Nested HEX-Programs*

Thomas Eiter, Thomas Krennwallner, and Christoph Redl

Institut für Informationssysteme, Technische Universität Wien
Favoritenstraße 9-11, A-1040 Vienna, Austria
{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 (like remote procedure calls) in general. The 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.

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 modular programming, 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 sets 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 scenario 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, 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 external sources of computation can be accessed. 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 HEX-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

* This research has been supported by the Austrian Science Fund (FWF) project P20840 and P20841, and by the Vienna Science and Technology Fund (WWTF) project ICT 08-020.
We briefly recall HEX. Via such atoms, arbitrary (computable) functions can be included. E.g., built-in functions can be realized via

where each

is an external predicate name, the

are predicate names or constants, and the

are terms. Informally, the semantics of an external

is given by a

-ary Boolean oracle function

. The external atom

is true relative to an interpretation

and a grounding substitution

iff

. Via such atoms, arbitrary (computable) functions can be included. 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

but the extension is routine; furthermore, [6] also allows terms for the

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

Using an external atom

, we may access all RDF triples

at the URL specified with

. To form belief sets of pairs that drop the third argument from RDF triples, we may use the rule

The semantics of HEX-program is given via answer sets, which are sets of ground literals closed under the rules that satisfy a stability condition as in [6]; we refer to [6] for technical details. The above program has a single answer set which consists of all literal

such some RDF triple

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, where for evaluation a C++ routine must be provided.

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

of such paths. In a procedural setting, this is easily computed: if a function returns all these paths in an array, linked list, or similar data structure, then counting its elements is trivial.
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\(^\dagger\), 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 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 in 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.

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

\[
\&\text{callhex}_{n}[P, p_1, \ldots, p_n](H) \quad \text{and} \quad \&\text{callhexfile}_{n}[FN, p_1, \ldots, p_n](H)
\]

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 \(p_i(a_1, \ldots, a_{m_i}) \leftarrow\) over \(p_i\) (with arity \(m_i\)) that are true 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.

\[
p_1(x, y) \leftarrow \quad p_2(a) \leftarrow \quad p_2(b) \leftarrow \quad handle(H) \leftarrow \&\text{callhexfile}[\text{sub.hex}, p_1, p_2](H)
\]

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

\(^\dagger\text{http://www.kr.tuwien.ac.at/research/systems/dlvhex/meld.html}\)
Formally, given an interpretation $I$, $f_{&\text{callhexfile}}(I, \text{file}, p_1, \ldots, p_n, h) = 1$ iff $h$ is the handle to the result of the program in file \text{file}, extended by the facts over predicates $p_1, \ldots, p_n$ that are true in $I$. The formal notion and use of $&\text{callhex}$ to call embedded subprograms is analogous to $&\text{callhexfile}$.

Example 3. Consider the following program:

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

The rules execute the program \text{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 $\{h_1(0), h_2(0), h_3(1)\}$ 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 \text{sub.hex} is referred to twice but executes it only once; $P_e$ is (possibly) different from \text{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 $&\text{answersets}[PH](AH)$ which maps handles $PH$ to call results to sets of respective answer set handles. Formally, for an interpretation $I$, $f_{&\text{answersets}}(I, h_P, h_A) = 1$ iff $h_A$ is a handle to an answer set of the program with program handle $h_P$.

Example 4. The program

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

calls the embedded subprogram $P_e = \{a \lor b \leftarrow \}$ and retrieves pairs $(PH, PA)$ of handles to its answer sets. $&\text{callhex}$ returns a handle $PH = 0$ to the result of $P_e$, which is passed to $&\text{answersets}$. 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 $\{\text{ash}(0, 0), \text{ash}(0, 1)\}$. 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 \text{paths.hex} is the search program and encodes each shortest path in a separate answer set. Consider the following program:

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

The second rule computes the first free handle $D$; the latter coincides with the number of answer sets of \text{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 $I$, $f_{&\text{predicates}}(I, h_P, h_A, p, a) = 1$ 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 according answer set.

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

$$\text{preds}(P, A) \leftarrow &\text{callhex}[\text{node}(a), \text{node}(b), \text{edge}(a, b)](PH), &\text{answersets}[PH](AH), &\text{predicates}[PH, AH](P, A)$$

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

\[
\begin{align*}
    h(PH, AH) &\leftarrow &\text{callhex} &\{ \text{node}(a), \text{node}(b), \text{node}(c), \text{edge}(a,b), \text{edge}(c,a), \}([PH], \text{answersets}[PH](AH)) \\
    \text{edge}(W, V) &\leftarrow h(PH, AH), \text{answersets}[PH, AH, \text{edge}](1, 0, V), \text{answersets}[PH, AH, \text{node}](1, 0, V) \\
    \text{node}(V) &\leftarrow h(PH, AH), \text{answersets}[PH, AH, \text{node}](1, 0, V)
\end{align*}
\]

It extracts the directed graph given by the embedded subprogram \( P_e \) and reverses all edges; the single answer set is \( \{ h(0, 0), \text{node}(a), \text{node}(b), \text{node}(c), \text{edge}(b,a), \text{edge}(a,c) \} \). 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, Pred](I, s, Sign) supports argument \( s \). When \( s = 0 \), &arguments will match the sign of the \( I \)-th positive literal over predicate \( Pred \) into \( Sign \), and when \( s = 1 \) it will match the corresponding classically negated atom.

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. [3]) 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. [2]): 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.
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). An prototype 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