Towards Efficient Evaluation of HEX-Programs*

Thomas Eiter, Giovambattista Ianni, Roman Schindlauer, and Hans Tompits
Institut für Informationssysteme, Technische Universität Wien,
Favoritenstraße 9–11, A-1040 Vienna, Austria
{eiter, ianni, roman, tompits}@kr.tuwien.ac.at

Abstract

We briefly report on the development status of dlvhex, a reasoning engine for HEX-programs, which are nonmonotonic logic programs with higher-order atoms and external atoms. Higher-order features are widely acknowledged as useful for various tasks and are essential in the context of meta-reasoning. Furthermore, the possibility to exchange knowledge with external sources in a fully declarative framework such as answer-set programming (ASP) is particularly important in view of applications in the Semantic-Web area. Through external atoms, HEX-programs can deal with external knowledge and reasoners of various nature, such as RDF datasets or description logics bases.

Introduction

Nonmonotonic semantics is often requested by Semantic-Web designers in cases where the reasoning capabilities of the *Ontology layer* of the Semantic Web turn out to be too limiting, since they are based on monotonic logics. The widely acknowledged answer-set semantics of nonmonotonic logic programs (Gelfond & Lifschitz 1991), which is arguably the most important instance of the *answer-set programming* (ASP) paradigm, is a natural host for giving nonmonotonic semantics to the *Rules* and *Logic* layers of the Semantic Web.

In order to address problems such as *meta-reasoning* in the context of the Semantic Web and interoperability with other software, in (Eiter *et al.* 2005), we have extended the answer-set semantics to HEX-*programs*, which are *higher-order logic programs* (which accommodate meta-reasoning through *higher-order atoms*) with *external atoms* for software interoperability. Intuitively, a higher-order atom allows to quantify values over predicate names, and to freely exchange predicate symbols with constant symbols, like in the rule

$$C(X) \leftarrow subClassOf(D, C), D(X).$$

An external atom facilitates the assignment of a truth value of an atom through an external source of computation. For instance, the rule

$$t(Sub, Pred, Obj) \leftarrow \&RDF[uri](Sub, Pred, Obj)$$

computes the predicate t taking values from the predicate $\&RDF$. The latter extracts RDF statements from the set of URIs specified by the extension of the predicate uri; this task is delegated to an external computational source (e.g., an external deduction system, an execution library, etc.). External atoms allow for a bidirectional flow of information to and from external sources of computation such as description logics reasoners. By means of HEX-programs, powerful meta-reasoning becomes available in a decidable setting, e.g., not only for Semantic-Web applications, but also for meta-interpretation techniques in ASP itself, or for defining policy languages.

Other logic-based formalisms, like TRIPLE (Sintek & Decker 2002) or F-Logic (Kifer, Lausen, & Wu 1995), feature also higher-order predicates for meta-reasoning in Semantic-Web applications. Our formalism is fully declarative and offers the possibility of nondeterministic predicate definitions with higher complexity in a decidable setting. This proved already useful for a range of applications with inherent nondeterminism, such as ontology merging (Wang *et al.* 2005) or matchmaking, and thus provides a rich basis for integrating these areas with meta-reasoning.

HEX-Programs

Syntax

HEX programs are built on mutually disjoint sets $\mathcal{C}$, $\mathcal{X}$, and $\mathcal{G}$ of *constant names*, *variable names*, and *external predicate names*, respectively. Unless stated otherwise, elements from $\mathcal{X}$ (resp., $\mathcal{C}$) are written with first letter in upper case (resp., lower case), and elements from $\mathcal{G}$ are prefixed with "&". Constant names serve both as individual and predicate names. Importantly, $\mathcal{C}$ may be infinite.

Elements from $\mathcal{C} \cup \mathcal{X}$ are called *terms*. A *higher-order atom* (or *atom*) is a tuple $(Y_0, Y_1, \ldots, Y_n)$, where $Y_0, \ldots, Y_n$ are terms and $n \geq 0$ is its *arity*. Intuitively, Y_0 is the predicate name; we thus also use the familiar notation $Y_0(Y_1, \ldots, Y_n)$. The atom is *ordinary*, if Y_0 is a constant. For example, $(x, rdf{:}type, c)$ and $node(X)$ are ordinary atoms, while $D(a, b)$ is a higher-order atom. An *external atom* is of the form

$$\&g[Y_1, \ldots, Y_n](X_1, \ldots, X_m), \tag{1}$$

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

*This work was partially supported by the Austrian Science Fund (FWF) under grant P17212-N04, and by the European Commission through the IST Networks of Excellence REWERSE (IST-2003-506779).

It is possible to specify *molecules* of atoms in F-Logic-like syntax. For instance, $gi[father \rightarrow X, Z \rightarrow iu]$ is a shortcut for the conjunction $father(gi, X), Z(gi, iu)$.

HEX-programs are sets of rules of the form

$$\alpha_1 \vee \cdots \vee \alpha_k \leftarrow \beta_1, \ldots, \beta_n, not\, \beta_{n+1}, \ldots, not\, \beta_m, \quad (2)$$

where $m, k \geq 0$, $\alpha_1, \ldots, \alpha_k$ are higher-order atoms, and $\beta_1, \ldots, \beta_m$ are either higher-order atoms or external atoms. The operator "*not*" is *negation as failure* (or *default negation*).

Semantics

The semantics of HEX-programs is given by generalizing the answer-set semantics (Eiter *et al.* 2005). The *Herbrand base* of a program P, denoted HB_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 of a rule r, $grnd(r)$, is defined accordingly, and the grounding of program P by $grnd(P) = \bigcup_{r \in P} grnd(r)$. An *interpretation relative to* P is any subset $I \subseteq HB_P$ containing only atoms.

We say that an interpretation $I \subseteq HB_P$ is a *model* of an atom $a \in HB_P$ iff $a \in I$. Furthermore, I is a model of a ground external atom $a = \&g[y_1, \ldots, y_n](x_1, \ldots, x_m)$ iff $f_{\&g}(I, y_1, \ldots, y_n, x_1, \ldots, x_m) = 1$, where $f_{\&g}$ is an $(n+m+1)$-ary Boolean function associated with $\&g$, called *oracle function*, assigning each element of $2^{HB_P} \times \mathcal{C}^{n+m}$ either 0 or 1.

Let r be a ground rule. We define (i) $I \models H(r)$ iff there is some $a \in H(r)$ such that $I \models a$, (ii) $I \models B(r)$ iff $I \models a$ for all $a \in B^+(r)$ and $I \not\models a$ for all $a \in B^-(r)$, and (iii) $I \models r$ iff $I \models H(r)$ whenever $I \models B(r)$. We say that I is a *model* of a HEX-program P, denoted $I \models P$, iff $I \models r$ for all $r \in grnd(P)$.

The *FLP-reduct* of P w.r.t. $I \subseteq HB_P$, 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 iff I is a minimal model of fP^I. By $ans(P)$ we denote the set of answer sets of P.

Note that the answer-set semantics may yield no, one, or multiple models (i.e., answer sets) in general. Therefore, for query answering, *brave* and *cautious reasoning* (truth in some resp. all models) is considered in practice, depending on the application.

We have seen that the truth value of an external atom is determined with respect to a specific interpretation, via the domain of the associated Boolean function. As a consequence, the satisfiability of an external atom in general cannot be stated apriori, but only regarding an entire model of a program. This implies not only that external atoms influence the truth values of ordinary atoms by occuring in rule bodies, but also that ordinary atoms can have an effect on the evaluation of external atoms. Hence, HEX-programs facilitate a bidirectional flow of knowledge between the answer set program and the external evaluation function.

In practice, it is useful to differentiate between two kinds of input attributes for external atoms. For an external predicate $\&g$ (exploited, say, in an atom $\&g[p](X)$), a term appearing in an attribute position of type *predicate* (in this case, p) means that the outcomes of $f_{\&g}$ are dependent from the current interpretation I, for what the extension of the predicate named p in I is concerned. An input attribute of type *constant* does not imply a dependency of $f_{\&g}$ from some portion of I. An external predicate whose input attributes are all of type constant does not depend from the current interpretation.

Example 1 The external predicate $\&RDF$ introduced before is implemented with a single input argument of type *predicate*, because its associated function finds the RDF-URIs in the extension of the predicate uri:

$$tr(S, P, O) \leftarrow \&RDF[uri](S, P, O),$$
$$uri(\text{"}\textit{file://foaf.rdf}\text{"}) \leftarrow .$$

Should the input argument be of type constant, an equivalent program would be:

$$tr(S, P, O) \leftarrow \&RDF[\text{"}\textit{file://foaf.rdf}\text{"}](S, P, O).$$

or

$$tr(S, P, O) \leftarrow \&RDF[X](S, P, O), uri(X),$$
$$uri(\text{"}\textit{file://foaf.rdf}\text{"}) \leftarrow .$$

Usability of HEX-Programs

An interesting application scenario, where several features of HEX-programs come into play, is *ontology alignment*. Merging knowledge from different sources in the context of the Semantic Web is a crucial task (Calvanese, Giacomo, & Lenzerini 2001) that can be supported by HEX-programs in various ways:

Importing external theories. This can be achieved by fragments of code such as:

$$triple(X, Y, Z) \leftarrow \&RDF[uri](X, Y, Z),$$
$$triple(X, Y, Z) \leftarrow \&RDF[uri2](X, Y, Z),$$
$$proposition(P) \leftarrow triple(P, rdf{:}type, rdf{:}statement).$$

Searching in the space of assertions. In order to choose nondeterministically which propositions have to be included in the merged theory and which not, statements like the following can be used:

$$pick(P) \vee drop(P) \leftarrow proposition(P).$$

Translating and manipulating reified assertions. For instance, it is possible to choose how to put RDF triples (possibly including OWL assertions) in an easier manipulable and readable format, and to make selected propositions true such as in the following way:

$$\begin{aligned}(X, Y, Z) \leftarrow\ & pick(P), triple(P, rdf{:}subject, X),\\ & triple(P, rdf{:}predicate, Y),\\ & triple(P, rdf{:}object, Z),\\ C(X) \leftarrow\ & (X, rdf{:}type, C).\end{aligned}$$

Defining ontology semantics. The semantics of the ontology language at hand can be defined in terms of entailment rules and constraints expressed in the language itself or in terms of external knowledge, like in

$$D(X) \leftarrow subClassOf(D, C), C(X),$$
$$\leftarrow \&inconsistent[pick],$$

where the external predicate $\&inconsistent$ takes a set of assertions as input and establishes through an external reasoner whether the underlying theory is inconsistent.

Performing default and closed-world reasoning.
Assuming that a generic external atom $\&DL[C](X)$ is available for querying the concept C in a given description logics base, the *closed-world assumption* (CWA) can be stated as follows:

$$C'(X) \leftarrow not\,\&DL[C](X), concept(C), cwa(C, C'),$$

where $concept(C)$ is a predicate which holds for all concepts and $cwa(C, C')$ states that C' is the CWA of C.
Inconsistency of the CWA can be checked by pushing back inferred values to the external knowledge base:

$$set\_false(C, X) \leftarrow cwa(C, C'), C'(X),$$
$$inconsistent \leftarrow \&DL1[set\_false](b),$$

where $\&DL1[N](X)$ effects a check whether a knowledge base, augmented with all negated facts $\neg c(a)$ such that $N(c, a)$ holds, entails the empty concept $\bot$ (entailment of $\bot(b)$, for any constant b, is tantamount to inconsistency).

Implementation

The challenge of implementing a reasoner for HEX-programs lies in the interaction between external atoms and the ordinary part of a program. Due to the bidirectional flow of information represented by its input list, an external atom cannot be evaluated prior to the rest of the program. However, the existence of established and efficient reasoners for answer-set programs led us to the idea of splitting and rewriting the program such that an existing answer-set solver can be employed alternatingly with the external atoms' evaluation functions. In the following, we will outline methods that are already implemented in our prototype HEX reasoner dlvhex. We will partly refer to (Eiter *et al.* 2006), modifying the algorithms and concepts presented there where it is appropriate in the view of an actual implementation.

Dependency Information

Taking the dependency between heads and bodies into account is a common tool for devising an operational semantics for ordinary logic programs, e.g., by means of the notions of *stratification* or *local stratification* (Przymusinski 1988), or through *modular stratification* (Ross 1994) or *splitting sets* (Lifschitz & Turner 1994). In (Eiter *et al.* 2006), we defined novel types of dependencies, considering that in HEX programs, dependency between heads and bodies is not the only possible source of interaction between predicates. Contrary to the traditional notion of dependency based on propositional programs, we consider relationships between nonground, higher-order atoms. In the view of an actual implementation of a dependency graph processing algorithm, we will present in the following a generalized definition of atom dependency of (Eiter *et al.* 2006).

Definition 1 *Let P be a program and a, b atoms occurring in some rule of P. Then, a* depends positively *on b ($a \rightarrow_p b$), if one of the following conditions holds:*

1. *There is some rule $r \in P$ such that $a \in H(r)$ and $b \in B^+(r)$.*
2. *There are some rules $r_1, r_2 \in P$ such that $a \in B(r_2)$ and $b \in H(r_1)$ and there exists a partial substitution θ of variables in a such that either $a\theta = b$ or $a = b\theta$. E.g., $H(a, Y)$ unifies with $p(a, X)$.*
3. *There is some rule $r \in P$ such that $a, b \in H(r)$. Note that this relation is symmetric.*
4. *a is an external predicate of form $\&g[\bar{X}](\bar{Y})$ where $\bar{X} = X_1, \ldots, X_n$, and b is of form $p(\bar{Z})$, and, for some i, $X_i = p$ and of type predicate (e.g., $\&count[item](N)$ is externally dependent on $item(X)$).*

Moreover, a depends negatively *on b ($a \rightarrow_n b$), if there is some rule $r \in P$ such that $a \in H(r)$ and $b \in B^-(r)$. We say that a depends on b, if $a \rightarrow b$, where $\rightarrow = \rightarrow_p \cup \rightarrow_n$. The relation $\rightarrow^+$ denotes the transitive closure of $\rightarrow$.*

These dependency relations let us construct a graph, which we call *dependency graph* of the corresponding program.

Example 2 Consider the program of Figure 2, modeling the search for personal contacts that stem from a *FOAF-ontology*,[1] which is accessible by a URL.
The first two facts specify the URLs of the FOAF ontologies we want to query. Rules 3 and 4 ensure that each answer set will be based on a single URL only. Rule 5 extracts all triples from an RDF file specified by the extension of $input$. Rule 6 converts triples that assign names to individuals into the predicate $name$. Finally, the last rule traverses the RDF graph to construct the relation $knows$.
Figure 2 shows the dependency graph of P.[2]

Evaluation Strategy

The principle of evaluation of a HEX-program relies on the theory of *splitting sets*. Intuitively, given a program P, a splitting set S is a set of ground atoms that induce a sub-program $grnd(P') \subset grnd(P)$ whose models $\mathcal{M} = \{M_1, \ldots, M_n\}$ can be evaluated separately. Then, an adequate *splitting theorem* shows how to plug in $\mathcal{M}$ in a modified version of $P \setminus P'$ so that the overall models can be computed. Here, we use a modified notion of splitting set, accomodating non-ground programs and suited to our definition of dependency graph.

Definition 2 *A* global splitting set *for a* HEX*-program P is a set of atoms A appearing in P, such that whenever $a \in A$ and $a \rightarrow b$ for some atom b appearing in P, then also $b \in A$.*

In (Eiter *et al.* 2006), we already defined an algorithm based on splitting sets. However, there we used a general approach, decomposing P into strongly connected components (SCC in the following), which leads to a potentially large number of splitting sets (considering that a single atom

[1] "FOAF" stands for "Friend Of A Friend", and is an RDF vocabulary to describe people and their relationships.

[2] Long constant names have been abbreviated for the sake of compactness.

(1) $url(\text{“}http://www.kr.tuwien.ac.at/staff/roman/foaf.rdf\text{”}) \leftarrow;$
(2) $url(\text{“}http://www.mat.unical.it/\tilde{}ianni/foaf.rdf\text{”}) \leftarrow;$
(3) $\neg input(X) \vee \neg input(Y) \leftarrow url(X), url(Y), X \neq Y;$
(4) $input(X) \leftarrow not\neg input(X), url(X);$
(5) $triple(X, Y, Z) \leftarrow \&RDF[A](X, Y, Z), input(A);$
(6) $name(X, Y) \leftarrow triple(X, \text{“}http://xmlns.com/foaf/0.1/name\text{”}, Y);$
(7) $knows(X, Y) \leftarrow name(A, X), name(B, Y), triple(A, \text{“}http://xmlns.com/foaf/0.1/knows\text{”}, B).$

Figure 1: Example program using the RDF-atom. TODO: how to center the tabbing inside the figure?

that does not occur in any cycle is a SCC by itself). However, since the evaluation of each splitting set requires an interaction with dlv (i.e., , one or more calls to dlv, depending on the nature of the program associated with the splitting set), in a practical setting the object must be to identify as few (and hence as large) splitting sets as possible in order to minimize the number of actual reasoning steps and increase overall efficiency. Therefore, we now modify and specialize the notions and methods given there.

Definition 3 *A* local splitting set *for a* HEX*-program P is a set of atoms A appearing in P, such that for each atom $a \in A$ there is no atom $b \notin A$ such that $a \rightarrow b$ and $b \rightarrow^+ a$.*

Thus, contrary to a global splitting set, a local splitting set does not necessarily include the lowest layer of the program, but it never “breaks” a cycle.

Definition 4 *The* bottom *of P w.r.t. set of atoms A is the set of rules $b_A(P) = \{r \in P \mid H(r) \cap A \neq \emptyset\}$.*

We define the concept of *external component*, which represents a part of the dependency graph including at least one external atom. Intuitively, an external component is the smallest possible local splitting set that contains one or more external atoms. We distinguish between different types of external components, each with a specific procedure of evaluation, i.e., computing its model(s) w.r.t. to a set of ground atoms I. Before these are laid out, we need to introduce some auxiliary notions.

From the viewpoint of program evaluation, it turns out to be impractical to define the semantics of an external predicate by means of a Boolean function. Again restricting the concepts presented in (Eiter *et al.* 2006) for our practical needs, we define $F_{\&g} : 2^{HB_P} \times D_1, \ldots, D_n \rightarrow 2^{R^m_{\mathcal{C}}}$ with $F_{\&g}(I, y_1, \ldots, y_n) = \langle x_1, \ldots, x_m \rangle$ iff $f_{\&g}(I, y_1, \ldots, y_n, x_1, \ldots, x_m) = 1$, where $R^m_{\mathcal{C}}$ is the set of all tuples of arity m that can be built with symbols from $\mathcal{C}$. If the input list $y_1, \ldots, y_n$ is not ground in the original program, safety restrictions for HEX-programs ensure that its values can be determined from the remaining rule body.

A ground external atom $\&g$ is monotonic providing $I \models a$ implies $I' \models \&g$, for $I \subseteq I' \subseteq HB_P$.

With P_{hex}, we denote the ordinary logic program having each external atom $\&g[\overline{y}](\overline{x})$ in P replaced by $d_{\&g}(\overline{x})$ (we call this kind of atoms *replacement atoms*), where $d_{\&g}$ is a fresh predicate symbol. The input list $\overline{y}$ does not appear in the replacement atom, but will be considered when creating a ground fact $d_{\&g}(\overline{c})$ w.r.t. a specific interpretation (see below).

The categories of external component we consider are:

- A single external atom $\&g$ that does not occur in any cycle. Its evaluation method returns for each tuple $\langle x_1, \ldots, x_m \rangle$ in $F_{\&g}(I, y_1, \ldots, y_n)$ a ground replacement atom $d_{\&g}(x_1, \ldots, x_m)$ as result. The external atom in Figure 1, surrounded by a rectangular box, represents such a component.
- A strongly connected component C without negative dependency and only monotonic external atoms. A simple method for computing the (unique) model of such a component is given by the fixpoint operation of the operator $\Lambda : 2^{HB_P} \rightarrow 2^{HB_P}$, defined by $\Lambda(I) = M(P_{hex} \cup D_P(I)) \cap HB_P$, where:
  - P_{hex} is an ordinary logic program as defined above, with $P = b_C$.
  - $D_P(I)$ is the set of all facts $d_{\&g}(\overline{c}) \leftarrow$ such that $I \models \&g[\overline{y}](\overline{c})$ for all external atoms $\&g$ in P; and
  - $M(P_{hex} \cup D_P(I))$ is the single answer set of $P_{hex} \cup D_P(I)$; since P_{hex} is stratified, this answer set is guaranteed to exist and to be unique.
- A strongly connected component C with negative dependencies or nonmonotonic external atoms. In this case, we cannot rely on an iterative approach, but are forced to guess the value of each external atom beforehand and validate each guess w.r.t. the remaining atoms:
  - Construct P_{hex} from $P = b_C$ as before and add for each replacement atom $d_{\&g}(\overline{x})$ all rules
  $$d_{\&g}(\overline{c}) \vee \neg d_{\&g}(\overline{c}) \leftarrow \qquad (3)$$
  such that $\&g[\overline{y}](\overline{c})$ is a ground instance of $\&g[\overline{y}](\overline{x})$. Intuitively, the rules (3) “guess” the truth values of the external atoms of C. Denote the resulting program by P_{guess}.
  - Compute the answer sets $Ans = \{M_1, \ldots, M_n\}$ of P_{guess}.
  - For each answer set $M \in Ans$ of P_{guess}, test whether the original “guess” of the value of $d_{\&g}(\overline{c})$ is compliant with $f_{\&g}$. That is, for each external atom a, check whether $M \models \&g[\overline{y}](\overline{c})$. If this condition does not hold, remove M from Ans.
  - Each remaining $M \in Ans$ is an answer set of P iff M is a minimal model of fP^M_{hex}.

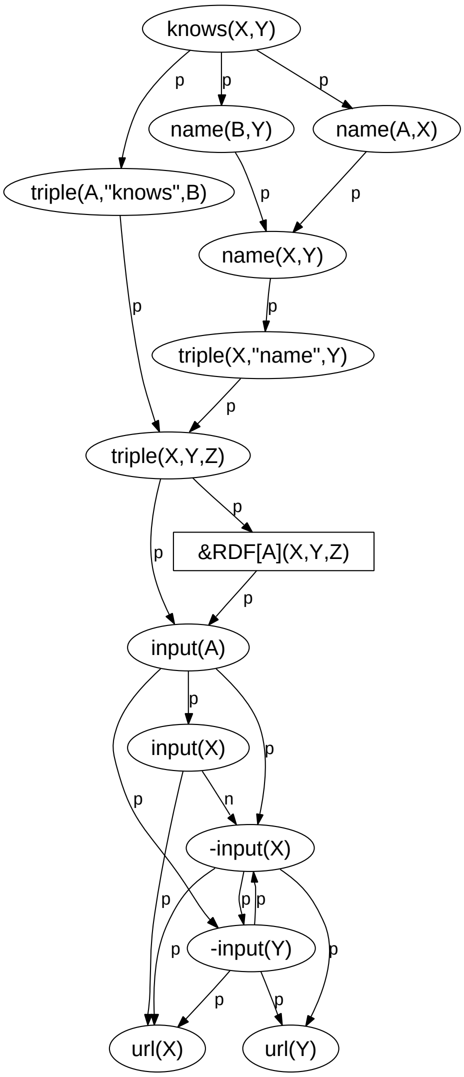

Figure 2: FOAF program graph.

Note that a cyclic subprogram must preserve certain safety rules in order to bound the number of symbols to be taken into account to a finite extent. To this end, we defined in (Eiter *et al.* 2006) the notion of *expansion-safety*, which avoids a potentially infinite ground program while still allowing external atoms to bring in additional symbols to the program.

The evaluation algorithm in (Figure 3) uses the following subroutines:

$eval(comp, \mathcal{I})$ Computes the models of an external component $comp$ (which is of one of the types described above) for each interpretation $I \in \mathcal{I}$; each I is added as a set of facts to the respective models.

$solve(P, I)$ Creates for the interpretation I a program $P' = P \cup A$ with A being the set of facts that corresponds to I and computes its answer sets.

EVALUATION ALGORITHM
(Input: a HEX-program P; Output: a set of models $\mathcal{M}$)

1. Determine the dependency graph G for P.
2. Find all external components C_i of P and build $Comp = \{C_1, \ldots, C_n\}$.
3. Set $T := Comp$ and $\mathcal{M} := \{F\}$, where F is the set of all facts originally contained in P. The set $\mathcal{M}$ will eventually contain $ans(P)$ (which is empty, in case inconsistency is detected).
4. While $P \neq \emptyset$ do
   (a) Let $\overline{T} = \{C \mid C \in T, \forall a \in C : \text{if } \exists a \rightarrow b \text{ then } b \in C\}$.
   (b) Let $\mathcal{M}' = \{\}$; for each C from $\overline{T}$:
   - let $\mathcal{M}' := \{\mathcal{A} \bigcup \mathcal{B} \mid (\mathcal{A}, \mathcal{B}) \in \mathcal{M}' \times eval(C, \mathcal{M})\}$,
   - remove C from $Comp$ and
   - let $P = P \setminus b_c(P)$.

   Let $\mathcal{M} = \mathcal{M}'$.
   (c) if $\mathcal{M} = \emptyset$ then halt.
   (d) Let $\mathcal{M} := \bigcup_{M \in \mathcal{M}} solve(P', M)$, where $P' = P_{hex} \setminus b_{\overline{C}}$ where $\overline{C}$ is the set of all nodes u such that either $u \rightarrow^{+} c$ with $c \in C$ or $u \in C$ for any $C \in Comp$; let $P = P \setminus P'$ and remove all atoms from the graph that are not in $\overline{C}$.

Figure 3: Evaluation algorithm.

Intuitively, the algorithm traverses the dependency graph from bottom to top, gradually pruning it while computing the respective models. Step (a) singles out all external components that do not depend on any further atom or component, i.e., that are on the "bottom" of the dependency graph. Those components are evaluated against the current known models in Step (b) and can be removed from the list of external components that are left to be solved. Moreover, Step (b) ensures that all rules of these components are removed from the program. From the remaining part, Step(d) extracts the largest possible subprogram that does not depend on any remaining external component, computes its models and removes it from the program resp. dependency graph.

Let us exemplarily step through the algorithm with Example 2 as input program P. First, the graph G is constructed as shown in Figure 2. Since P contains only a single external atom, the set $Comp$ constructed in Step 2 contains just one external component C, the $\&RDF$-atom itself. Step (a) extracts those components of $Comp$ that form a global splitting set, i.e., that do not depend on any atom not in the component. Clearly, this is not the case for C and hence, $\overline{T}$ is empty. Step (d) constructs an auxiliary program P' by removing the bottom of $\overline{C}$, which contains each component that is still in $Comp$ and every atom "above" it in the dependency graph:

$$\neg input(X) \vee \neg input(Y) \leftarrow url(X), url(Y), X \neq Y;$$
$$input(X) \leftarrow not \neg input(X), url(X);$$

$solve(P', M)$ in Step (d) yields the answer sets of P', where M is the set of the original facts from P (the two URIs).

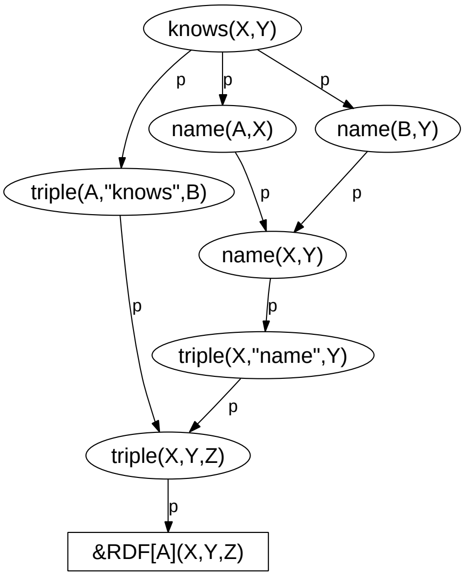

Figure 4: Pruned dependency graph.

P' is removed from P and $\overline{C}$ from the dependency graph (the resulting subgraph is shown in Figure 4). Continuing with (a), now the external component C is contained in T_c, and therefore in Step (b) evaluated for each set in $\mathcal{M}$. After removing C from $Comp$, $\overline{C}$ is empty in Step (d) and $P' = P_{hex}$, i.e., an ordinary, stratified program, which is evaluated against each set in $\mathcal{M}$ - note that these sets now also contain the result of the external atom, represented as ground replacement atoms. At this point, P is empty and the algorithm terminates, having $\mathcal{M}$ as result.

We obtain the following property:

Theorem 1 *Let P be a* HEX*-program and $\mathcal{M}$ the output of the evaluation algorithm from Figure 3. Then, M is an answer set of P iff $M \in \mathcal{M}$.*

Proof 1 *(Sketch). idea: The Splitting Set Theorem of Lifschitz and Turner says (slightly restructured, hope still right): Let U be a splitting set for a program P. Then, a set A is an answer set of program P iff A is an answer set of $P' = P \setminus b_U \cup B$ for some B which is an answer set of b_U. Step (a) of the algorithm finds subprograms that are bottoms of P. Evaluating each bottom in step (b) means to apply the splitting set theorem. If one of the components was inconsistent, then the entire program must be inconsistent and no answer set exists (Step (c)). Step (d) again applies the splitting set theorem on the remaining program. This means, after each cycle of the algorithm, the answer set(s), represented as set(s) of ground facts grow(s), while the program is pruned. Finally, all answer sets of P are left.*

Available External Atoms

External Atoms are provided by so-called *plugins*, i.e., libraries that define one or more external atom functions. Currently, we implemented the *RDF plugin*, the *Description Logics Plugin* and the *String Plugin*.

The RDF Plugin RDF (Resource Description Framework) is a language for representing information about resources in the World-Wide Web and is intended to represent meta-data about Web resources which is machine-readable and -processable. RDF is based on the idea of identifying objects using Web identifiers (called *Uniform Resource Identifiers*, or URIs), and describing resources in terms of simple properties and property values. The *RDF plugin* provides a single external atom, the $\&RDF$ atom, which enables the user to import RDF-triples from any RDF knowledge base. It takes a single constant as input, which denotes the RDF-source (a file path or Web address).

The Description-Logics Plugin Description logics are an important class of formalisms for expressing knowledge about concepts and concept hierarchies (often denoted as *ontologies*). The basic building blocks are *concepts*, *roles*, and *individuals*. Concepts describe the common properties of a collection of individuals and can be considered as unary predicates interpreted as sets of objects. Roles are interpreted as binary relations between objects. In previous work (Eiter *et al.* 2004), we introduced *dl-programs* as a method to interface description-logic knowledge bases with answer-set programs, allowing a bidirectional flow of information. To model dl-programs in terms of HEX-programs, we developed the *description-logics plugin*, which includes three external atoms (these atoms, in accord to the semantics of dl-programs, also allow for extending a description logic knowledge base, before submitting a query, by means of the atoms' input parameters):

- the $\&dlC$ atom, which queries a concept (specified by an input parameter of the atom) and retrieves its individuals,
- the $\&dlR$ atom, which queries a role and retrieves its individual pairs, and
- the $\&dlConsistent$ atom, which tests the (possibly extended) description logic knowledge base for consistency.

The description-logics plugin can access OWL ontologies, i.e., description logic knowledge bases in the language $\mathcal{SHOIN}(\mathbf{D})$, utilizing the RACER reasoning engine (Haarslev & Möller 2001).

The String Plugin For simple string manipulation routines, we provide the string plugin. It currently consists of a single atom, the $\&concat$ atom, which lets the user specify two constant strings in the input list and returns their concatenation as a single output value.

Current Prototype

dlvhex has been implemented as a command-line application. It takes one or more HEX-programs as input and directly prints the resultant models as output. Both input and output are given in classical textual logic-programming notation. For the core reasoning process, dlvhex itself needs the answer-set solver DLV (Leone *et al.* 2005) (and DLT (Ianni *et al.* 2004) if F-Logic syntax is used).

Assuming that the program from Example 2 is represented by the file `rdf.lp`, dlvhex is called as follows:

```
user@host:~> dlvhex --filter=friend rdf.lp
```

The `--filter` switch reduces the output of facts to the given predicate names. The result contains two answer sets:

```
{knows("Giovambattista Ianni",
        "Axel Polleres"),
 {knows("Giovambattista Ianni",
        "Francesco Calimeri"),
 {knows("Giovambattista Ianni",
        "Wolfgang Faber"),
 {knows("Giovambattista Ianni",
        "Roman Schindlauer")}

 {knows("Roman Schindlauer",
        "Giovambattista Ianni"),
 {knows("Roman Schindlauer",
        "Wolfgang Faber"),
 {knows("Roman Schindlauer",
        "Hans Tompits")}
```

We will make dlvhex available both through source and binary packages. To ease becoming familiar with the system, we also offer a simple Web-interface available at

`http://www.kr.tuwien.ac.at/research/dlvhex.`

It allows for entering a HEX-program and filter predicates and displays the resultant models. On the same Web-page, we also supply a toolkit for developing custom plugins, embedded in the GNU autotools environment, which takes care for the low-level, system-specific build process and lets the plugin author concentrate his or her efforts on the implementation of the plugin's actual core functionality.

References

Calvanese, D.; Giacomo, G. D.; and Lenzerini, M. 2001. A Framework for Ontology Integration. In *Proceedings of the First Semantic Web Working Symposium*, 303–316.

Eiter, T.; Ianni, G.; Schindlauer, R.; and Tompits, H. 2004. Nonmonotonic Description Logic Programs: Implementation and Experiments. In *Logic for Programming, Artificial Intelligence, and Reasoning, 11th International Conference, LPAR 2004*, 511–527.

Eiter, T.; Ianni, G.; Schindlauer, R.; and Tompits, H. 2005. A Uniform Integration of Higher-Order Reasoning and External Evaluations in Answer Set Programming. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI-05)*. Morgan Kaufmann.

Eiter, T.; Ianni, G.; Schindlauer, R.; and Tompits, H. 2006. Effective Integration of Declarative Rules with external Evaluations for Semantic Web Reasoning. In *European Semantic Web Conference 2006, Proceedings*. To appear.

Gelfond, M., and Lifschitz, V. 1991. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing* 9:365–385.

Haarslev, V., and Möller, R. 2001. RACER System Description. In *Proceedings IJCAR-2001*, volume 2083 of *LNCS*, 701–705.

Ianni, G.; Ielpa, G.; Pietramala, A.; Santoro, M. C.; and Calimeri, F. 2004. Enhancing Answer Set Programming with Templates. In Delgrande, J. P., and Schaub, T., eds., *Proceedings NMR*, 233–239.

Kifer, M.; Lausen, G.; and Wu, J. 1995. Logical Foundations of Object-Oriented and Frame-Based Languages. *J. ACM* 42(4):741–843.

Leone, N.; Pfeifer, G.; Faber, W.; Eiter, T.; Gottlob, G.; Perri, S.; and Scarcello, F. 2005. The DLV System for Knowledge Representation and Reasoning. *ACM Transactions on Computational Logic*. To appear.

Lifschitz, V., and Turner, H. 1994. Splitting a Logic Program. In *Proceedings ICLP-94*, 23–38. Santa Margherita Ligure, Italy: MIT-Press.

Przymusinski, T. 1988. On the declarative semantics of deductive databases and logic programs. In *Foundations of Deductive Databases and Logic Programming*. Morgan Kaufmann. 193–216.

Ross, K. A. 1994. Modular stratification and magic sets for datalog programs with negation. *J. ACM* 41(6):1216–1266.

Sintek, M., and Decker, S. 2002. TRIPLE - A Query, Inference, and Transformation Language for the Semantic Web. In *International Semantic Web Conference*, 364–378.

Wang, K.; Antoniou, G.; Topor, R. W.; and Sattar, A. 2005. Merging and Aligning Ontologies in dl-Programs. In Adi, A.; Stoutenburg, S.; and Tabet, S., eds., *Proceedings First International Conference on Rules and Rule Markup Languages for the Semantic Web (RuleML 2005), Galway, Ireland, November 10-12, 2005*, volume 3791 of *LNCS*, 160–171. Springer.