HEX-Programs with Nested Program Calls

Thomas Eiter, Thomas Krennwallner, and Christoph Redl

Institut für Informationssysteme, Technische Universität Wien
{eiter,tkren,redl}@kr.tuwien.ac.at

Abstract. Answer-Set Programming (ASP) is an established declarative programming paradigm. However, classical ASP lacks subprogram calls as in procedural programming, and access to external computations (akin to remote procedure calls) in general. This feature is desired for increasing modularity and—assuming proper access in place—(meta-)reasoning over subprogram results. While HEX-programs extend classical ASP with external source access, they do not support calls of (sub-)programs upfront. We present *nested* HEX-*programs*, which extend HEX-programs to serve the desired feature in a user-friendly manner. Notably, the answer sets of called sub-programs can be individually accessed. This is particularly useful for applications that need to reason over answer sets like belief set merging, user-defined aggregate functions, or preferences of answer sets. We will further present a novel method for rapid prototyping of external sources by the use of nested programs.

1 Introduction

Answer-Set Programming, based on [8], has been established as an important declarative programming formalism [3]. However, a shortcoming of classical ASP is the lack of means for modularity, i.e., dividing programs into several interacting components. Even though reasoners such as DLV, CLASP, and DLVHEX allow to partition programs into several files, they are still viewed as a single monolithic set of rules. On top of that, passing input to selected (sub-)programs is not possible upfront.

In procedural programming, the idea of calling subprograms and processing their output is in permanent use. Also in functional programming such modularity is popular. This helps reducing development time (e.g., by using third-party libraries), the length of source code, and, last but not least, makes code human-readable. Reading, understanding, and debugging a typical size application written in a monolithic program is cumbersome. Modular extensions of ASP have been considered [9, 5] with the aim of building an overall answer set from program modules; however, multiple results of subprograms (as typical for ASP) are respected, and no reasoning about such results is supported. XASP [11] is an SMODELS interface for XSB-Prolog. This system is related to our work, but in this work the meta-reasoner is Prolog and thus different from the semantics of its subprograms, which are under stable model semantics. The subprograms are monolithic programs and cannot make further calls. This is insufficient for some applications, e.g., for the MELD belief set merging system[10], which require hierarchical nesting of arbitrary depth. Adding such nesting to available approaches is not easily possible and requires to adapt systems similar to our approach.

HEX-programs [6] extend ASP with higher-order atoms, which allow the use of predicate variables, and external atoms, through which a bidirectional communication with external sources is enabled. But HEX-programs do not support modularity and meta-reasoning directly. In this context, modularity means the encapsulation of subprograms which interact through well-defined interfaces only, and meta-reasoning requires reasoning over *sets of* answer sets. Moreover, in HEX-programs external sources are realized as procedural C++ functions. Therefore, as soon as external sources are queried, we leave the declarative formalism. However, the generic notion of external atom, which facilitates a bidirectional data flow between the logic program and an external source (viewed as abstract Boolean function), can be utilized to provide these features.

To this end, we present *nested* HEX-*programs*, which support (possibly parameterized) *subprogram calls*. It is the nature of nested HEX-programs to have multiple programs which reason over the answer sets of each individual subprogram. This can be done in a user-friendly way and enables the user to write purely *declarative* applications consisting of multiple interacting modules. Notably, call results and answer sets are *objects* that can be accessed by identifiers and processed in the calling program. Thus, different from [9, 5] and related formalisms, this enables *(meta)-reasoning about the set of answer sets* of a program. In contrast to [11], both the calling and the called program are in the same formalism. In particular, the calling program has also a declarative semantics. As an important difference to [1], nested HEX-programs do not require extending the syntax and semantics of the underlying formalism, which is the HEX-semantics. The integration is, instead, by defining some external atoms (which is already possible in ordinary HEX-programs), making the approach simple and user-friendly for many applications. Furthermore, as nested HEX-programs are based on HEX-programs, they additionally provide access to external sources other than logic programs. This makes nested HEX-programs a powerful formalism, which has been implemented using the DLVHEX reasoner for HEX-programs; applications like belief set merging [10] show its potential and usefulness. Moreover, we will show how nested programs can be used for external source simulation. This allows for rapid prototyping without actually implementing plugins for the reasoner, which is time-consuming.

2 HEX-Programs

We briefly recall HEX-programs, which have been introduced in [6] as a generalization of (disjunctive) extended logic programs under the answer set semantics [8]; for more details and background, we refer to [6]. A HEX-program consists of rules of the form

$$a_1 \vee \cdots \vee a_n \leftarrow b_1, \ldots, b_m, \text{not } b_{m+1}, \ldots, \text{not } b_n \ , \ (m, n \geq 0)$$

where each a_i is a literal, i.e., an atom $p(t_1, \ldots, t_\ell)$ or a negated atom $\neg p(t_1, \ldots, t_\ell)$, and each b_j is either a classical literal or an external atom, and not is negation by failure (under stable semantics). An *external atom* is of the form

$$\&g[q_1, \ldots, q_k](t_1, \ldots, t_\ell) \ ,$$

where g is an external predicate name, the q_i are predicate names or constants, and the t_j are terms. Informally, the semantics of an external g is given by a $k + \ell + 1$-ary

Boolean *oracle function* $f_{\&g}$. The external atom is true relative to an interpretation I and a grounding substitution θ iff $f_{\&g}(I, q_1, \ldots, q_k, t_1\theta, \ldots, t_\ell\theta) = 1$. External atoms allow for including arbitrary (computable) functions. E.g., built-in functions can be realized via external atoms, or library functions such as string manipulations, sorting routines, etc. As external sources need not be on the same machine, knowledge access across the Web is possible, e.g., belief set import. Strictly, [6] omits classical negation $\neg$ but the extension is routine; furthermore, [6] also allows terms for the q_i and variables for predicate names, which we do not consider.

Example 1. Suppose an external knowledge base consists of an RDF file located on the web at http://.../data.rdf. Using an external atom $\&rdf[\mathtt{url}](X, Y, Z)$, we may access all RDF triples (s, p, o) at the URL specified with `url`. To form belief sets of pairs that drop the third argument from RDF triples, we may use the rule

$$bel(X, Y) \leftarrow \&rdf[\text{http://.../data.rdf}](X, Y, Z) \ .$$

The above program has a single answer set which consists of all literal $bel(c_1, c_2)$ such some RDF triple (c_1, c_2, c_3) occurs at the respective URL.

We use the DLVHEX system from http://www.kr.tuwien.ac.at/research/systems/dlvhex/ as a backend. DLVHEX implements (a fragment of) HEX-programs. It provides a plugin mechanism for external atoms. Besides library atoms, the user can define her own atoms using C++ methods.

3 Nested HEX-Programs

Limitations of ASP. As a simple example demonstrating the limits of ordinary ASP, assume a program computing the shortest paths between two (fixed) nodes in a connected graph. The answer sets of this program then correspond to the shortest paths. Suppose we are just interested in the *number* of such paths. In a procedural setting, this is easily computed if the function returns all paths in an suitable data structure (e.g., an array or a linked list).

In ASP, the solution is non-trivial if the given program must not be modified (e.g., if it is provided by a third party); above, we must count the answer sets. Thus, we need to reason on *sets of* answer sets, which is infeasible inside the program. Means to call the program at hand and reason about the results of this *"callee"* (*subprogram*) in the *"calling program"* (*host program*) would be useful. Aiming at a logical counterpart to procedural function calls, we define a framework which allows to input facts to the subprogram prior to its execution. Host and subprograms are decoupled and interact merely by relational input and output values. To realize this mechanism, we exploit external atoms, leading to nested HEX-programs.

Architecture. Nested HEX-programs are realized as a plugin for the reasoner DLVHEX,[1] which consists of a *set of external atoms* and an *answer cache* for the results of subprograms (see Fig. 1). Technically, the implementation is part of the belief set merging

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

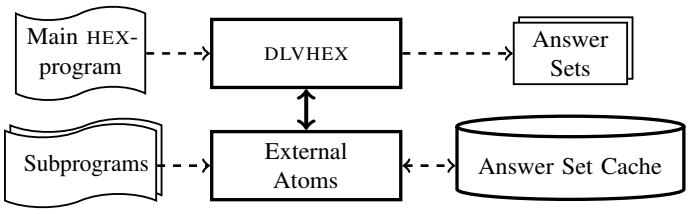

Fig. 1: System Architecture of Nested HEX (data flow ⇢, control flow →)

system MELD, which is an application on top of a nested HEX-programs core. This core can be used independently from the rest of the system.

When a subprogram call (corresponding to the evaluation of a special external atom) is encountered of the host program, the plugin creates another instance of the reasoner to evaluate the subprogram. Its result is then stored in the answer cache and identified with a unique *handle*, which can later be used to reference the result and access its components (e.g., predicate names, literals, arguments) via other special external atoms.

There are two possible sources for the called subprogram: (1) either it is *directly embedded* in the host program, or (2) it is *stored in a separate file*. In (1), the rules of the subprogram must be represented within the host program. To this end, they are encoded as string constants. An embedded program must not be confused with a subset of the rules of the host program. Even though it is syntactically part of it, it is logically separated to allow independent evaluation. In (2) merely the *path* to the location of the external program in the file system is given. Compared to embedded subprograms, code can be reused without the need to copy, which is clearly advantageous when the subprogram changes. We now present concrete external atoms $\&callhex_n$, $\&callhexfile_n$, $\&answersets$, $\&predicates$, and $\&arguments$ which are used to realize nested HEX-programs.

External Atoms for Subprogram Handling. We start with two families of external atoms

$$\&callhex_n[\mathtt{P}, p_1, \ldots, p_n](H) \quad \text{and} \quad \&callhexfile_n[\mathtt{FN}, p_1, \ldots, p_n](H)$$

that allow to execute a subprogram given by a string `P` respectively in a file `FN`; here n is an integer specifying the number of predicate names p_i, $1 \leq i \leq n$, used to define the input facts. When evaluating such an external atom relative to an interpretation I, the system adds all facts $\{p_i(a_1, \ldots, a_{m_i}) \leftarrow \mid p_i(a_1, \ldots, a_{m_i}) \in I\}$ to the specified program, creates another instance of the reasoner to evaluate it, and returns a symbolic handle H as result. For convenience, we do not write n in $\&callhex_n$ and $\&callhexfile_n$ as it is understood from the usage.

Example 2. In the following program, we use two predicates p_1 and p_2 to define the input to the subprogram `sub.hex` ($n = 2$), i.e., all atoms over these predicates are added to the subprogram prior to evaluation. The call derives a handle H as result.

$$\begin{gathered} p_1(x, y) \leftarrow \qquad p_2(a) \leftarrow \qquad p_2(b) \leftarrow \\ handle(H) \leftarrow \&callhexfile[\mathtt{sub.hex}, p_1, p_2](H) \end{gathered}$$

A *handle* is a unique integer representing a certain program answer cache entry. In the implementation, handles are consecutive numbers starting with 0. Hence in the example the unique answer set of the program is $\{handle(0)\}$ (neglecting facts).

Formally, given an interpretation I, $f_{\&callhexfile_n}(I, file, p_1, \ldots, p_n, h) = 1$ iff h is the handle to the result of the program in file *file*, extended by the facts over predicates $p_1, \ldots, p_n$ that are true in I. The formal notion and use of $\&callhex_n$ to call embedded subprograms is analogous to $\&callhexfile_n$.

Example 3. Consider the following program:

$$h_1(H) \leftarrow \&callhexfile[\texttt{sub.hex}](H)$$
$$h_2(H) \leftarrow \&callhexfile[\texttt{sub.hex}](H)$$
$$h_3(H) \leftarrow \&callhex[\texttt{a} \leftarrow . \ \texttt{b} \leftarrow .](H)$$

The rules execute the program `sub.hex` and the embedded program $P_e = \{a \leftarrow,\ b \leftarrow\}$. No facts will be added in this example. The single answer set is $\{h_1(0), h_2(0), h_3(1)\}$ resp. $\{h_1(1), h_2(1), h_3(0)\}$ depending on the order in which the subprograms are executed (which is irrelevant). While $h_1(X)$ and $h_2(X)$ will have the same value for X, $h_3(Y)$ will be such that $Y \neq X$. Our implementation realizes that the result of the program in `sub.hex` is referred to twice but executes it only once; P_e is (possibly) different from `sub.hex` and thus evaluated separately.

Now we want to determine how many (and subsequently which) answer sets it has. For this purpose, we define external atom $\&answersets[PH](AH)$ which maps handles PH to call results to sets of respective answer set handles. Formally, for an interpretation I, $f_{\&answersets}(I, h_P, h_A) = 1$ iff h_A is a handle to an answer set of the program with program handle h_P.

Example 4. The single rule

$$ash(PH, AH) \leftarrow \&callhex[\texttt{a} \vee \texttt{b} \leftarrow .](PH), \&answersets[PH](AH)$$

calls the embedded subprogram $P_e = \{a \vee b \leftarrow .\}$ and retrieves pairs (PH, PA) of handles to its answer sets. $\&callhex$ returns a handle $PH = 0$ to the result of P_e, which is passed to $\&answersets$. This atom returns a *set* of answer set handles (0 and 1, as P_e has two answer sets, viz. $\{a\}$ and $\{b\}$). The overall program has thus the single answer set $\{ash(0,0), ash(0,1)\}$. As for each program the answer set handles start with 0, only a pair of program and answer set handles uniquely identifies an answer set.

We now are ready to solve our example of counting shortest paths from above.

Example 5. Suppose `paths.hex` is the search program and encodes each shortest path in a separate answer set. Consider the following program:

$$as(AH) \leftarrow \&callhexfile[\texttt{paths.hex}](PH), \&answersets[PH](AH)$$
$$number(D) \leftarrow as(C), D = C + 1, \text{not } as(D)$$

The second rule computes the first free handle D; the latter coincides with the number of answer sets of `paths.hex` (assuming that some path between the nodes exists).

At this point we still treat answer sets of subprograms as black boxes. We now define an external atom to investigate them.

Given an interpretation I, $f_{\&predicates}(I, h_P, h_A, p, a) = 1$ iff p occurs as an a-ary predicate in the answer set identified by h_P and h_A. Intuitively, the external atom maps pairs of program and answer set handles to the predicates names with their associated arities occurring in the accourding answer set.

Example 6. We illustrate the usage of $\&predicates$ with the following program:

$$\begin{aligned} preds(P, A) \leftarrow\ & \&callhex[\texttt{node(a). node(b). edge(a, b).}](PH), \\ & \&answersets[PH](AH), \&predicates[PH, AH](P, A) \end{aligned}$$

It extracts all predicates (and their arities) occurring in the answer of the embedded program P_e, which specifies a graph. The answer set is $\{preds(node, 1), preds(edge, 2)\}$ as the answer set of P_e has atoms with predicate $node$ (unary) and $edge$ (binary).

The final step to gather all information from the answer of a subprogram is to extract the *literals* and their *parameters* occurring in a certain answer set. This can be done with external atom $\&arguments$, which is best demonstrated with an example.

Example 7. Consider the following program:

$$\begin{aligned} h(PH, AH) \leftarrow\ & \&callhex[\texttt{node(a). node(b). node(c). edge(a, b).edge(c, a).}](PH), \\ & \&answersets[PH](AH) \\ edge(W, V) \leftarrow\ & h(PH, AH), \&arguments[PH, AH, \texttt{edge}](I, 0, V), \\ & \&arguments[PH, AH, \texttt{edge}](I, 1, W) \\ node(V) \leftarrow\ & h(PH, AH), \&arguments[PH, AH, \texttt{node}](I, 0, V) \end{aligned}$$

It extracts the directed graph given by the embedded subprogram P_e and reverses all edges; the answer set is $\{h(0,0), node(a), node(b), node(c), edge(b, a), edge(a, c)\}$. Indeed, P_e has a single answer set, identified by $PH = 0, AH = 0$; via $\&arguments$ we can access in the second resp. third rule the facts over $edge$ resp. $node$ in it, which are identified by a unique literal id I; the second output term of $\&arguments$ is the argument position, and the third the actual value at this position. If the predicates of a subprogram were unknown, we can determine them using $\&predicates$.

To check the sign of a literal, the external atom $\&arguments[PH, AH, P](I, s, S)$ supports argument s. When $s = 0$, $\&arguments$ will match the sign of the I-th *positive* literal over predicate P into S, and when $s = 1$ it will match the corresponding classically negated atom.

External Atoms for External Source Prototyping. Our system provides another family of external atoms for rapid prototyping of (simple) external sources directly in ASP. This it, the input-output behavior of hypothetical external sources is encoded by ASP rules. This is useful for quick experiments before a new external source is actually implemented. It comes with less implementation overhead compared to a native implementation in C++. This gives the user the possibility to see how the planned external atom will behave in a program even before it is developed. However, it is clear the possibility of simulating external sources cannot replace the plugin mechanism of DLVHEX as

it cannot access real external sources. Moreover, simulation is less efficient than a native implementation in C++.

For simulation our system supports the external atom:

$$\&simulator_{n,m}[\mathtt{F}, p_1, \ldots, p_n](X_1, \ldots X_m)$$

The simulator atom takes as arguments a filename F, which refers to the ASP program defining the input-output behavior of the prototypical external source, and predicate inputs $p_1, \ldots, p_n$. The output list $X_1, \ldots, X_m$ is used to retrieve the tuples from produced by the simulated external source.

When a simulator atom is encountered in the host program, it will evaluate the ASP-program in `F` extended by the input parameters defined over $p_1, \ldots, p_n$. In particular, the system will add for each input atom $p_i(a_1, \ldots, a_k)$ a fact of form $in_i(a_1, \ldots, a_k)$ to `F`. The renaming of the predicates is necessary in order to make `F` independent of the input predicate names in the host program. The result of `F` is expected to consist of exactly one answer set, where all atoms of form $out(o_1, \ldots, o_m)$ define the output of the simulated external source.

Example 8. Consider the following program P given by the rules:

$$\begin{aligned} dom(a) \leftarrow \qquad dom(b) \leftarrow \qquad dom(c) \leftarrow \\ sel(X) \leftarrow dom(X), \&simulator_{2,1}[\mathtt{Q}, dom, nsel](X) \\ nsel(X) \leftarrow dom(X), \&simulator_{2,1}[\mathtt{Q}, dom, sel](X) \end{aligned}$$

Let further `Q` refer to the program:

$$out(X) \leftarrow in_1(X), \text{not } in_2(X).$$

Then `Q` simulates an external source which computes the set difference, where the extension of the second predicate input in_2 is subtracted from the extension of the first predicate input in_1. The program P computes then the two sets sel and $nsel$, corresponding to all partitionings of $\{a, b, c\}$ into two subsets.

4 Applications

MELD. The MELD system [10] deals with merging multiple *collections of belief sets*. Roughly, a belief set is a set of classical ground literals. Practical examples of belief sets include explanations in abduction problems, encodings of decision diagrams, and relational data. The merging strategy is defined by tree-shaped *merging plans*, whose leaves are the collections of belief sets to be merged, and whose inner nodes are *merging operators* (provided by the user). The structure is akin to syntax trees of terms.

The automatic evaluation of tree-shaped merging plans is based on nested HEX-programs; it proceeds bottom-up, where every step requires inspection of the subresults, i.e., accessing the answer sets of subprograms. Note that for nesting of ASP-programs with arbitrary (finite) depth, XASP [11] is not appropriate.

Aggregate Functions. Nested programs can also emulate aggregate functions [7] (e.g., `#sum`, `#count`, `#max`) where the (user-defined) host program computes the

function given the result of a subprogram. This can be generalized to aggregates over *multiple* answer sets of the subprogram; e.g., to answer set counting, or to find the minimum/maximum of some predicate over all answer sets (which may be exploited for global optimization).

Generalized Quantifiers. Nested HEX-programs make the implementation of brave and cautious reasoning for query answering tasks very easy, even if the backend reasoner only supports answer set enumeration. Furthermore, extended and user-defined types of query answers (cf. [5]) are definable in a very user-friendly way, e.g., majority decisions (at least half of the answer sets support a query), or minimum and/or maximum number based decisions (qualified number restrictions).

Preferences. Answer sets as accessible objects can be easily compared wrt. user-defined preference rules, and used for filtering as well as ranking results (cf. [4]): a host program selects appropriate candidates produced by a subprogram, using preference rules. The latter can be elegantly implemented as ordinary integrity constraints (for filtering), or as rules (possibly involving further external calls) to derive a rank. A popular application are online shops, where the past consumer behavior is frequently used to filter or sort search results. Doing the search via an ASP program which delivers the matches in answer sets, a host program can reason about them and act as a filter or ranking algorithm.

Nested Programs as a Development Tool for DLVHEX. The further development of our system DLVHEX uses the idea of annotated external sources. This is, known properties like monotonicity and functionality shall be exploited for speeding up the reasoning process. Developing appropriate algorithms and heuristics requires empirical experiments with a variety of external sources. As it would be cumbersome to implement all of them as real plugins to DLVHEX, simulating them via our $\&simulator_{n,m}$ atom seems to be a good alternative.

5 Conclusion

To overcome limitations of classical ASP regarding subprograms and reasoning about their possible outcomes, we briefly presented *nested* HEX-*programs*, which realize subprogram calls via special external atoms of HEX-programs; besides modularity, a plus for readability and program reusability, they allow for reasoning over *multiple* answer sets (of subprograms). Moreover, nested HEX-programs can also be used as a tool for rapid external source prototyping. An implementation on top of DLVHEX is available. Related to this is the work on macros in [2], which allow to call macros in logic programs.

The possibility to access answer sets in a host program, in combination with access to other external computations, makes nested HEX-programs a powerful tool for a number of applications. In particular, libraries and user-defined functions can be incorporated into programs easily. As an interesting aspect is that dynamic program assembly (using a suitable string library) and execution are possible, which other approaches to modular ASP programming do not offer. Exploring this remains for future work.

References

1. Analyti, A., Antoniou, G., Damásio, C.V.: Mweb: a principled framework for modular web rule bases and its semantics. ACM Trans. Comput. Log. 12(2), 17:1–17:46 (2011)
2. Baral, C., Dzifcak, J., Takahashi, H.: Macros, Macro calls and Use of Ensembles in Modular Answer Set Programming. In: ICLP'06. pp. 376–390. Springer (2006)
3. Brewka, G., Eiter, T., Truszczyński, M.: Answer set programming at a glance. Commun. ACM 54(12), 92–103 (2011)
4. Delgrande, J.P., Schaub, T., Tompits, H., Wang, K.: A classification and survey of preference handling approaches in nonmonotonic reasoning. Comput. Intell. 20(2), 308–334 (2004)
5. Eiter, T., Gottlob, G., Veith, H.: Modular Logic Programming and Generalized Quantifiers. In: LPNMR'97. pp. 290–309. Springer (1997)
6. Eiter, T., Ianni, G., Schindlauer, R., Tompits, H.: A Uniform Integration of Higher-Order Reasoning and External Evaluations in Answer Set Programming. In: IJCAI'05. pp. 90–96. Professional Book Center (2005)
7. Faber, W., Leone, N., Pfeifer, G.: Semantics and complexity of recursive aggregates in answer set programming. Artif. Intell. 175(1), 278–298 (2011)
8. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and deductive databases. New Generat. Comput. 9, 365–385 (1991)
9. Janhunen, T., Oikarinen, E., Tompits, H., Woltran, S.: Modularity Aspects of Disjunctive Stable Models. J. Artif. Intell. Res. 35, 813–857 (2009)
10. Redl, C., Eiter, T., Krennwallner, T.: Declarative Belief Set Merging using Merging Plans. In: PADL'11. pp. 99–114. Springer (2011)
11. Swift, T., Warren, D.S.: XSB: Extending Prolog with Tabled Logic Programming. Theor. Pract. Log. Prog. 12(1–2), pp. 157–187 (2012)