HEX-Programs with Existential Quantification*

Thomas Eiter, Michael Fink, Thomas Krennwallner, and Christoph Redl

Institut für Informationssysteme, Technische Universität Wien
Favoritenstraße 9-11, A-1040 Vienna, Austria
{eiter,fink,tkren,redl}@kr.tuwien.ac.at

Abstract. HEX-programs extend ASP by external sources. In this paper, we present *domain-specific existential quantifiers* on top of HEX-programs, i.e., ASP programs with external access which may introduce new values that also show up in the answer sets. Pure logical existential quantification corresponds to a specific instance of our approach. Programs with existential quantifiers may have infinite groundings in general, but for specific reasoning tasks a finite subset of the grounding can suffice. We introduce a generalized grounding algorithm for such problems, which exploits domain-specific termination criteria in order to generate a finite grounding for bounded model generation. As an application we consider query answering over existential rules. In contrast to other approaches, several extensions can be naturally integrated into our approach. We further show how terms with *function symbols* can be handled by HEX-programs, which in fact can be seen as a specific form of existential quantification.

1 Introduction

Answer Set Programming (ASP) is a declarative programming approach which due to expressive and efficient systems like SMODELS, DLV and CLASP, has been gaining popularity for many applications [3]. Current trends in computing, such as context awareness or distributed systems, raised the need for access to external sources in a program, which, e.g., on the Web ranges from light-weight data access (e.g., XML, RDF, or data bases) to knowledge-intensive formalisms (e.g., description logics).

To cater for this need, HEX-programs [7] extend ASP with so-called external atoms, through which the user can couple any external data source with a logic program. Roughly, such atoms pass information from the program, given by predicate extensions, into an external source which returns output values of an (abstract) function that it computes. This convenient extension has been exploited for many different applications, including querying data and ontologies on the Web, multi-context reasoning, or e-government, to mention a few; however, it can also be used to realize built-in functions. The extension is highly expressive as also recursive data access is possible.

A particular feature of external atoms is *value invention*, i.e., that they introduce new values that do not occur in the program. Such values may also occur in an answer set of a HEX-program, e.g., if we have a rule like

$$lookup(X, Y) \leftarrow p(X), \&do\_hash[X](Y)$$

* This research has been supported by the Austrian Science Fund (FWF) project P20840, P20841, P24090, and by the Vienna Science and Technology Fund (WWTF) project ICT08-020.

where intuitively, the external atom $\&do\_hash[X](Y)$ generates a hash key Y for the input X and records it in the fact $lookup(X, Y)$. Here, the variable Y can be seen under existential quantification, i.e., as $\exists Y$, where the quantifier is externally evaluated, by taking domain-specific information into account; in the example above, this would be a procedure to calculate the hashkey. Such domain-specific quantification occurs frequently in applications, be it e.g. for built-in functions (just think of arithmetic), the successor of a current situation in situation calculus, retrieving the social security number of a person etc. To handle such quantifiers in ordinary ASP is cumbersome; they amount to interpreted functions and require proper encoding and/or special solvers.

HEX-programs however provide a uniform approach to represent such domain-specific existentials. The external treatment allows to deal elegantly with datatypes (e.g., the social security number, or an IBAN of bank account, or strings and numbers like reals), to respect parameters, and to realize partial or domain-restricted quantification of the form $\exists Y.\phi(X) \supset p(X, Y)$ where $\phi(X)$ is a formula that specifies the domain of elements X for which an existential value needs to exist; clearly, also range-restricted quantification $\exists Y.\psi(Y) \supset p(X, Y)$ that limits the value of Y to elements that satisfy ψ can be conveniently realized.

In general, such value invention on an infinite domain (e.g.,for strings) leads to infinite models, which can not be finitely generated. Under suitable restrictions on a program Π, this can be excluded, in particular if a finite portion of the grounding of Π is equivalent to its full, infinite grounding. This is exploited by various notions of *safety* of HEX-programs that generalize safety of logic programs.

In particular, *liberal domain-expansion safety (de-safety)* [6] is a recent notion based on term-bounding functions, which makes it modular and flexible; various well-known notions of safety are subsumed by it. For example, consider the program

$$\Pi = \{\ s(a); \quad t(Y) \leftarrow s(X), \&concat[X, a](Y); \quad s(X) \leftarrow t(X), d(X)\ \}, \qquad (1)$$

where $\&concat[X, a](Y)$ is true iff Y is the string concatenation of X and a. Program Π is safe (in the usual sense) but $\&concat[X, a](Y)$ could hold for infinitely many Y, if one disregards the semantics of $concat$; however, if this is done by a term bounding function in abstract form, then the program is found to be liberally de-safe and thus a finite part of Π's grounding is sufficient to evaluate it.

Building on a grounding algorithm for liberally de-safe programs [5], we can effectively evaluate HEX-programs with domain-specific existentials that fall in this class. Moreover, we in fact generalize this algorithm with *domain specific termination*, such that for non-safe programs, a finitely *bounded grounding* is generated. Roughly speaking, such a bounded grounding amounts to domain-restricted quantification $\exists Y.\phi(X) \supset p(X, Y)$ where the domain condition $\phi(X)$ is dynamically evaluated during the grounding, and information about the grounding process may be also considered. Thus, domain-specific termination leads to a partial (bounded) grounding of the program, Π', yielding *bounded models* of the program Π; the idea is that the grounding is faithful in the sense that every answer set of Π' can be extended to a (possibly infinite) answer set of Π, and considering bounded models is sufficient for an application. This may be fruitfully exploited for applications like query answering over existential rules, reasoning about actions, or to evaluate classes of logic programs with function symbols like FDNC programs [8]. Furthermore, even if bounded models are not faithful (i.e., may not be

extendible to models of the full grounding), they might be convenient e.g. to provide strings, arithmetic, recursive data structures like lists, trees etc, or action sequences of bounded length resp. depth. The point is that the bound does not have to be "coded" in the program (like `maxint` in `DLV` to bound the integer range), but can be provided via termination criteria in the grounding, which gives greater flexibility.

Organization. After the preliminaries we proceed as follows.

- We introduce domain-specific existential quantification in HEX-programs and considers its realization (Section 3). To this end, we introduce a generalized grounding algorithm with *hooks* for termination criteria, which enables bounded grounding. Notably, its output for de-safe programs (using trivial criteria) is equivalent to the original program, i.e., it has the same answer sets.

We illustrate some advantages of our approach, which cannot easily be integrated into direct implementations of existential quantifiers.

- As an example, we consider the realization of *null values* (which are customary in databases) as a domain-specific existential quantifier, leading to $\text{HEX}^{\exists}$-programs (Section 4); they include existential rules of form $\forall\mathbf{X}\forall\mathbf{Z}\exists\mathbf{Y}.\psi(\mathbf{Z},\mathbf{Y}) \leftarrow \phi(\mathbf{X},\mathbf{Y},\mathbf{Z})$ (also known as tuple-generating dependencies), where $\psi(\mathbf{Z},\mathbf{Y})$ is an atom[1] and $\phi(\mathbf{X},\mathbf{Y},\mathbf{Z})$ is a conjunction of atoms. Our framework can be thus exploited for bounded grounding, and in combination with a HEX-solver for bounded model generation of such programs.
- As an immediate application, we consider query answering over existential rules (Section 5), which reduces for prominent settings to query answering over a universal model. Under de-safety, a finite such model can be generated using our framework; this allows to cover a range of acyclic existential rules, including the very general notion of model-faithful acyclicity [14]. For non-de safe programs, a bounded universal model may be generated under suitable conditions; we illustrate this for Shy-programs - a class of programs with existential rules for which query answering is decidable, cf. [17].
- Furthermore, we show how terms with function symbols can be processed using an encoding as a HEX-program (Section 6). To this end, we use dedicated external atoms to construct and decompose functional terms; bounded grounding enables us here to elegantly restrict the term depth, which is useful for applications such as reasoning with actions in situation calculus under bounded horizon, or reasoning from FDNC programs.

We conclude with a discussion and an outlook on future work in Section 7. Our prototype system is available at http://www.kr.tuwien.ac.at/research/systems/dlvhex.

2 Preliminaries

HEX-Program Syntax. HEX-programs generalize (disjunctive) logic programs under the answer set semantics [13] with external source access; for details and background see [7]. They are built over mutually disjoint sets $\mathcal{P}$, $\mathcal{X}$, $\mathcal{C}$, and $\mathcal{V}$ of ordinary predicates, external predicates, constants, and variables, respectively. Every $p \in \mathcal{P}$ has an arity $ar(p) \geq 0$, and every external predicate $\&g \in \mathcal{X}$ has an input arity $ar_i(\&g) \geq 0$ of input parameters and an output arity $ar_o(\&g) \geq 0$ of output arguments.

[1] In general, $\psi(\mathbf{Z},\mathbf{Y})$ might be a conjunction of atoms but this may be normalized.

An *external atom* is of the form $\&g[\mathbf{X}](\mathbf{Y})$, where $\&g \in \mathcal{X}$, $\mathbf{X} = X_1, \ldots, X_\ell$ ($\ell = ar_i(\&g)$) are input parameters with $X_i \in \mathcal{P} \cup \mathcal{C} \cup \mathcal{V}$ for all $1 \leq i \leq \ell$, and $\mathbf{Y} = Y_1, \ldots, Y_m$ ($m = ar_o(\&g)$) are output terms with $Y_i \in \mathcal{C} \cup \mathcal{V}$ for all $1 \leq i \leq m$; we use lower case $\mathbf{x} = x_1, \ldots, x_\ell$ resp. $\mathbf{y} = y_1, \ldots, y_m$ if $\mathbf{X}$ resp. $\mathbf{Y}$ is variable-free. We assume the input parameters of $\&g$ are typed by $type(\&g, i) \in \{\texttt{const}, \texttt{pred}\}$ for $1 \leq i \leq ar_i(\&g)$, and that $X_i \in \mathcal{P}$ if $type(\&g, i) = \texttt{pred}$ and $X_i \in \mathcal{C} \cup \mathcal{V}$ otherwise.

A HEX-program consists of rules

$$a_1 \vee \cdots \vee a_k \leftarrow b_1, \ldots, b_m, \text{not } b_{m+1}, \ldots, \text{not } b_n \ , \tag{2}$$

where each a_i is an (ordinary) atom $p(X_1, \ldots, X_\ell)$ with $X_i \in \mathcal{C} \cup \mathcal{V}$ for all $1 \leq i \leq \ell$, each b_j is either an ordinary atom or an external atom, and $k + n > 0$.

The *head* of a rule r is $H(r) = \{a_1, \ldots, a_n\}$ and the *body* is $B(r) = \{b_1, \ldots, b_m, \text{not } b_{m+1}, \ldots, \text{not } b_n\}$. We call b or $\text{not } b$ in a rule body a *default literal*; $B^+(r) = \{b_1, \ldots, b_m\}$ is the *positive body*, $B^-(r) = \{b_{m+1}, \ldots, b_n\}$ is the *negative body*. For a program Π (rule r), let $A(\Pi)$ ($A(r)$) be the set of all ordinary atoms and $EA(\Pi)$ ($EA(r)$) be the set of all external atoms occurring in Π (in r).

HEX-Program Semantics. Following [11], a *(signed) ground literal* is a positive or a negative formula $\mathbf{T}a$ resp. $\mathbf{F}a$, where a is a ground ordinary atom. For a ground literal $\sigma = \mathbf{T}a$ or $\sigma = \mathbf{F}a$, let $\overline{\sigma}$ denote its opposite, i.e., $\overline{\mathbf{T}a} = \mathbf{F}a$ and $\overline{\mathbf{F}a} = \mathbf{T}a$. An *assignment* $\mathbf{A}$ is a consistent set of literals $\mathbf{T}a$ or $\mathbf{F}a$, where $\mathbf{T}a$ expresses that a is true and $\mathbf{F}a$ that a is false. We also identify a complete assignment $\mathbf{A}$ with its true atoms, i.e., $\mathbf{T}(\mathbf{A}) = \{a \mid \mathbf{T}a \in \mathbf{A}\}$. The semantics of a ground external atom $\&g[\mathbf{x}](\mathbf{y})$ wrt. a complete assignment $\mathbf{A}$ is given by a $1{+}k{+}l$-ary Boolean-valued *oracle function*, $f_{\&g}(\mathbf{A}, \mathbf{x}, \mathbf{y})$. Parameter x_i with $type(\&g, i) = \texttt{pred}$ is *monotonic* (*antimonotonic*), if $f_{\&g}(\mathbf{A}, \mathbf{x}, \mathbf{y}) \leq f_{\&g}(\mathbf{A}', \mathbf{x}, \mathbf{y})$ ($f_{\&g}(\mathbf{A}', \mathbf{x}, \mathbf{y}) \leq f_{\&g}(\mathbf{A}, \mathbf{x}, \mathbf{y})$) whenever $\mathbf{A}'$ increases $\mathbf{A}$ only by literals $\mathbf{T}a$, where a has predicate x_i; otherwise, x_i is called *nonmonotonic*.

Non-ground programs are handled by grounding as usual. The set of constants appearing in a program Π is denoted C_Π. The *grounding* $grnd_C(r)$ of a rule r wrt. $C \subseteq \mathcal{C}$ is the set of all rules $\{\sigma(r) \mid \sigma : \mathcal{V} \mapsto C\}$, where σ is a *grounding substitution*, and $\sigma(r)$ results if each variable X in r is replaced by $\sigma(X)$. The *grounding of a program* Π wrt. C is defined as $grnd_C(\Pi) = \bigcup_{r \in \Pi} grnd_C(r)$.

Satisfaction of rules and programs [13] is extended to HEX-rules r and programs Π in the obvious way. The *FLP-reduct* is defined as $fgrnd_{\mathcal{C}}(\Pi)^{\mathbf{A}} = \{r \in grnd_{\mathcal{C}}(\Pi) \mid \mathbf{A} \models B(r)\}$. An *answer set* of a program Π is a model of $fgrnd_{\mathcal{C}}(\Pi)^{\mathbf{A}}$ that is subset-minimal in its positive part [9]. We denote by $\mathcal{AS}(\Pi)$ the set of all answer sets of Π.

Take as an example the program $\Pi = \{str(N) \leftarrow str(L), \&head[L](N); str(N) \leftarrow str(L), \&tail[L](N)\}$, where $\&head[L](N)$ ($\&tail[L](N)$) is true iff string N is string L without the last (first) character. For $str(x)$, Π computes all substrings of string x.

Safety. In general, $\mathcal{C}$ has constants that do not occur in Π and can even be infinite (e.g., the set of all strings). Safety criteria guarantee that a finite portion $\Pi' \subseteq grnd_{\mathcal{C}}(\Pi)$ (also called *finite grounding* of Π; usually by restricting to a finite $C \subseteq \mathcal{C}$) has the same answer sets as Π. Ordinary safety requires that every variable in a rule r occurs either in an ordinary atom in $B^+(r)$, or in the output list $\mathbf{Y}$ of an external atom $\&g[\mathbf{X}](\mathbf{Y})$ in $B^+(r)$ where all variables in $\mathbf{X}$ are safe. However, this notion is not sufficient.

Example 1. Let $\Pi = \{s(a); t(Y) \leftarrow s(X), \&concat[X, a](Y); s(X) \leftarrow t(X), d(X)\}$, where $\&concat[X, a](Y)$ is true iff Y is the string concatenation of X and a. Then Π is safe but $\&concat[X, a](Y)$ can introduce infinitely many values.

The general notion of *(liberal) domain-expansion safety (de-safety)* subsumes a range of other well-known notions and can be easily extended in a modular fashion [6]. It is based on *term bounding functions* (TBFs), which intuitively declare terms in rules as *bounded*, if there are only finitely many substitutions for this term in a *canonical grounding* $CG(\Pi)$ of Π.[2] The latter is infinite in general but finite for de-safe programs.

More specifically we consider *attributes* and *ranges*. For an ordinary predicate $p \in \mathcal{P}$, let $p{\restriction}i$ be the *i-th attribute of p* for all $1 \leq i \leq ar(p)$. For an external predicate $\&g \in \mathcal{X}$ with input list $\mathbf{X}$ in rule r, let $\&g[\mathbf{X}]_r{\restriction}_T i$ with $T \in \{\text{I}, \text{O}\}$ be the *i-th input resp. output attribute of $\&g[\mathbf{X}]$ in r* for all $1 \leq i \leq ar_T(\&g)$. For a ground program Π, an attribute *range* is, intuitively, the set of ground terms which occur in the position of the attribute. Formally, for an attribute $p{\restriction}i$ we have $range(p{\restriction}i, \Pi) = \{t_i \mid p(t_1, \ldots, t_{ar(p)}) \in A(\Pi)\}$; for $\&g[\mathbf{X}]_r{\restriction}_T i$ it is $range(\&g[\mathbf{X}]_r{\restriction}_T i, \Pi) = \{x_i^T \mid \&g[\mathbf{x}^{\text{I}}](\mathbf{x}^{\text{O}}) \in EA(\Pi)\}$, where $\mathbf{x}^s = x_1^s, \ldots, x_{ar_s(\&g)}^s$. Now term bounding functions are introduced as follows:

Definition 1 (Term Bounding Function (TBF)). *A TBF $b(\Pi, r, S, B)$ maps a program Π, a rule $r \in \Pi$, a set S of already safe attributes, and a set B of already bounded terms in r to an enlarged set $b(\Pi, r, S, B) \supseteq B$ of bounded terms, s.t. every $t \in b(\Pi, r, S, B)$ has finitely many substitutions in $CG(\Pi)$ if (i) the attributes S have a finite range in $CG(\Pi)$ and (ii) each term in $terms(r) \cap B$ has finitely many substitutions in $CG(\Pi)$.*

Liberal domain-expansion safety of programs is then parameterized with a term bounding function, such that concrete syntactic and/or semantic properties can be plugged in; concrete term bounding functions are described in [6]. The concept is defined in terms of domain-expansion safe attributes $S_\infty(\Pi)$, which are stepwise identified as $S_n(\Pi)$ in mutual recursion with bounded terms $B_n(r, \Pi, b)$ of rules r in Π.

Definition 2 ((Liberal) Domain-expansion Safety). *Given a TBF b, the set of* bounded terms $B_n(r, \Pi, b)$ *in step $n \geq 1$ in a rule $r \in \Pi$ is $B_n(r, \Pi, b) = \bigcup_{j \geq 0} B_{n,j}(r, \Pi, b)$ where $B_{n,0}(r, \Pi, b) = \emptyset$ and for $j \geq 0$, $B_{n,j+1}(r, \Pi, b) = b(\Pi, r, S_{n-1}(\Pi), B_{n,j})$.*

The set of domain-expansion safe attributes $S_\infty(\Pi) = \bigcup_{i \geq 0} S_i(\Pi)$ *of a program Π is iteratively constructed with $S_0(\Pi) = \emptyset$ and for $n \geq 0$:*

- *$p{\restriction}i \in S_{n+1}(\Pi)$ if for each $r \in \Pi$ and atom $p(t_1, \ldots, t_{ar(p)}) \in H(r)$, it holds that $t_i \in B_{n+1}(r, \Pi, b)$, i.e., t_i is* bounded*;*
- *$\&g[\mathbf{X}]_r{\restriction}_{\text{I}} i \in S_{n+1}(\Pi)$ if each $\mathbf{X}_i$ is a* bounded *variable, or $\mathbf{X}_i$ is a predicate input parameter p and $p{\restriction}1, \ldots, p{\restriction}ar(p) \in S_n(\Pi)$;*
- *$\&g[\mathbf{X}]_r{\restriction}_{\text{O}} i \in S_{n+1}(\Pi)$ if and only if r contains an external atom $\&g[\mathbf{X}](\mathbf{Y})$ such that $\mathbf{Y}_i$ is bounded, or $\&g[\mathbf{X}]_r{\restriction}_{\text{I}} 1, \ldots, \&g[\mathbf{X}]_r{\restriction}_{\text{I}} ar_{\text{I}}(\&g) \in S_n(\Pi)$.*

A program Π is (liberally) de-safe*, if it is safe and all its attributes are de-safe.*

Example 2. The program Π from Example 1 is liberally de-safe using the TBF b_{synsem} from [6] as the generation of infinitely many values is prevented by $d(X)$ in the last rule.

Every de-safe HEX-program has a finite grounding that preserves all answer sets [6].

[2] $CG(\Pi)$ is the least fixed point $G_\Pi^\infty(\emptyset)$ of a monotone operator $G_\Pi(\Pi') = \bigcup_{r \in \Pi} \{r\theta \mid r\theta \in grnd_{\mathcal{C}}(r), \exists \mathbf{A} \subseteq \mathcal{A}(\Pi'), \mathbf{A} \not\models \bot, \mathbf{A} \models B^+(r\theta)\}$ on programs Π' [6].

3 HEX-Programs with Existential Quantification

In this section, we consider HEX-programs with *domain-specific existential quantifiers*. This term refers to the introduction of new values in rule bodies which are propagated to the head such that they may appear in the answer sets of a program. Logical existential quantification is a special case of our approach (used in Section 4 to illustrate a specific instance), where just the existence but not the structure of values is of interest. Instead, in our work also the structure of introduced values may be relevant and can be controlled by external atoms.

Instantiating, i.e., applying, our approach builds on an extension of the grounding algorithm for HEX-programs in [5] by additional *hooks*. They support the insertion of application-specific termination criteria, and thus can be exploited for computing a finite subset of the grounding in case of non-de-safe HEX-programs. The latter may be sufficient to consider a certain reasoning task, e.g., for bounded model building. For instance, we discuss *queries* over (positive) programs with (logical) existential quantifiers in Section 5, which can be answered by computing a finite part of a canonical model.

HEX-Program Grounding. For introducing our *bounded grounding algorithm* BGroundHEX, we make use of so-called input auxiliary rules. We say that an external atom $\&g[\mathbf{Y}](\mathbf{X})$ *joins* an atom b, if some variable from $\mathbf{Y}$ occurs in b, where in case b is an external atom the occurrence is in the output list of b.

Definition 3 (Input Auxiliary Rule). *Let Π be a* HEX*-program. Then for each external atom $\&g[\mathbf{Y}](\mathbf{X})$ occurring in rule $r \in \Pi$, a rule $r_{inp}^{\&g[\mathbf{Y}](\mathbf{X})}$ is composed as follows:*

- *The head is $H(r_{inp}^{\&g[\mathbf{Y}](\mathbf{X})}) = \{g_{inp}(\mathbf{Y})\}$, where g_{inp} is a fresh predicate; and*
- *The body $B(r_{inp}^{\&g[\mathbf{Y}](\mathbf{X})})$ contains all $b \in B^+(r) \backslash \{\&g[\mathbf{Y}](\mathbf{X})\}$ which join $\&g[\mathbf{Y}](\mathbf{X})$.*

Intuitively, input auxiliary rules are used to derive all ground input tuples $\mathbf{y}$, under which the external atom needs to be evaluated.

Our grounding approach is based on a grounder for ordinary ASP programs. Compared to the naive grounding $grnd_C(\Pi)$, we allow the ASP grounder GroundASP to eliminate rules if their body is always false, and ordinary body literals from the grounding that are always true, as long as this does not change the answer sets. More formally, a rule r' is an *o-strengthening* (ordinary-strengthening) of a rule r, if $H(r') = H(r)$, $B(r') \subseteq B(r)$ and $B(r) \setminus B(r')$ contains only ordinary literals, i.e., no external atom replacements.

Definition 4. *An algorithm GroundASP that takes as input a program Π and outputs a ground program Π' is a* faithful ASP grounder *for a safe program Π, if:*

- *$\mathcal{AS}(\Pi') = \mathcal{AS}(grnd_{C_\Pi}(\Pi))$;*
- *Π' consists of o-strengthenings of rules in $grnd_{C_\Pi}(\Pi)$;*
- *if $r \in grnd_{C_\Pi}(\Pi)$ has no o-strengthening in Π', then every answer set of $grnd_{C_\Pi}(\Pi)$ falsifies some ordinary literal in $B(r)$; and*
- *if $r \in grnd_{C_\Pi}(\Pi)$ has some o-strengthening $r' \in \Pi'$, then every answer set of $grnd_{C_\Pi}(\Pi)$ satisfies $B(r) \setminus B(r')$.*

Intuitively, the bounded grounding algorithm BGroundHEX can be explained as follows. Program Π is the non-ground input program. Program Π_p is the non-ground

Algorithm BGroundHEX

Input: A HEX-program Π
Output: A ground HEX-program Π_g

(a) $\Pi_p = \Pi \cup \{r_{inp}^{\&g[\mathbf{Y}](\mathbf{X})} \mid \&g[\mathbf{Y}](\mathbf{X}) \text{ in } r \in \Pi\}$
Replace all external atoms $\&g[\mathbf{Y}](\mathbf{X})$ in all rules r in Π_p by $e_{r,\&g[\mathbf{Y}]}(\mathbf{X})$
$i \leftarrow 0$
(b) **while** *Repeat()* **do**
  $i \leftarrow i + 1$ `// Remember already processed input tuples at iteration` i
(c) Set $NewInputTuples \leftarrow \emptyset$ and $PIT_i \leftarrow \emptyset$
  repeat
    $\Pi_{pg} \leftarrow$ GroundASP(Π_p) `// partial grounding`
(d) **for** $\&g[\mathbf{Y}](\mathbf{X})$ *in a rule* $r \in \Pi$ **do** `// evaluate all external atoms`
(e) `// do this under all relevant assignments`
      $\mathbf{A}_{ma} = \{\mathbf{T}p(\mathbf{c}) \mid a(\mathbf{c}) \in A(\Pi_{pg}), p \in \mathbf{Y}_m\} \cup \{\mathbf{F}p(\mathbf{c}) \mid a(\mathbf{c}) \in A(\Pi_{pg}), p \in \mathbf{Y}_a\}$
      for $\mathbf{A}_{nm} \subseteq \{\mathbf{T}p(\mathbf{c}), \mathbf{F}p(\mathbf{c}) \mid p(\mathbf{c}) \in A(\Pi_{pg}), p \in \mathbf{Y}_n\}$ *s.t.* $\nexists a : \mathbf{T}a, \mathbf{F}a \in \mathbf{A}_{nm}$ **do**
        $\mathbf{A} = (\mathbf{A}_{ma} \cup \mathbf{A}_{nm} \cup \{\mathbf{T}a \mid a \leftarrow\in \Pi_{pg}\}) \setminus \{\mathbf{F}a \mid a \leftarrow\in \Pi_{pg}\}$
(f) **for** $\mathbf{y} \in \{\mathbf{c} \mid r_{inp}^{\&g[\mathbf{Y}](\mathbf{X})}(\mathbf{c}) \in A(\Pi_{pg})$ *s.t.* *Evaluate*$(r_{inp}^{\&g[\mathbf{Y}](\mathbf{X})}(\mathbf{c})) = true$ **do**
(g) `// add ground guessing rules and remember` $\mathbf{y}$`-evaluation`
          $\Pi_p \leftarrow \Pi_p \cup \{e_{r,\&g[\mathbf{y}]}(\mathbf{x}) \vee ne_{r,\&g[\mathbf{y}]}(\mathbf{x}) \leftarrow \mid f_{\&g}(\mathbf{A}, \mathbf{y}, \mathbf{x}) = 1\}$
          $NewInputTuples \leftarrow NewInputTuples \cup \{r_{inp}^{\&g[\mathbf{Y}](\mathbf{X})}(\mathbf{y})\}$
    $PIT_i \leftarrow PIT_i \cup NewInputTuples$
  until Π_{pg} *did not change*
(h) Remove input auxiliary rules and external atom guessing rules from Π_{pg}
Replace all $e_{\&g[\mathbf{y}]}(\mathbf{x})$ in Π_{pg} by $\&g[\mathbf{y}](\mathbf{x})$
return Π_{pg}

ordinary ASP *prototype program*, which is an iteratively updated variant of Π enriched with additional rules. In each step, the *preliminary ground program* Π_{pg} is produced by grounding Π_p using a standard ASP grounding algorithm. Program Π_{pg} is intended to converge against a fixpoint, i.e., a final *ground* HEX*-program* Π_g. For this purpose, the loop at (b) and the abortion check at (f) introduce two *hooks* (Repeat and Evaluate) which allow for realizing application-specific termination criteria. They need to be substituted by concrete program fragments depending on the reasoning task; for now we assume that the loop at (f) runs exactly once and the check at (f) is always true (which is sound and complete for model computation of de-safe programs, cf. Proposition 1).

The algorithm first introduces input auxiliary rules $r_{inp}^{\&g[\mathbf{Y}](\mathbf{X})}$ for every external atom $\&g[\mathbf{Y}](\mathbf{X})$ in a rule r in Π in Part (a). Then, all external atoms $\&g[\mathbf{Y}](\mathbf{X})$ in all rules r in Π_p are replaced by ordinary *replacement atoms* $e_{r,\&g[\mathbf{Y}]}(\mathbf{X})$. This allows the algorithm to use an ordinary ASP grounder GroundASP in the main loop at (b). After the grounding step, it is checked whether the grounding contains all relevant constants. For this, the algorithm checks, for all external atoms (d) and all relevant input interpretations (e), potential output tuples at (f), if they contain any new value that was not yet respected in the grounding. (Note that, $\mathbf{Y}_m, \mathbf{Y}_a, \mathbf{Y}_n$ denote the sets of *monotonic*, *antimonotonic*, and *nonmonotonic* predicate input parameters in $\mathbf{Y}$, respectively.) It adds the relevant constants in form of guessing rules at (g) to Π_p (this may also be expressed by unstratified negation). Then the main loop starts over again. Eventually, the algorithm is intended to find a program respecting all relevant constants. Then at (h), auxiliary input rules are removed and replacement atoms are translated to external atoms.

Let us illustrate the grounding algorithm with the following example.

Example 3. Let Π be the following program:

$$f : d(a).\ d(b).\ d(c). \quad \begin{aligned} r_1 &: s(Y) \leftarrow d(X), \&diff[d,n](Y), d(Y). \\ r_2 &: n(Y) \leftarrow d(X), \&diff[d,s](Y), d(Y). \\ r_3 &: c(Z) \leftarrow \&count[s](Z). \end{aligned}$$

Here, $\&diff[s_1, s_2](x)$ is true for all elements x, which are in the extension of s_1 but not in that of s_2, and $\&count[s](i)$ is true for the integer i corresponding to the number of elements in the extension of s. The program first partitions the domain (extension of d) into two sets (extensions of s and n) and then computes the size of s. Program Π_p at the beginning of the first iteration is as follows, where $e_1(Y)$, $e_2(Y)$ and $e_3(Z)$ are shorthands for $e_{r_1,\&diff[d,n]}(Y)$, $e_{r_2,\&diff[d,s]}(Y)$, and $e_{r_3,\&count[s]}(Z)$, respectively.

$$f : d(a).\ d(b).\ d(c). \quad \begin{aligned} r_1 &: s(Y) \leftarrow d(X), e_1(Y), d(Y). \\ r_2 &: n(Y) \leftarrow d(X), e_2(Y), d(Y). \\ r_3 &: c(Z) \leftarrow e_3(Z). \end{aligned}$$

Program Π_{pg} contains no instances of r_1, r_2 and r_3 because the optimizer recognizes that $e_1(Y)$, $e_2(Y)$ and $e_3(Z)$ occur in no rule head and no ground instance can be true in any answer set. Then the algorithm moves to the checking phase. It evaluates the external atoms in r_1 and r_2 under $\mathbf{A} = \{d(a), d(b), d(c)\}$ (note that $\&diff[s_1, s_2](x)$ is monotonic in s_1 and antimonotonic in s_2) and adds the rules $\{e_i(Z) \vee ne_i(Z) \leftarrow \mid\ Z \in \{a, b, c\}, i \in \{1, 2\}\}$ to Π_p. Then it evaluates $\&count[s](Z)$ under all $\mathbf{A} \subseteq \{s(a), s(b), s(c)\}$ because it is nonmonotonic in s, and adds the rules $\{e_3(Z) \vee ne_3(Z) \leftarrow \mid\ Z \in \{0, 1, 2, 3\}\}$. It terminates after the second iteration. □

The main difference to the algorithm from [5] is the addition of the two hooks at (c) (Repeat) and at (f) (Evaluate), that need to be defined for a concrete instance of the algorithm (which we do in the following). We assume that the hooks are substituted by code fragments with access to all local variables. Moreover, the set PIT_i contains the input atoms for which the corresponding external atoms have been evaluated in iteration i. Evaluate decides for a given input atom $r_{inp}^{\&g[\mathbf{Y}](\mathbf{X})}(\mathbf{c})$ if the corresponding external atom shall be evaluated under $\mathbf{c}$. This allows for abortion of the grounding even if it is incomplete, which can be exploited for reasoning tasks over programs with infinite groundings where a finite subset of the grounding is sufficient. The second hook Repeat allows for repeating the core algorithm multiple times such that Evaluate can distinguish between input tuples processed in different iterations. Naturally, soundness and completeness of the algorithm cannot be shown in general, but depends on concrete instances for (c) and (f) which in turn may vary for different reasoning tasks.

Domain-specific Existential Quantification in HEX-Programs. We can realize domain-specific existential quantification naturally in HEX-programs by appropriate external atoms that introduce new values to the program. The realization exploits *value invention* as supported by HEX-programs, i.e., external atoms which return constants that do not show up in the input program. Realizing existentials by external atoms also allows to use constants different from Skolem terms, i.e., datatypes with a specific semantics. The values introduced may depend on input parameters passed to the external atom.

Example 4. Consider the following rule:

$$iban(B, I) \leftarrow country(B, C), bank(B, N), \&iban[C, B, N](I).$$

Suppose $bank(b, n)$ models financial institutions b with their associated national number n, and $country(b, c)$ holds for an institution b and its home country c. Then one can use $\&iban[C, B, N](I)$ to generate an IBAN (*International Bank Account Number*) from the country, the bank name and account number.

Here, the structure of the introduced value is relevant, but an algorithm which computes it can be hidden from the user. The introduction of new values may also be subject to additional conditions which cannot easily be expressed in the program.

Example 5. Consider the following rule:

$$lifetime(M, L) \leftarrow machine(M, C), \&lifetime[M, C](L).$$

It expresses that each purchased machine m with cost c ($machine(m, c)$) higher than a given limit has assigned an expected lifetime l ($lifetime(m, l)$) used for fiscal purposes, whereas purchases below that limit are fully tax deductible in the year of acquirement. Then testing for exceedance of the limit might involve real numbers and cannot easily be done in the logic program. However, the external atom can easily be extended in such a way that a value is only introduced if this side constraint holds.

Counting quantifiers may be realized in this way, i.e., expressing that there exist *exactly* k or *at least* k elements, which is used e.g. in description logics.While a direct implementation of existentials requires changes in the reasoner, a simulation using external atoms is easily extensible.

4 HEX$^{\exists}$-Programs

We now realize the logical existential quantifier as a specific instance of our approach, which can also be written in the usual syntax; a rewriting then simulates it by using external atoms which return dedicated *null values* to represent a representative for the unnamed values introduced by existential quantifiers. We start by introducing a language for HEX-programs with logical existential quantifiers, called HEX$^{\exists}$*-programs*.

A HEX$^{\exists}$*-program* is a finite set of rules of form

$$\forall \mathbf{X} \exists \mathbf{Y} : p(\mathbf{X}', \mathbf{Y}) \leftarrow \mathbf{conj}[\mathbf{X}], \tag{3}$$

where $\mathbf{X}$ and $\mathbf{Y}$ are disjoint sets of variables, $\mathbf{X}' \subseteq \mathbf{X}$, $p(\mathbf{X}', \mathbf{Y})$ is an atom, and $\mathbf{conj}[\mathbf{X}]$ is a conjunction of default literals or default external literals containing all and only the variables $\mathbf{X}$; without confusion, we also omit $\forall \mathbf{X}$.

Intuitively speaking, whenever $\mathbf{conj}[\mathbf{X}]$ holds for some vector of constants $\mathbf{X}$, then there should exist a vector $\mathbf{Y}$ of (unnamed) individuals such that $p(\mathbf{X}', \mathbf{Y})$ holds. Existential quantifiers are simulated by using *new* null values which represent the introduced unnamed individuals. Formally, we assume that $\mathcal{N} \subseteq \mathcal{C}$ is a set of dedicated null values, denoted by ω_i with $i \in \mathbb{N}$, which do not appear in the program.

We transform HEX$^{\exists}$-programs to HEX-programs as follows. For a HEX$^{\exists}$-program Π, let $T_{\exists}(\Pi)$ be the HEX-program with each rule r of form (3) replaced by

$$p(\mathbf{X}', \mathbf{Y}) \leftarrow \mathbf{conj}[\mathbf{X}], \&exists^{|\mathbf{X}'|,|\mathbf{Y}|}[r, \mathbf{X}'](\mathbf{Y}),$$

where $f_{\&exists^{n,m}}(\mathbf{A}, r, \mathbf{x}, \mathbf{y}) = 1$ iff $\mathbf{y} = \omega_1, \ldots, \omega_m$ is a vector of *fresh and unique null values* for $r, \mathbf{x}$, and $f_{\&exists^{n,m}}(\mathbf{A}, r, \mathbf{x}, \mathbf{y}) = 0$ otherwise.

Each existential quantifier is replaced by an external atom $\&exists^{|\mathbf{X}'|,|\mathbf{Y}|}[r, \mathbf{X}'](\mathbf{Y})$ of appropriate input and output arity which exploits value invention for simulating the logical existential quantifier similar to the *chase* algorithm.

We call a HEX$^\exists$-program Π liberally de-safe iff $T_\exists(\Pi)$ is liberally de-safe. Various notions of cyclicity have been introduced, e.g., in [14]; here we use the one from [6].

Example 6. The following set of rules is a HEX$^\exists$-program Π:

$$\begin{array}{c} employee(john). \quad employee(joe). \\ r_1 : \exists Y : office(X,Y) \leftarrow employee(X). \quad r_2 : room(Y) \leftarrow office(X,Y) \end{array}$$

Then $T_\exists(\Pi)$ is the following de-safe program:

$$\begin{array}{rc} & employee(john). \quad employee(joe). \\ r_1' : & office(X,Y) \leftarrow employee(X), \&exists^{1,1}[r_1, X](Y). \\ r_2 : & room(Y) \leftarrow office(X,Y) \end{array}$$

Intuitively, each employee X has some unnamed office Y of X, which is a room. The unique answer set of $T_\exists(\Pi)$ is $\{employee(john), employee(joe), office(john, \omega_1), office(joe, \omega_2), room(\omega_1), room(\omega_2)\}$.

For grounding de-safe programs, we simply let Repeat test for $i < 1$ and Evaluate return $true$. Explicit model computation is in general infeasible for non-de-safe programs. However, the resulting algorithm GroundDESafeHEX always terminates for de-safe programs. For non-de-safe programs, we can support bounded model generation by other hook instantiations. This is exploited e.g. for query answering over cyclic programs (described next). One can show that the algorithm computes all models of the program.

Proposition 1. *For de-safe programs Π, $\mathcal{AS}(\mathsf{GroundDESafeHEX}(\Pi)) \equiv^{pos} \mathcal{AS}(\Pi)$, where $\equiv^{pos}$ denotes equivalence of the answer sets on positive atoms.*

5 Query Answering over Positive HEX$^\exists$-Programs

The basic idea for query answering over programs with possibly infinite models is to compute a ground program with a single answer set that can be used for answering the query. Positive programs with existential variables are essentially grounded by simulating the *parsimonious chase procedure* from [17], which uses null values for each existential quantification. However, for termination of BGroundHEX we need to provide specific instances of the hooks in the grounding algorithm.

We start by restricting the discussion to a fragment of HEX$^\exists$-programs, called $Datalog^\exists$-programs [17]. A $Datalog^\exists$-program is a HEX$^\exists$-program where every rule body $\mathbf{conj}[\mathbf{X}]$ consists of positive ordinary atoms. Thus compared to HEX$^\exists$-programs, default negation and external atoms are excluded.

As an example, the following set of rules is a $Datalog^\exists$-program:

$$\begin{array}{c} person(john). \quad person(joe). \\ r_1 : \exists Y : father(X,Y) \leftarrow person(X). \quad r_2 : person(Y) \leftarrow father(X,Y). \end{array} \tag{4}$$

Next, we recall *homomorphisms* as used for defining $Datalog^\exists$-semantics and query answering over $Datalog^\exists$-programs. A *homomorphism* is a mapping $h\colon \mathcal{N} \cup \mathcal{V} \to \mathcal{C} \cup \mathcal{V}$. For a homomorphism h, let $h|_S$ be its restriction to $S \subseteq \mathcal{N} \cup \mathcal{V}$, i.e., $h|_S(X) = h(X)$ if $X \in S$ and is undefined otherwise. For any atom a, let $h(a)$ be the atom where each

variable and null value V in a is replaced by $h(V)$; this is likewise extended to $h(S)$ for sets S of atoms and/or vectors of terms. A homomorphism h is a *substitution*, if $h(N) = N$ for all $N \in \mathcal{N}$. An atom a is *homomorphic* (*substitutive*) to atom b, if some homomorphism (substitution) h exists such that $h(a) = b$. An isomorphism between two atoms a and b is a bijective homomorphism h s.t. $h(a) = b$ and $h^{-1}(b) = a$.

A set M of atoms is a model of a $Datalog^{\exists}$-program Π, denoted $M \models \Pi$, if $h(B(r)) \subseteq M$ for some substitution h and $r \in \Pi$ of form (3) implies that $h|_{\mathbf{X}}(H(r))$ is substitutive to some atom in M; the set of all models of Π is denoted by $mods(\Pi)$.

Next, we can introduce queries over $Datalog^{\exists}$-programs. A *conjunctive query* (CQ) q is an expression of form $\exists \mathbf{Y} : \leftarrow \mathbf{conj}[\mathbf{X} \cup \mathbf{Y}]$, where $\mathbf{Y}$ and $\mathbf{X}$ (the free variables) are disjoint sets of variables and $\mathbf{conj}[\mathbf{X} \cup \mathbf{Y}]$ is a conjunction of ordinary atoms containing all and only the variables $\mathbf{X} \cup \mathbf{Y}$.

The answer of a CQ q with free variables $\mathbf{X}$ wrt. a model M is defined as follows:

$$ans(q, M) = \{h|_{\mathbf{X}} \mid h \text{ is a substitution and } h(\mathbf{conj}[\mathbf{X} \cup \mathbf{Y}]) \subseteq M\}.$$

Intuitively, this is the set of assignments to the free variables such that the query holds wrt. the model. The answer of a CQ q wrt. a program Π is then defined as the set $ans(q, \Pi) = \bigcap_{M \in mods(\Pi)} ans(q, M)$.

Query answering can be carried out over some *universal model* U of the program that is embeddable into each of its models by applying a suitable homomorphism. Formally, a model U of a program Π is called *universal* if, for each $M \in mods(\Pi)$, there is a homomorphism h s.t. $h(U) \subseteq M$. Thus, a universal model may be obtained using null values for unnamed individuals introduced by existential quantifiers. Moreover, it can be used to answer any query according to the following proposition [10]:

Proposition 2 ([10]). *Let U be a universal model of $Datalog^{\exists}$-program Π. Then, for any CQ q, it holds that $h \in ans(q, \Pi)$ iff $h \in ans(q, U)$ and $h : \mathcal{V} \to \mathcal{C} \setminus \mathcal{N}$.*

Intuitively, the set of all answers to q wrt. U which map all variables to non-null constants is exactly the set of answers to q wrt. Π.

Example 7. Let Π be the program consisting of rules (4). The CQ $\exists Y :\leftarrow person(X), father(X, Y)$ asks for all persons who have a father. The model $U = \{person(john), person(joe), father(john, \omega_1), father(joe, \omega_2), person(\omega_1), person(\omega_2), \ldots\}$ is a universal model of Π. Hence, $ans(q, \Pi)$ contains answers $h_1(X) = john$ and $h_2(X) = joe$.

Thus, computing a universal model is a key issue for query answering. A common approach for this step is the chase procedure. Intuitively, it starts from an empty interpretation and iteratively adds the head atoms of all rules with satisfied bodies, where existentially quantified variables are substituted by fresh nulls. However, in general this procedure does not terminate. Thus, a restricted *parsimonious chase procedure* was introduced in [17], which derives less atoms, and which is guaranteed to terminate for the class of *Shy-programs*. Moreover, it was shown that the interpretation computed by the parsimonious chase procedure is, although not a model of the program in general, still sound and complete for query answering and a *bounded model* in our view.

For query answering over $Datalog^{\exists}$-programs we reuse the translation in Section 4.

Example 8. Consider the $Datalog^{\exists}$-program Π and its HEX translation $T_{\exists}(\Pi)$:

Π :

$person(john).$ $person(joe).$

$\exists Y : father(X,Y) \leftarrow person(X).$

$person(Y) \leftarrow father(X,Y).$

$T_\exists(\Pi)$:

$person(john).$ $person(joe).$

$father(X,Y) \leftarrow person(X),$

$\&exists^{1,1}[r_1, X](Y).$

$person(Y) \leftarrow father(X,Y).$

Intuitively, each person X has some unnamed father Y of X which is also a person.

Note that $T_\exists(\Pi)$ is *not* de-safe in general. However, with the hooks in Algorithm BGroundHEX one can still guarantee termination. Let GroundDatalog$^\exists(\Pi, k) =$ BGroundHEX$(T_\exists(\Pi))$ where Repeat tests for $i < k + 1$ where k is the number of existentially quantified variables in the query, and Evaluate$(PIT_i, x) = true$ iff atom x is *not* homomorphic to any $a \in PIT_i$.

The produced program has a single answer set, which essentially coincides with the result of $pChase$ [17] that can be used for query answering. Thus, query answering over Shy-programs is reduced to grounding and solving of a HEX-program.

Proposition 3. *For a Shy-program Π, GroundDatalog$^\exists(\Pi, k)$ has a unique answer set which is sound and complete for answering CQs with up to k existential variables.*

The main difference to $pChase$ in [17] is essentially due to the homomorphism check. Actually, $pChase$ instantiates existential variables in rules with satisfied body to new null values only if the resulting head atom is not homomorphic to an already derived atom. In contrast, our algorithm performs the homomorphism check for the input to $\&exists^{n,m}$ atoms. Thus, homomorphisms are detected when constants are cyclically sent to the external atom. Consequently, our approach may need one iteration more than $pChase$, but allows for a more elegant integration into our algorithm.

Example 9. For the program and query from Example 8, the algorithm computes a program with answer set $\{person(john), person(joe), father(john, \omega_1), father(joe, \omega_2), person(\omega_1), person(\omega_2)\}$. In contrast, $pChase$ would stop already earlier with the interpretation $\{person(john), person(joe), father(john, \omega_1), father(joe, \omega_2)\}$ because $person(\omega_1), person(\omega_2)$ are homomorphic to $person(john), person(joe)$.

More formally, one can show that GroundDatalog$^\exists(\Pi, k)$ yields, for a Shy-program Π, a program with a single answer set that is equivalent to $pChase(\Pi, k + 1)$ in [17]. Lemma 4.9 in [17] implies that the resulting answer set can be used for answering queries with k different existentially quantified variables, which proves Proposition 3.

While $pChase$ intermingles grounding and computing a universal model, our algorithm cleanly separates the two stages; modularized program evaluation by the solver will however also effect such intermingling. We nevertheless expect the more clean separation to be advantagagoues for extending Shy-programs to programs that involve existential quantifiers and other external atoms, which we leave for future work.

6 HEX-Programs with Function Symbols

In this section we show how to process terms with function symbols by a rewriting to de-safe HEX-programs. We will briefly discuss advantages of our approach compared to a direct implementation of function symbols.

We consider HEX-programs, where the arguments X_i for $1 \leq i \leq \ell$ of ordinary atoms $p(X_1, \ldots, X_\ell)$, and the constant input arguments in $\mathbf{X}$ and the output $\mathbf{Y}$ of an external atom $\&g[\mathbf{X}](\mathbf{Y})$ are from a set of *terms* $\mathcal{T}$, that is the least set $\mathcal{T} \supseteq \mathcal{V} \cup \mathcal{C}$ such that $f \in \mathcal{C}$ (constant symbols are also used as function symbols) and $t_1, \ldots, t_n \in \mathcal{T}$ imply $f(t_1, \ldots, t_n) \in \mathcal{T}$.

Following [4], we introduce for every $k \geq 0$ two external predicates $\&comp_k$ and $\&decomp_k$ with $ar_{\text{I}}(\&comp_k) = 1 + k$, $ar_{\text{O}}(\&comp_k) = 1$, $ar_{\text{I}}(\&decomp_k) = 1$, and $ar_{\text{O}}(\&decomp_k) = 1 + k$. We define

$$f_{\&comp_k}(\mathbf{A}, f, X_1, \ldots, X_k, T) = f_{\&decomp_k}(\mathbf{A}, T, f, X_1, \ldots, X_k) = 1,$$

iff $T = f(X_1, \ldots, X_k)$.

Composition and decomposition of function terms can be simulated using these external predicates. Function terms are replaced by new variables and appropriate additional external atoms with predicate $\&comp_k$ or $\&decomp_k$ in rule bodies to compute their values. More formally, we introduce the following rewriting.

For any HEX-program Π with function symbols, let $T_f(\Pi)$ be the HEX-program where each occurrence of a term $t = f(t_1, \ldots, t_n)$ in a rule r such that $B(r) \neq \emptyset$ is recursively replaced by a new variable V, and if V occurs afterwards in $H(r)$ or the input list of an external atom in $B(r)$, we add $\&comp_n[f, t_1, \ldots, t_n](V)$ to $B(r)$; otherwise (i.e., V occurs afterwards in some ordinary body atom or the output list of an external atom), we add $\&decomp_n[V](f, t_1, \ldots, t_n)$ to $B(r)$.

Intuitively, $\&comp_n$ is used to construct a nested term from a function symbol and arguments, which might be nested terms themselves, and $\&decomp_n$ is used to extract the function symbol and the arguments from a nested term. The translation can be optimized wrt. evaluation efficiency, but we disregard this here for space reasons.

Example 10. Consider the HEX-program Π with function symbols and its translation:

$$\begin{array}{lll}
\Pi: & q(z).\ q(y). & T_f(\Pi): q(z).\ q(y). \\
& p(f(f(X))) \leftarrow q(X). & \quad p(V) \leftarrow q(X), \&comp_1[f, X](U), \\
& \quad r(X) \leftarrow p(X). & \qquad \&comp_1[f, U](V). \\
& \quad r(X) \leftarrow r(f(X)). & \quad r(X) \leftarrow p(X). \\
& & \quad r(X) \leftarrow r(V), \&decomp_1[V](f, X).
\end{array}$$

Intuitively, $T_f(\Pi)$ builds $f(f(X))$ for any X on which q holds using two atoms over $\&comp_1$, and it extracts terms X from derived $r(f(X))$ facts using a $\&decomp_1$-atom.

Note that $\&decomp_n$ supports a well-ordering on term depth such that its output has always a strictly smaller depth than its inputs. This is an important property for proving finite groundability of a program by exploiting the TBFs introduced in [6].

Example 11. The program $\Pi = \{q(f(f(a)));\ q(X) \leftarrow q(f(X))\}$ is translated to $T_f(\Pi) = \{q(f(f(a)));\ q(X) \leftarrow q(V), \&decomp_1[V](f, X)\}$. Since $\&decomp_1$ supports a well-ordering, the cycle is *benign* [6], i.e., it cannot introduce infinitely many values because the nesting depth of terms is strictly decreasing with each iteration.

The realization of function symbols via external atoms (which can in fact also be seen as domain-specific existential quantifiers) has the advantage that their processing can be controlled. For instance, the introduction of new nested terms may be restricted by additional conditions which can be integrated in the semantics of the external predicates

$\&comp_k$ and $\&decomp_k$. A concrete example is *data type checking*, i.e., testing whether the arguments of a function term are from a certain domain. In particular, values might also be rejected, e.g., bounded generation up to a maximal term depth is possible. Another example is to compute some of the term arguments automatically from others, e.g., constructing the functional term $num(7, vii)$ from 7, where the second argument is the Roman representation of the first one.

Another advantage is that the use of external atoms for functional term processing allows for exploiting de-safety of HEX-programs to guarantee finiteness of the grounding. An expressive framework for handling domain-expansion safe programs [6] can be reused without the need to enforce safety criteria specific for function terms.

7 Discussion and Conclusion

We presented model computation and query answering over HEX-programs with domain-specific existential quantifiers, based on external atoms and a new grounding algorithm. In contrast to usual handling of existential quantifiers, ours especially allows for an easy integration of extensions such as additional constraints (even of non-logical nature) or data types. This is useful e.g. for model building applications where particular data is needed for existential values, and gives one the possibility to implement domain-restricted quantifiers and introduce null values, as in databases. The new grounding algorithm allows for controlled bounded grounding; this can be exploited for *bounded model generation*, which might be sufficient (or convenient) for applications. Natural candidates are configuration or, at an abstract level, generating finite models of general first-order formulas as in [12], where an incremental computation of finite models is provided by a translation into incremental ASP. There, grounding and solving is interleaved by continously increasing the bound on the number of elements in the domain. (Note that, although not designed for interleaved evaluation, our approach is flexible enough to also mimic exactly this technique with suitable external atoms.) The work in [1] aims at grounding first-order sentences with complex terms such as functions and aggregates for model expansion tasks. Similar to ours, it is based on bottom-up computation, but we do not restrict to finite structures and allow for potentially infinite domains. As a show case, we considered purely logical existentials (null values), for which our grounding algorithm amounts to a simulation of the one in [17] for $Datalog^{\exists}$-programs. However, while [17] combine grounding and model building, our approach clearly separates the two steps; this may ease possible extensions.

We then realized function symbol processing as in [4], by using external atoms to manipulate nested terms. In contrast to other approaches, no extension of the reasoner is needed for this. Furthermore, using external atoms has the advantage that nested terms can be subject to (even non-logical) constraints given by the semantics of the external atoms, and that finiteness of the grounding follows from de-safety of HEX-programs.

In model-building over HEX$^{\exists}$-programs, we can combine existentials with function symbols, as HEX$^{\exists}$-programs can have external atoms in rule bodies. To allow this for query answering over $Datalog^{\exists}$-programs remains to be considered. More generally, also combining existentials with arbitrary external atoms and the use of default-negation in presence of existentials is an interesting issue for future research. This leads to *non-monotonic existential rules*, which most recently are considered in [18] and in [15],

which equips the $Datalog^{\pm}$formalism, which is tailored to ontological knowledge representation and tractable query answering, with well-founded negation. Another line for future research is to allow disjunctive rules and existential quantification as in $Datalog^{\exists,\vee}$ [2], leading to a generalization of the class of Shy-programs. Continuing on the work on guardedness conditions as in open answer set programming [16], $Datalog^{\exists}$, and $Datalog^{\pm}$ should prove useful to find important techniques for constructing more expressive variants of HEX-programs with domain-specific existential quantifiers. The separation of grounding and solving in our approach should be an advantage for such enhancements.

References

1. Aavani, A., Wu, X.N., Ternovska, E., Mitchell, D.: Grounding formulas with complex terms. In: Canadian AI. pp. 13–25. Springer (2011)
2. Alviano, M., Faber, W., Leone, N., Manna, M.: Disjunctive datalog with existential quantifiers: semantics, decidability, and complexity issues. TPLP 12(4-5), 701–718 (2012)
3. Brewka, G., Eiter, T., Truszczyński, M.: Answer set programming at a glance. Commun. ACM 54(12), 92–103 (2011)
4. Calimeri, F., Cozza, S., Ianni, G.: External Sources of Knowledge and Value Invention in Logic Programming. Ann. Math. Artif. Intell. 50(3–4), 333–361 (2007)
5. Eiter, T., Fink, M., Krennwallner, T., Redl, C.: Grounding HEX-Programs with Expanding Domains (2013), Manuscript, submitted
6. Eiter, T., Fink, M., Krennwallner, T., Redl, C.: Liberal safety for answer set programs with external sources. In: AAAI 2013, pp. 267–275. AAAI Press (2013)
7. Eiter, T., Ianni, G., Schindlauer, R., Tompits, H.: A uniform integration of higher-order reasoning and external evaluations in answer-set programming. In: IJCAI. pp. 90–96. (2005)
8. Eiter, T., Simkus, M.: FDNC: Decidable nonmonotonic disjunctive logic programs with function symbols. ACM Trans. Comput. Log. 11(2), 14:1–14:50 (2010)
9. Faber, W., Leone, N., Pfeifer, G.: Semantics and complexity of recursive aggregates in answer set programming. Artif. Intell. 175(1), 278–298 (2011)
10. Fagin, R., Kolaitis, P., Miller, R., Popa, L.: Data Exchange: Semantics and Query Answering. Theor. Comput. Sci. 336(1), 89–124 (2005)
11. Gebser, M., Kaufmann, B., Schaub, T.: Conflict-driven answer set solving: From theory to practice. Artif. Intell. 187–188, 52–89 (2012)
12. Gebser, M., Sabuncu, O., Schaub, T.: An incremental answer set programming based system for finite model computation. AI Commun. 24(2), 195–212 (2011)
13. Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases. New Generat. Comput. 9(3–4), 365–386 (1991)
14. Grau, B.C., Horrocks, I., Krötzsch, M., Kupke, C., Magka, D., Motik, B., Wang, Z.: Acyclicity conditions and their application to query answering in description logics. In: KR 2012.
15. Hernich, A., Kupke, C., Lukasiewicz, T., Gottlob, G.: Well-founded semantics for extended datalog and ontological reasoning. In: PODS 2013, pp. 225–236. ACM (2013)
16. Heymans, S., Nieuwenborgh, D.V., Vermeir, D.: Open answer set programming with guarded programs. ACM Trans. Comput. Logic 9(4), 26:1–26:53 (2008)
17. Leone, N., Manna, M., Terracina, G., Veltri, P.: Efficiently computable datalog$^{\exists}$ programs. In: KR 2012. AAAI Press.
18. Magka, D., Krötzsch, M., Horrocks, I.: Computing Stable Models for Nonmonotonic Existential Rules. In: IJCAI 2013. AAAI Press, to appear