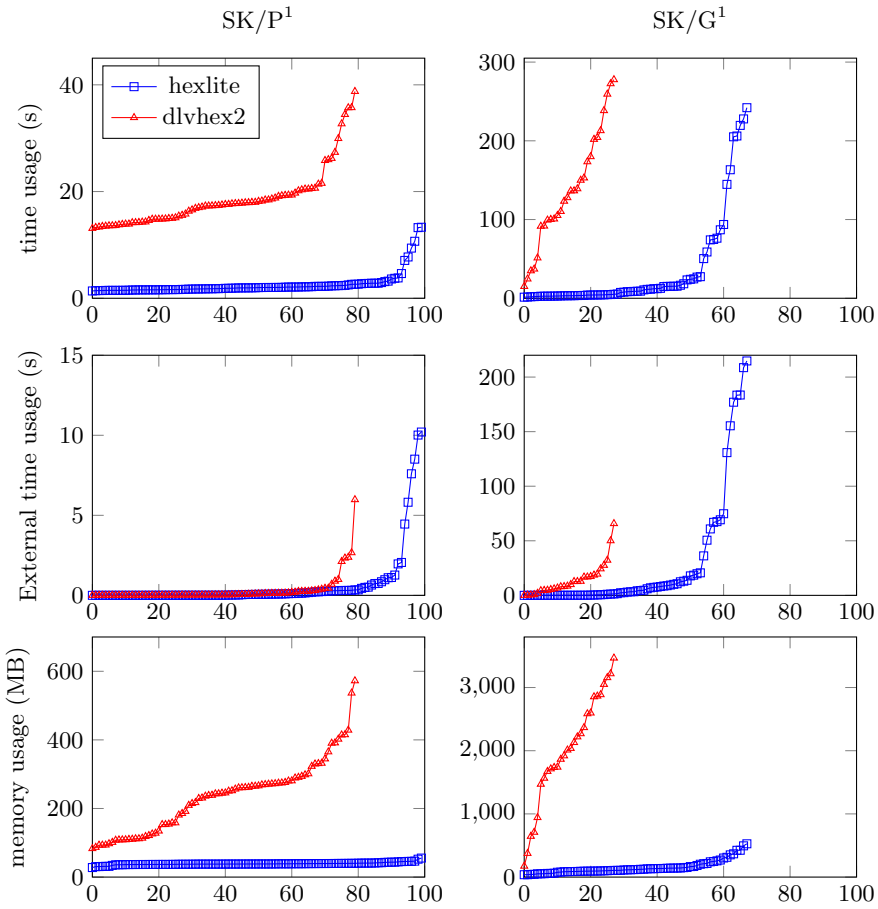

Fig. 2. Abduction benchmark: cactus plots for time, external computation time, and memory usage with two different value invention limits (SK/P^1 and SK/G^1).

5.4 Results

We will write TO (resp., MO) to indicate that the time (resp., memory) limit was exceeded.

Figure 2 shows cactus plots of the experiments on the abduction benchmark. For SK/P^1, HEXLITE successfully solves all instances within at most 14 s while DLVHEX fails to solve 20 instances (8 times TO and 12 times MO). For SK/G^1, HEXLITE fails to solve 32 instances because of TO and DLVHEX fails to solve 72 instances (46 times TO and 26 times MO). HEXLITE solves all instances that DLVHEX manages to solve within the timeout. The corresponding cactus plots of external computation times show, that HEXLITE and DLVHEX call the same plugin differently often to solve the same instances. The cactus plots of memory consumption show, that memory consumption of DLVHEX rises much steeper than

Problem	Engine	Time s	Space MB	Result	External Computations Calls #	Time s	Learned #
Import	DLVHEX	1,800	4,321	TO	1	19	0
	HEXLITE	31	254	OK	1	25	0
Vegas	DLVHEX	185	987	OK	2	38	0
	HEXLITE	20	191	OK	1	17	0
Marathon	DLVHEX	1,127	2,462	OK	2,543	58	29,860
	HEXLITE	28	199	OK	777	23	21,322

Table 1. RDF benchmark: results of evaluating DLVHEX and HEXLITE on the COLINDA conference knowledge graph.

the one of HEXLITE. Overall, HEXLITE shows a much better performance than DLVHEX on the abduction benchmark, and the difference between the solvers is more striking for SK/G[1] variant.

Table 1 shows results of running the RDF benchmark using DLVHEX and HEXLITE. Interestingly, the only timeout happens for Import where the whole set of RDF triples is represented in the answer set. DLVHEX cannot compute the result within 5 min. For Vegas, where many triples can be ignored, both DLVHEX and HEXLITE are more efficient than for Import, however DLVHEX requires 3 min while HEXLITE requires 20 s to perform this (deterministic) computation. The search/optimization problem Marathon shows a big gap between DLVHEX and HEXLITE, both in terms of time and space. In particular, HEXLITE performs fewer external atom calls and requires learning of fewer nogoods from the external computation in order to find an optimal answer set and prove its optimality. This can be explained by the lightweight usage of the CLINGO optimization feature in HEXLITE, while DLVHEX performs a lot of bookkeeping and additional computation to perform optimization.

6 Discussion and Conclusion

Our experiments show that HEXLITE has better performance than DLVHEX in two real-world applications of HEX which are both in the Pragmatic HEX Fragment. Clearly, not all HEX programs are in PHF and some problems will be difficult to convert into PHF. Nevertheless, wherever HEXLITE can be applied it can be a faster alternative to DLVHEX that is also easier to install as it is fully implemented in PYTHON.

There are several reasons for the better performance of HEXLITE compared with DLVHEX. Firstly, HEXLITE can remove many external atoms during grounding and they will not even be seen by the solver, while DLVHEX creates guesses for all external atoms, even those that do not depend on the interpretation, and needs to verify their truth during the solving. This eliminates a lot of poten-

tial for backend solver preprocessing in DLVHEX. Secondly, HEXLITE passes all optimization tasks to the CLINGO solver backend and just checks during propagation whether the current assignment is in itself consistent (all interpretations are checked against external atom semantics, non-partial interpretations are additionally checked against the FLP property). Opposed to that, DLVHEX performs a lot of internal bookkeeping related to optimization and maintains its own representation of the interpretation and its own cost representation for answer sets, which causes significant memory and computation overhead.

Performance issues of DLVHEX were the original reason that the tool for cost-based abduction described in [28] is formalized using the HEX formalism but implemented using a custom reasoner based on CLINGO. With HEXLITE, this implementation could be realized with much less effort and without a dedicated algorithm just by using HEXLITE and implementations of two external computations.

HEXLITE uses many aspects of the CLINGO API, however it does not use CLINGO Externals [21] which are truth values given 'from the outside' to the solving process. Instead, the HEX notion of external computation is in a tight interaction with the answer set candidate and the results of other external computations, and during HEX evaluation, truth values of externals might be reconsidered before finding an answer set. What we do use in the FLP checker are CLASP 'assumptions' to communicate the answer set candidate to the FLP checker (which uses a single rewriting of the ground program for checking the FLP property of all potential answer sets).

Related to this work is a study on lazy instantiation of constraints and an alternative usage of propagators instead of constraints [6] using the WASP solver [1, 2]. In the future, WASP could be integrated into HEXLITE as an alternative to the CLINGO backend, the PYTHON API of WASP could be used to deal with external atoms of type (S). The new DLV grounder and its externals [5] could be integrated to handle external atoms of type (G).

We are glad that a study on Inductive Logic Programming [23] was successfully realized using HEXLITE. A (similarly pragmatic) fragment of the ACTHEX extension of HEX [3, 20] based on HEXLITE is available in the HEXLITE Git repository.

Acknowledgements

We are grateful to Stefano Germano, Tobias Kaminski, Christoph Redl, Antonius Weinzierl, and the anonymous reviewers for feedback about the HEXLITE system and this manuscript. This work has been partially supported by the research project DynaCon, funded by the Austrian Federal Ministry of Transport, Innovation and Technology (BMVIT) under grant agreement 861263.

References

1. Alviano, M., Dodaro, C., Faber, W., Leone, N., Ricca, F.: WASP: A native ASP solver based on constraint learning. In: International Conference on Logic Pro-

gramming and Non-monotonic Reasoning (LPNMR). pp. 54–66 (2013)
2. Alviano, M., Dodaro, C., Leone, N., Ricca, F.: Advances in WASP. In: International Conference on Logic Programming and Non-monotonic Reasoning (LPNMR). pp. 40–54 (2015)
3. Basol, S., Erdem, O., Fink, M., Ianni, G.: Hex programs with action atoms. In: LIPIcs-Leibniz International Proceedings in Informatics. vol. 7. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2010)
4. Bögl, M., Eiter, T., Fink, M., Schüller, P.: The MCS-IE system for explaining inconsistency in multi-context systems. In: European Conference on Logics in Artificial Intelligence (JELIA). pp. 356–359 (2010)
5. Calimeri, F., Fusca, D., Perri, S., Zangari, J.: External Computations and Interoperability in the New DLV Grounder. In: AI*IA. pp. 172–185 (2017)
6. Cuteri, B., Dodaro, C., Ricca, F., Schüller, P.: Constraints, lazy constraints, or propagators in ASP solving: An empirical analysis. Theory and Practice of Logic Programming **17**(5-6) (2017)
7. Eiter, T., Fink, M., Ianni, G., Krennwallner, T., Redl, C., Schüller, P.: A model building framework for answer set programming with external computations. Theory and Practice of Logic Programming **16**(4) (2016)
8. Eiter, T., Fink, M., Krennwallner, T., Redl, C.: HEX-programs with existential quantification. In: Workshop on Logic Programming. pp. 99–117 (2013)
9. Eiter, T., Fink, M., Krennwallner, T., Redl, C.: Liberal Safety for Answer Set Programs with External Sources. In: AAAI Conference on Artificial Intelligence. pp. 267–275 (2013)
10. Eiter, T., Fink, M., Krennwallner, T., Redl, C.: Domain Expansion for ASP-Programs with External Sources. Artificial Intelligence **233**, 84–121 (2016)
11. Eiter, T., Germano, S., Ianni, G., Kaminski, T., Redl, C., Schüller, P., Weinzierl, A.: The DLVHEX System. KI - Künstliche Intelligenz **32**(2), 187–189 (2018)
12. Eiter, T., Ianni, G., Lukasiewicz, T., Schindlauer, R., Tompits, H.: Combining answer set programming with description logics for the semantic web. Artificial Intelligence **172**(12-13), 1495–1539 (2008)
13. Eiter, T., Ianni, G., Schindlauer, R., Tompits, H.: A Uniform Integration of Higher-Order Reasoning and External Evaluations in Answer-Set Programming. In: International Joint Conference on Artificial Intelligence (IJCAI). pp. 90–96 (2005)
14. Eiter, T., Ianni, G., Schindlauer, R., Tompits, H.: Effective Integration of Declarative Rules with External Evaluations for Semantic-Web Reasoning. In: European Semantic Web Conference (ESWC). pp. 273–287 (2006)
15. Eiter, T., Kaminski, T.: Exploiting contextual knowledge for hybrid classification of visual objects. In: Proc. Logics in Artificial Intelligence (JELIA). pp. 223–239 (2016)
16. Eiter, T., Kaminski, T., Redl, C., Schüller, P., Weinzierl, A.: Answer Set Programming with External Source Access. In: Reasoning Web International Summer School, vol. 10370 LNCS, pp. 204–275. Springer (2017)
17. Eiter, T., Kaminski, T., Redl, C., Weinzierl, A.: Exploiting partial assignments for efficient evaluation of answer set programs with external source access. Journal of Artificial Intelligence Research **62**, 665–727 (2018)
18. Erdem, E., Patoglu, V., Schüller, P.: A Systematic Analysis of Levels of Integration between Low-Level Reasoning and Task Planning. AI Communications **29**(2), 319–349 (2016)
19. Faber, W., Leone, N., Pfeifer, G.: Recursive aggregates in disjunctive logic programs: Semantics and complexity. In: Proc. Logics in Artificial Intelligence (JELIA). pp. 200–212 (2004)

20. Fink, M., Germano, S., Ianni, G., Redl, C., Schüller, P.: ActHEX: Implementing HEX programs with action atoms. In: International Conference on Logic Programming and Non-monotonic Reasoning (LPNMR). pp. 317–322 (2013)
21. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: Multi-shot asp solving with clingo. Theory and Practice of Logic Programming pp. 1–56 (2018)
22. Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Deductive Databases. New Generation Computing **9**, 365–385 (1991)
23. Kaminski, T., Eiter, T., Inoue, K.: Exploiting Answer Set Programming with External Sources for Meta-Interpretive Learning. Theory and Practice of Logic Programming **18**(3-4), 571–588 (2018)
24. Lassila, O., Swick, R.: Resource description framework (RDF) model and syntax specification (1999), `www.w3.org/TR/1999/REC-rdf-syntax-19990222`
25. Lierler, Y.: Relating constraint answer set programming languages and algorithms. Artif. Intell. **207**, 1–22 (2014)
26. Ng, H.T., Mooney, R.J.: Abductive Plan Recognition and Diagnosis: A Comprehensive Empirical Evaluation. In: Knowledge Representation and Reasoning (KR). pp. 499–508 (1992)
27. Rosis, A.D., Eiter, T., Redl, C., Ricca, F.: Constraint answer set programming based on HEX-programs. In: Proc. Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP) (2015)
28. Schüller, P.: Modeling variations of first-order horn abduction in answer set programming. Fundamenta Informaticae **149**(1-2) (2016)
29. Softic, S.: Conference linked data (colinda), version 1.0, last updated July 30, 2016. 149020 triples, `http://www.colinda.org`.