Decomposition of Declarative Knowledge Bases with External Functions*

Thomas Eiter and Michael Fink and Thomas Krennwallner
Institute of Information Systems, Vienna University of Technology
Favoritenstrasse 9–11, A-1040 Vienna, Austria
{eiter,fink,tkren}@kr.tuwien.ac.at

Abstract

We present a method to decompose a declarative knowledge base, given by a logic program under Answer Set Semantics with access to external sources. It overcomes the ineffectiveness of current methods due to a lack of structural information about these sources, viewed as black boxes, by exploiting independency information in accesses to them. To this end, we develop a generic notion of domain independence that allows to restrict the evaluation domain and, as a consequence, to prune unnecessary dependency assumptions between atoms. This leads to increased decomposability; we demonstrate this by an evaluation method for HEX-programs based on program rewriting, which may yield large performance gains. While developed for a particular formalism, the notions and ideas of this paper might be adapted to related formalisms as well.

1 Introduction

In the last years, there has been a steady increase of the desire to integrate different information sources and software into applications, in order to provide enhanced usability. To address this need, some declarative knowledge representation formalisms have been extended with the capability to access external sources. Often, this is realized via an interface in the style of an API; examples of rule based such formalisms are various extensions of ASP solvers like DLV, XSB,\(^1\) Agent Programs [Subrahmanian et al., 2000], dl-programs [Eiter et al., 2008], defeasible description logic [Wang et al., 2004], Prolog engines with an interface for external functions, etc.

However, the enhanced capabilities of such formalisms come at a performance price. Indeed, structural dependency information, which is usually exploited for efficient evaluation, is blurred or completely lost by access through external sources. This is an inevitable consequence of treating these sources as black boxes, which causes one to cautiously assume that numerous dependencies exists, even if this not true in reality. This may in particular be detrimental in case of mutual dependency between the information in the knowledge base and the external source, which can lead to almost universal dependency assumptions. As a result, decomposing the knowledge base into smaller, local parts for evaluation (like strongly connected components or splitting sets [Lifschitz and Turner, 1994]) is hindered, and large parts have to be evaluated in an uninformed generate and test manner.

We illustrate this on the canonical Nixon Diamond, couched in HEX-programs [Eiter et al., 2006a], which extend Answer Set Programming (ASP) with access to external sources via external atoms, in a setting where default rules should be imposed on top of an external ontology.

Example 1 (Nixon Diamond) Suppose \(T = \{r(n), q(n), \forall x. np(x) \equiv \neg p(x)\}\) contains the facts that Nixon was a republican \((r(n))\) and a quaker \((q(n))\); we further know that (i) quakers are normally pacifists, while (ii) republicans are normally not pacifists. In spirit of dl-programs [Eiter et al., 2008], the HEX-program \(P_N\) below encodes (i) and (ii) in the rules (2) and (3), resp.; there, \(a(p, X)\) and \(a(np, X)\) encode the (reified) assumptions that \(X\) is a pacifist resp. is not a pacifist. The program accesses \(T\) via external atoms of form \&\(T_\alpha[a](X)\), which evaluate to true if, after adding all facts \(\pi(c)\) to \(T\) such that \(a(\pi, c)\) is in the model of the program, the atom \(\chi(X)\) is true in \(T(d(X))\) is a simple domain predicate:

\[
d(n). \tag{1}
\]

\[
a(p, X) \leftarrow d(X), \& T_r[a](X), \text{not} \& T_{np}[a](X). \tag{2}
\]

\[
a(np, X) \leftarrow d(X), \& T_q[a](X), \text{not} \& T_p[a](X). \tag{3}
\]

Without explaining the semantics here in detail, \(P_N\) has two models, \(\{d(n), a(p, n)\}\) and \(\{d(n), a(np, n)\}\), corresponding to the views that Nixon is a pacifist resp. is not a pacifist.

Possible dependencies between atoms are shown in the dependency graph of \(P_N\) in Figure 1, where an arc from \(\alpha\) to

\[\text{Figure 1: Dependency graph of } P_N\]

---

\(*\)This work has been supported by the Austrian Science Fund (FWF) projects P20840 and P20841.

\(1\)See www.dlvsystem.com, xsb.sourceforge.net

\[\alpha \]
β stands for a (possible) dependency of α from β, and “n” means dependency through negation (“not”). All atoms except d(X) mutually depend on each other, and form one component, C2, for evaluation, which depends on the (singleton) component C1. Now suppose we have m clones of Nixon and d(n1), . . . , d(nm) instead of d(n1) in (1). Then the respective program Pm N instantiates to m copies, and we have 2m models that contain d(n1), . . . , d(nm) and one of α(p, ni), α(np, ni), for each i = 1, . . . , m. However, the dependency graph of Pm N’s grounding, which traditionally is the basis for efficient evaluation, is not the union of k copies of the dependency graph of Pn N’s grounding, as intuitively expected: there is, e.g., an arc &Tg[a](n1) → α(p, n2), reflecting that α(p, n2) could impact the query p(n1) to T with update a. In fact, according to the dependency graph of Pm N’s grounding, each atom except the d(ni) are on negative cycles, in particular all external atoms; a standard evaluation method thus has to consider all these atoms at once, along with (exponentially many) different guesses for possible input a to them. This results in long execution times.

Due to the black box view, a traditional evaluation of Pm N is unaware that each external atom actually depends only on two facts, e.g., &Tg[a](n1) on α(p, n), α(np, n), and that (as by intuition) indeed a splitting of the Pm N into k disjoint copies is feasible which can be evaluated independently.

This calls for refined methods to decompose a declarative knowledge base with access to external sources. In this paper, we address this issue and make the following contributions.

• We consider domain independence information as a basis for more effective decomposition of programs with access to external sources. Domain independence means that the evaluation of an external atom depends, for a given set of arguments, only on a subset of the domain, and that other elements in the domain can be safely ignored. Building on this, the idea is to split the domain in different parts such that external atoms can be evaluated independently in different parts. This is formalized as greatest local split of a set of ground atoms, which is algebraically defined in Section 3.

• Using greatest local splits, we refine the traditional method of evaluating programs using the ground dependency graph, by pruning unnecessary dependency assumptions of atoms caused by external functions (Section 4).

• We develop a decomposition method for HEX-programs that uses program rewriting on the non-ground level, based on independence information (Section 4). The evaluation of programs resorts then to standard technology, such that the method can be realized with rather little effort (Section 5).

Some experiments with an implementation, in particular on the program in Example 1 above, show that the rewriting may lead to large performance gains and scalability.

2 Preliminaries

Partial orders and partitions. Recall that a poset (S, ≤) has a binary relation ≤ on a set S that is reflexive, anti-symmetric, and transitive. For any X ⊆ S, an element u ∈ S is an upper bound of X, if b ≤ u for all b ∈ X, and is the least upper bound of X, if u ≤ u′ for all upper bounds u′ of X.

A partition π on a set S is a collection of nonempty disjoint subsets of S such that S = ∪π.

Syntax of HEX-Programs. Let C, X, and G be mutually disjoint sets whose elements are called constant names, variable names, and external predicate names, respectively. Unless explicitly specified, elements from X (resp., C) are denoted with first letter in upper case (resp., lower case), while elements from G are prefixed with the “&” symbol. We note that constant names serve both as individual and predicate names.

Elements from C ∪ X are called terms. A higher-order atom (or atom) is a tuple (Y0, Y1, . . . , Yn), where Y0, . . . , Yn are terms; n ≥ 0 is the arity of the atom. Intuitively, Y0 is the predicate name; we thus use the more familiar notation Y0(Y1, . . . , Yn). The atom is ordinary, if Y0 is a constant.

An external atom is of the form

\&g[Y1, . . . , Yn](X1, . . . , Xm) (4)

where Y1, . . . , Yn and X1, . . . , Xm are two lists of terms (called input and output lists, respectively), and &g ∈ G is an external predicate name. We assume that &g has fixed lengths m(&g) = n and out(&g) = m for input and output lists, respectively. Intuitively, an external atom provides a way for deciding the truth value of an output tuple depending on the extension of a set of input predicates. For any set of atoms A, we denote by A the set of external atoms in A.

A rule r is of the form

α1 ∨ · · · ∨ αk ← β1, . . . , βm, not βm+1, . . . , not βn (5)

where m, k ≥ 0, each αi is an atom, and each βj is an atom or an external atom. We let H(r) = {α1, . . . , αk} and B(r) = B+(r) ∪ B−(r), where B+(r) = {β1, . . . , βm} and B−(r) = {βm+1, . . . , βn}. Rule r is a constraint, if H(r) = ∅ and B(r) ≠ ∅, and a fact, if B(r) = ∅ and H(r) ≠ ∅.

A HEX-program (or program) is a finite set P of rules; it is ordinary, if all its rules are ordinary, i.e., it contains only ordinary atoms.

Example 2. The program PN for the Nixon Diamond above consists of one fact and two rules that use negation “not”, and they have external atoms &Tg[a](X), &Tnp[a](X), etc.}

Semantics of HEX-Programs. The Herbrand base HB P of the program P is the set of all possible ground instances of atoms and external atoms in P using constants from C. The grounding of a rule r, grd(r), is analogous, and the grounding of P is given by grd(P) = ∪r∈P grd(r). By default, C, X, and G are implicitly given by P.

An interpretation of P is any subset I ⊆ HB P that contains only ordinary atoms; we say that I is a model of

• an atom α ∈ HB P, denoted I |= α, if α ∈ I,
• a ground external atom α = &g[c2](c1) (denoted I |= α), if fκg(I, c2, c1) = 1, where fκg is a (fixed) (n+m+1)-ary Boolean function for &g, where n = in(&g), m = out(&g), I ⊆ HB P, and c2 ∈ Cn, c1 ∈ Cm.
• a ground rule r (I |= r), if I |= H(r) or I |= B(r), where (i) I |= H(r), if I |= α for some α ∈ H(r) and (ii) I |= B(r), if I |= α for all α ∈ B+(r) and I ⊬ α for all α ∈ B−(r).
• a program P (I |= P), if I |= r for all r ∈ grd(P).

The FLP-reduct [Faber et al., 2004] of a program P wrt. an interpretation I is IP = {r ∈ grd(P) | I |= B(r)}; then I is an answer set of P, if I is a ¬-minimal model of IP. For ordinary P, this amounts to answer sets as in [Gelfond and
We start with fixing some notation. In what follows, the elements of the domain into disjoint sets, such that the evaluation of an external atom only depends on the elements that are in the sets associated with the arguments of the atom. We call a suitable such partitioning a basis. For program decomposition, we need to take the structure of the rules into account, respecting syntactic dependency of atoms and arguments according to the dependency graph, assuming that elements in the same partition are mutually dependent. This leads us to a refinement of the basis (i.e., sets are further partitioned) to a unique greatest local split of a set of atoms. The latter is defined algebraically.

**Definition 1 (Basis)** Let \( D \subseteq P \), let \( \alpha = \& g(p)c \) be a ground instance of an external atom over \( D \) with Boolean function \( f_{\&g} \). We call \( B \) a basis for \( \alpha \), if for all interpretations \( I \) and \( B' \supseteq B \), \( f_{\&g}(I \cap p[B], p, c) = f_{\&g}(I \cap p[B'], p, c) \).

**Example 5** Let \( D \) be a domain, \( I = \{ p(a), p(b), q(a) \} \) be an interpretation, and \( p = p, q \). Then \( B_1 = \{ a \} \) and \( B_2 = \{ b, c \} \) be two subsets of \( D \). Then \( I \cap p[B_1] = \{ p(a), q(a) \} \) and \( I \cap p[B_2] = \{ p(b) \} \). Now suppose \( \alpha = \& g(p)c \) is such that for all interpretations \( I \), \( f_{\&g}(I, p, a) = 1 \) if \( p(a) \in I \). Then, \( B_1 \) is a basis for \( \alpha \), and \( f_{\&g}(I \cap p[B_1], p, a) \).

Given an external atom \( \alpha \), define for any \( a, b \in D \) that \( a \sim_\alpha b \) if \( a, b \in B \) for basis \( B \), and \( c \sim_\alpha c \) for every \( c \in D \). Then \( \sim_\alpha \) is an equivalence relation, whose quotient set \( D/\sim_\alpha \) forms a partition of \( D \).

The notion of greatest local split is now defined as follows.

**Definition 2 (Greatest Local Split)** For any set of ground atoms \( A \), the greatest local split of \( A \), denoted \( gls(A) \), is the partition \( \pi \in D \) which is the least upper bound of \( B = \{ D/\sim_\alpha \mid \alpha \in \&g(A) \} \) in \( (D, \preceq) \) (recall \&A from above).

Obviously, \( gls(A) \) is unique and always exists.

**Example 6** Let \( \sim_\alpha = \{ \{ a, a \}, \{ b, b \}, \{ c, c \}, \{ d, d \} \} \) and \( \sim_\beta = \{ \{ a, a \}, \{ b, b \}, \{ c, c \}, \{ d, d \} \} \) be equivalence relations over \( D = \{ a, b, c, d \} \) for \( \alpha \) and \( \beta \), respectively. Then, \( D/\sim_\alpha \) and \( D/\sim_\beta \) are two possible \( \sim_\alpha \) and \( \sim_\beta \) of external atoms in \( A \). We obtain \( gls(A) = D/\sim_\alpha = \{ \{ a, b \}, \{ c \}, \{ d \} \} \), whenever all possible \( \sim_\alpha \) or \( \sim_\beta \) of external atoms in \( A \) are equal to \( \sim_\alpha \) or \( \sim_\beta \).

The following result then, whose proof is not difficult, forms the basis for exploiting joint dependency information.

**Proposition 1** Let \( \& g[p]c \) be an external atom, let \( A \) be a set of atoms such that \( \& g[p]c \in \text{ground}(A) \), and let \( I \) be an interpretation. Then, for every partition \( \pi \) such that \( gls(A) \leq \pi \), there exists some \( B' \in \pi \) such that

\[
f_{\&g}(I, p, c) = f_{\&g}(I \cap p[B'], p, c).
\]

By this proposition, we can partition the domain, at different levels of granularity, in a way such that we can evaluate the atoms in \( A \) in different sets of the partition.

**Proposition 2** Given a set of atoms \( A \) and \( B \) as in Def. 2, we can compute \( gls(A) \) in linear time in the size of \( A \) and \( B \).

Indeed, we can compute \( gls(A) \), e.g., as connected components of a graph that we construct from \( B \) in linear time.

## 4 Rewriting with Domain Independence

In this section, we apply the results obtained in the previous section in order to improve the evaluation of HEX-programs.
by taking dependency information of external atoms into account. The idea is to first prune unnecessary ground dependencies from the dependency graph $G_P$ of a given program $P$, like for & $T_f[a](n_1) \rightarrow a(p, n_2)$ in the grounding of $P^N_n$ with $n$ Nixon clones. In a second step, we rewrite the program into an equivalent program, such that the evaluation of strongly connected components is restricted to relevant subdomains whenever possible. Intuitively, this step accounts for unnecessary dependences between non-ground atoms.

In this section, we assume that the domain $D$ is finite; note that even for a basically infinite set $C$, $P$ may be equivalent to its grounding over a finite subset of $C$; e.g., domain-expansion safe programs [Eiter et al., 2006a] has this property.

**Step 1: Dependency Graph Pruning.** We assume that we get a HEP-program without higher order atoms—note that w.l.o.g. higher order atoms can be removed by partial grounding—and that every ground external atom in the program has an associated basis $B_a$ in $D$. Furthermore, for a proper treatment of the dependency between input and output of a ground external atom, we assume bases $B_a$, such that all elements of the output $c$ of $\alpha$ are in $B_a$.

We first reduce the dependency graph $G_P$ to a graph $G'_P$ as follows. For any graph $G$, let $SCC(G)$ be the set of strongly connected components in $G$. Consider any component $C \in SCC(G_P)$ and $grnd(\text{ground}(C))$ (which is uniquely determined by the bases of external atoms in $\text{ground}(C)$). We remove every arc $g[p](c_1) \rightarrow p(c_2)$ from $G$, such that $c_1$ and $c_2$ are variable-free and there is an element in $c_2$ which does not belong to the partition of $c_1$ in $\text{ground}(C)$. Let $G'_P$ denote the resulting graph. Then we can evaluate $P$ using $G'_P$ instead of $G_P$. More formally, this is detailed as follows.

Let $\text{Comp} = \{C_1, \ldots, C_n\}$ be a partitioning of the atoms occurring in $P$ such that all $\text{ground}(C_i)$ are pairwise disjoint, and let $P_{C_i}$ contain all rules $r \in \text{ground}(P)$ such that $H(r) \cap \text{ground}(C_i) \neq \emptyset, i = 1, \ldots, n$. A partial ordering $\mathcal{E}(P) = (\text{Comp}, \sqsubseteq)$ is then an evaluation ordering for $P$, if $H(r) \sqsubseteq \text{ground}(C_i)$, for each $r \in P_{C_i}, i = 1, \ldots, n$, and the following property (6) holds. Let $P_{C_i} \downarrow = \bigcup_{C_i \sqsubseteq C_i, C_i \in \text{Comp}} P_{C_i}$; then

$$\text{AS}(P_{C_i} \downarrow) = \bigcup_{M_j \in \text{AS}(P_{C_i} \downarrow), j = 1, \ldots, n_i} \text{AS}(P_{C_i} \downarrow \cup \bigcup_{j = 1}^{n_i} M_j);$$

(6)

where $\{C_{i_1}, \ldots, C_{i_{n_i}}\} = \{C_j \subseteq C_i \mid \forall C_j', C_j \not\subseteq C_j' \subseteq C_i \Rightarrow C_j = C_j'\}$ are the maximal sets $C_j$ smaller than $C_i$. That is, the answer sets of the rules associated with $C_i$ and all $C_j$ below are obtained as the answer sets of the rules associated with $C_i$ to which the answer sets of the rules associated with $C_j$ and all $C_j' \subseteq C_j$ below are added.

Then, given that $C_{i_1}, \ldots, C_{i_l}$ are the maximal elements of $\mathcal{E}(P)$ and $P_{con}$ are all constraints in $P$, it holds that

$$\text{AS}(P) = \{N = \bigcup_{j = 1}^{l} M_{i_j} \mid M_{i_j} \in \text{AS}(P_{C_{i_j} \downarrow}), 1 \leq j \leq l, N \models P_{con}\}.$$  

It is well-known that $\text{SCC}(G_P)$ is an evaluation ordering for $P$, cf. [Eiter et al., 2006a]. We similarly have:

**Proposition 3 SCC($G'_P$) is an evaluation ordering for $P$.**

Intuitively, the strongly connected components of $G'_P$ refine those of $G_P$ by removing unnecessary input dependencies of external atoms. We note that coarser evaluation orderings can be obtained e.g. by collapsing suitable components in SCC($G_P$) resp. SCC($G'_P$). As for the precise evaluation ordering is immaterial, we work in the sequel with a generic given evaluation ordering $\mathcal{E}(P)$.

**Step 2: Program Rewriting.** We proceed by introducing a rewriting of $P$ wrt. $\mathcal{E}(P)$ in order to further improve the evaluation of non-ground programs. The idea is to restrict the applicability of rules with external atoms to relevant ground instances given respective domain independence information. To avoid inhibiting the application of relevant ground instances, and to keep the rewriting general and simple, we consider here the maximal number $k = k(P)$ of distinct variables in a rule $r \in P$ as a bound for potential dependencies.\(^3\)

In the following, we let for every $p \in C$ be $p^\dagger$ a fresh symbol with arity of $p$. For a set of atoms $A$, let $\text{const}(A) = \{c_1, \ldots, c_n \mid p(c_1, \ldots, c_n) \in A \wedge A\} \cap C$.

The i-restricting substitution in $A$ is the partial substitution

$$\sigma_A = \{p/p^\dagger \mid p(t) \in A\},$$

which maps predicates $p$ of every ordinary atom $p(t)$ in $A$ to $p^\dagger$. An i-restricting $A$ in $P$, short $P_A$, is the set of rules $\{\{r/h\} \leftarrow B(r)^\dagger, \text{dom}(X_1), \ldots, \text{dom}(X_r) \mid r \in b_A(P)\}$, where $\text{dom}(\alpha)$ is a unary predicate symbol and $X_1, \ldots, X_r$ are all output variables which occur in some external atoms in $B(r)$. Furthermore, let

$$\pi_A^\dagger(A) = \{S_{1} \cup \cdots \cup S_k \cup \text{const}(A) \mid S_{1}, \ldots, S_k \in \pi\}$$

be the k-fold union of a partition $\pi$ augmented with $\text{const}(A)$.

**Definition 3 (Decomposition $d_C(P)$)** Let $C \in \mathcal{E}(P)$ and $\text{gls}(\text{ground}(C))^\dagger(C) = \{D_1, \ldots, D_k\}$ for $k = k(P)$. The decomposition of $P$ wrt. $C$, denoted $d_C(P)$, is defined as

$$\bigcup_{l \leq l \leq k} P_{C} \cup \{\alpha \leftarrow \theta \mid \alpha \in H(r), r \in b_C(P)\} \cup \text{dom}[D_i].$$

**Example 7** Reconsider the cloned version $P^N_n$ of Example 1 and an evaluation order with the component $C_1, C_2$ in Figure 1, where $C_1 \sqsubseteq C_2$. Note that this is the evaluation order given by the strongly connected components of $G_P$ (which is not affected by the pruning step). Given that $\& T_k[a](n_i)$ only depends on $a(p, n_i), a(np, n_i)$, for $1 \leq i \leq m$, we obtain that $\text{gls}(\text{ground}(C_2))^\dagger(C_2) = \{\{n_1, np\} \mid 1 \leq i \leq m\}$.

Thus, the decomposition of $P^N_n$ wrt. $C_2$ contains $m$ rewriting rules of rule (2), e.g., for $i = 1$ given by

$$a^\dagger(p, X) \leftarrow a^\dagger(X), \text{ & } T_k[a^\dagger](X), \text{ not & } T_k[a](X^\dagger), \text{ dom}^\dagger(X),$$

where $\text{dom}^\dagger$ ranges over $D_1 = \{p_1, np\}$. Similarly for rule (3). This reduces the applicability of the two rules to the corresponding domain $D_i$ of the respectively rewritten rules, which intuitively amounts to removing any dependencies to ordinary ground atoms (inputs) in other subdomains.

\(^3\)This is a cautious overestimate, and often $k$ may be smaller.
every \( \text{dom}^i(c) \subseteq B(r^i) \). Given \( C \in \mathcal{E}(P) \), with 
\( \text{gls}(\text{grnd}(C))^k(C) = \{D_1, \ldots, D_k\} \) for \( k = |P| \), and 
(a set \( N \subseteq \text{HB}_P \) of ground ordinary atoms, let \( \mathcal{M}(N) \) be the set 
\[
\mathcal{M}(N) = N \cup \bigcup_{i=1}^{k} N^i \cup \text{dom}^i[D_i],
\]
where \( N^i = \{p^i(c) \mid p(c) \in N \land (\varphi_a \lor \varphi_b)\} \), with conditions 
\( \varphi_a = \exists r^i \in \text{grnd}(C) \land p(c) \not\in \text{grnd}(C) \), and 
\( \varphi_b = \exists r^i \in \text{grnd}(dc(P)) : r^i \) is \( D_i \)-applicable \( \land p^i(c) \in 
H(r^i) \cup B(r^i) \land N = B(r^i) \land \text{grnd}(C) \cup B(r^i) \).

Furthermore, for any set \( M \subseteq \text{ground ordinary atoms}, 
\( M_c = \{p(c) \in M \mid p \in \mathcal{C}\} \). Obviously, if \( M \) is of the 
form \( \mathcal{M}(N) \), then \( N = M_c \).

Informally, given a set \( N \) of ground ordinary atoms, in 
adition to the original atoms and a respective domain closure (i.e., \( \text{dom}^i[D_i] \) atoms) \( \mathcal{M}(N) \) contains a rewritten atom \( p^i(c) \) for some \( p(c) \in N \): If \( p(c) \) is an atom that is modified by the 
rewriting and appears in a potentially applicable rule in \( P \), (Condition \( \varphi_a \)), or if \( p(c) \) is an input atom, i.e., one that is not 
modified by the rewriting, such that the predicate \( p \) is subject to 
rewriting, however (Condition \( \varphi_b \)). The latter guarantees 
that it is taken into account as (potential) input for rewritten 
external atoms (as it is for the respective unmodified external atoms).

Towards correctness of the rewriting technique, we first establish 
the following lemma.

**Lemma 4** Let \( C \in \mathcal{E}(P) \), let \( r \in \text{grnd}(bc(P)) \), and let \( r^i \in \text{grnd}(dc(P)) \) such that \( r^i \) is \( D_i \)-applicable. Suppose \( N \subseteq \text{HB}_P \) consists of ordinary ground atoms such that, for every 
\( \& (p(c) \in B(r) \cap \text{grnd}(C)) \) and ground atom \( \alpha \), it holds that 
\( \alpha \in N \cap p[D_i] \) if \( \alpha \) is pairwise disjoint sets of com-
ponents, such that \( \bigcup_{i=1}^{k} E_i = \text{Comp} \) and for every \( C \in E_i \), 
\( C' \in E_j \), and \( i < j \), it holds that \( C' \not\subseteq C \). For an 
iterative recursion, we recursively define sets \( I \) of sets of ground 
orinary atoms as follows: \( I_0 = \{\emptyset\} \), and \( I_{i+1} = \bigcup_{C \in E_i} \{N \mid N = C \cup_{C \in E_i} N_c(I)\} \), where 
\( N_c(I) = M_c \) if \( M \in \mathcal{M}(dc(P) \cup M(I)) \) and \( dc(P) \) is a unique 
rewriting technique for the evaluation of a program.

**Example 8** Consider a cloned version \( P''_N \) of Example 1 
with the evaluation order \( C_1 \subseteq C_2 \). For \( C_1 \), which is a set of 
facts, we obtain a single answer set of \( dc(C_1,P'')_N \): \( N_{C_1}(\emptyset) = \{d(n)_i \mid 1 \leq i \leq m\} \). Since 
\( dc(C_1,P'') \) is a unique split wrt. \( N_{C_1}(\emptyset) \), the we can compute the answer 
sets of \( P''_N \) as the sets \( M_{C_1} \), where \( M \in \mathcal{M}(dc(C_2,P''_N) \cup M(N_{C_1}(\emptyset))) \).

Observe that for ground programs \( P \), \( k(P) = 0 \), i.e., rule 
application cannot be further constrained by domain restrictions.

This is reflected in the rewriting, which decomposes \( P \) into a single copy for every component without any \( \text{dom} \) 
predicates in the body. However, for ground programs, the 
pruning of dependencies as described in Step 1 is most effective, 
and captures the cases otherwise handled by the rewriting.

On the other hand, for non-ground programs, the evaluation 
may be improved by partial grounding, which reduces \( k(P) \) (thus the number of domain partitions to be joined in a 
k-fold union). We leave a more fine-grained analysis of the program in order to reduce \( k(P) \) as further work for optimization.

## 5 Realization and Experimentation

The standard algorithm to evaluate a given \( \text{hex-program} P \) 
was shown in [Eiter et al., 2006b]. Its principal idea is to 
break up the computation into smaller tasks, and, in spirit of 
[Lifschitz and Turner, 1994], to perform a bottom-up evaluation 
of the using the strongly connected components of \( G_P \).

The procedure uses a subroutine \( \text{eval}(P, C, I) \), which helps 
creating the models wrt. a component \( C \) and an interpretation 
\( I: \text{eval} \) keeps track of cycles in \( C \) and chooses the appropriate 
solving strategy for it, and generates models compliant
Algorithm 1: decomp-eval\((P, C, S, I)\): Compute the answer sets of a component \(C\) and greatest local split \(S\) in \(P\) wrt. \(I\)

**Input:** HEX-program \(P\), component \(C\), 

\[ S = \text{gls}(\text{grnd}(C)), \text{ interpretation } I \]

**Result:** collection of answer sets \(N\)

(a) Compute \(S(t) = \{D_1, \ldots, D_l\}\)

(b) for \(j = 1, \ldots, l\) do

\[
\begin{align*}
\text{Set } \sigma^j_C &= \{p/p^\alpha : p(t) \in C\} \\
\text{P}^j = \text{P} \cup \text{P}^j_C \cup \{\alpha \leftarrow \alpha \sigma^j_C : \alpha \in H(r), r \in b_C(P)\} \cup \text{dom}^j[D_j] 
\end{align*}
\]

end

(c) Evaluate \(P^j\) and store the answer sets of it in \(N\)

return \(N\)

with the external atoms in \(C\) (see [Eiter et al., 2006b] for the details).

We realize the evaluation of our decomposition method by adapting the standard evaluation algorithm. For that, we introduce the refined component evaluation subroutine \(\text{decomp-eval}\((P, C, S, I)\) shown in Algorithm 1. The major changes for the overall refined HEX-program evaluation are:

1. **after creating the dependency graph \(G_P\), we create from \(\text{SCC}(G_P) = \{C_1, \ldots, C_n\}\) the greatest local splits \(S_i = \text{gls}(\text{grnd}(C_i)), i = 1, \ldots, n\).**

2. **we generate the pruned dependency graph \(G'_{P}\) and perform the bottom-up evaluation wrt. \(\text{SCC}(G'_{P})\);**

3. **then, while evaluating the components, we check for each \(C_i\) whether \(\text{dc}_C(P)\) is a unique split wrt. \(I\) and call then**

   (i) \(\text{decomp-eval}\((P, C_i, S_i, I)\) instead of \(\text{eval}\) in order to evaluate a component \(C_i\) with greatest local split \(S_i\) in a HEX-program \(P\) wrt. a interpretation \(I\), and take, transparently to the evaluation procedure, dependency information into account; otherwise,

   (ii) if \(\text{dc}_C(P)\) is not a unique split wrt. \(I\), we just call \(\text{eval}\((P, C_i, I)\).

Regarding (2), we prune \(G_P\) to \(G'_{P}\) by stripping ground dependencies in \(C_i\) wrt. \(S_i\) as described in Section 4. The resulting dependency graph \(G'_{P}\), then consists of components \(\text{Comp}'\). The further evaluation of \(P\), which is kept as is, will be performed with \(\text{Comp}'\).

After preprocessing the components of \(P\) outlined as above, the refined components will be evaluated in the usual way. To this end, we may call \(\text{decomp-eval}\((P, C_i, S_i, I)\) as described in (3i), provided that \(\text{dc}_C(P)\) is a unique split wrt. \(I\) (see step (3)). Alternatively, we fall back to the standard HEX-program evaluation procedure \(\text{eval}(P, C, I)\), as done in (3ii).

Let \(C \in \text{SCC}(G'_{P}), S = \text{gls}(\text{grnd}(C)), \) and \(I\) be an interpretation. In \(\text{decomp-eval}\((P, C, S, I)\) from above, we first generate the \(k\)-fold union of \(S\) in step (a), create \(\text{dc}_C(P)\) in step (b), and then evaluate the decomposed program in step (c). The result is stored in \(N\). The following states the correctness.

**Proposition 8** Given a HEX-program \(P\), component \(C\), greatest local split \(S = \text{gls}(\text{grnd}(C))\), and interpretation \(I\), we have \(N \in \text{AS}(b_C(P) \cup I) \text{iff } M(N) \in N\), where \(N\) is the output of Algorithm 1.

For experimental evaluation, we adapted the development version of the dlvhex prototype system,\(^4\) which is an open source reasoner for HEX-programs.

We have realized our running example, the Nixon diamond, using external atoms that query a description logic knowledge base (DL-KB) in spirit of dl-atoms in description logic programs [Eiter et al., 2008]. To this end, we encoded \(T\) as DL-KB. The external atoms & \(T_{\chi}[a](c)\) are then realized as instance retrieval queries over \(T\) using a DL-reasoner (RacerPro 1.9.2beta).\(^5\) All tests have been conducted on a P4 3GHz Linux system with 1GB of main memory. For a program \(P\), we write short \(\alpha P\) for the standard HEX-program evaluation strategy, whereas \(dP\) denotes our new decomposition evaluation strategy using \(\text{decomp-eval}\).

We fixed \(a \sim_\alpha \alpha\) for all individuals \(a\) and all ground external atoms \(\alpha\) for \(dP\) 's. In the decomposed program evaluation of \(dP\) we have \(m+2\) components (see Figure 2a for a schematic view). We computed all \(2^m\) answer sets of the test programs, and the experimental results are displayed in Figure 2b. They show the effectiveness of the method; we

\(^4\) www.kr.tuwien.ac.at/research/systems/dlvhex/

\(^5\) www.racer-systems.com
gain an exponential speed-up with our decomposed evaluation strategy \( d_P^m \).

We have created another example by slightly adapting Example 1. If we add to \( P^N_0 \) the additional rule

\[
a(p, X) \leftarrow d(X), \text{not} \& T_{nra}[X].
\]

we obtain a program \( P_{nra}^m \) that disables the applicability of rule (3) and forces all Nixon to be pacifists. As above, we have fixed \( a \sim_a \alpha \) for all individuals and all ground external atoms \( \alpha \) for \( P_{nra}^m \). The evaluation with our decomposed programs then consists again of \( m+2 \) components. We computed the single answer set of each of these test programs.

The detailed outcome of both Nixon test series is listed in Table 1, where "--" means timeout (1 min). The results show that both \( oP^m_N \) and \( oP_{nra}^m \) evaluation have an early breakdown \((m=4)\) while all runs of the \( d_P^m \) evaluation up to \( m=16 \), and \( d_{P_{nra}}^m \) up to \( m=64 \), finished in time.

The check for uniqueness of the split \( dc(P) \) was fast (a few msecs) on our examples by employing a non-recursive datalog program.

Due to space limitations, we only report here the Nixon example. We have looked at similar encodings, like the Twetty example, and the experiments show a similar picture.

To get a better understanding of the improvements of the evaluation, notice that implementations that feature external functions usually adopt a guess and check strategy when an external atom \( \alpha = &g[y](x) \) appears in a (negative) cycle in the dependency graph. A guess represents a model of the cyclic program component and this model has to comply with the function \( f_x \) for \( \alpha \), which may be expensive to evaluate. The guesses must be made wrt. a domain of individuals (in the worst case, all constants). Using our theory of domain independence, we can rewrite the program such that components range over smaller domains. This not only effects that fewer guesses are made for the external atoms, but also that the number of (costly) compliance checks decreases.

6 Discussion and Conclusion

We have presented a method to decompose declarative knowledge bases with access to external sources, formalized as \( \text{HEX} \)-programs, by using domain independence information. The method may lead to a large (exponential) improvement in performance, as evidenced by examples. While our method has been developed for \( \text{HEX} \)-programs, the underlying ideas and principles are generic and may be adapted for other similar KR formalisms (cf. Introduction).

To the best of our knowledge, program optimization based on domain independence about external sources, with possible non-monotonic negation, has not been addressed before. Note that domain independence is well-known in databases [Abiteboul et al., 1995]. It is related to active domain semantics and usually enforced by safety of queries and rules.

Domain decomposition techniques are also used in constraint processing [Cohen and Jeavons, 2003], but in a different setting.

Several issues remain for further investigation. One issue is a more involved analysis of dependencies, to make the set of atoms for which greatest local splits are computed smaller, e.g., by respecting unifiability over transitive arcs or respecting syntactic conditions (cf. [Costantini, 2006]). Also the notions of greatest local split and \( k \)-fold union may be refined. An orthogonal direction is predicate splitting, where input predicates \( p \) are removed from external atoms \( \&g[\ldots,p,\ldots](X) \) whose values are independent of \( p \) wrt. relevant interpretations; this may be applied as a post processing after domain splitting.

Another issue is how to obtain dependency information. Clearly, this depends on the nature of the external sources, and there is no general recipe apart from (expensive) verification of the definitions. However, often the knowledge engineer has insight in dependency information that she can supply, like in the Nixon diamond example. To some extent, (in)dependency information may also be obtained by (semi)automated analysis of external sources. For logic knowledge bases, notions of relevance for reasoning might be exploited, and for others (e.g., wordnet ontologies, URL lists) specific methods developed.

References


Table 1: Nixon experiments, avg. running time in seconds

<table>
<thead>
<tr>
<th>( m )</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>( oP^m_N )</td>
<td>0.21</td>
<td>1.63</td>
<td>14.5</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>( dP^m_N )</td>
<td>0.22</td>
<td>0.43</td>
<td>0.65</td>
<td>0.91</td>
<td>2.33</td>
<td>28.12</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>( oP^m_{nra} )</td>
<td>0.21</td>
<td>1.53</td>
<td>14.2</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>( dP^m_{nra} )</td>
<td>0.23</td>
<td>0.41</td>
<td>0.61</td>
<td>0.85</td>
<td>2.12</td>
<td>5.81</td>
<td>16.27</td>
<td>49.83</td>
</tr>
</tbody>
</table>