Promoting Modular Nonmonotonic Logic Programs*

Thomas Krennwallner

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

Abstract

Modularity in Logic Programming has gained much attention over the past years. To date, many formalisms have been proposed that feature various aspects of modularity. In this paper, we present our current work on Modular Nonmonotonic Logic Programs (MLPs), which are logic programs under answer set semantics with modules that have contextualized input provided by other modules. Moreover, they allow for (mutually) recursive module calls. We pinpoint issues that are present in such cyclic module systems and highlight how MLPs addresses them.

Keywords and phrases Knowledge Representation, Nonmonotonic Reasoning, Modular Logic Programming, Answer Set Programming

1 Introduction and Problem Description

Answer set programming (ASP) is an approach for declarative problem solving geared towards search problems. More specifically, problems are represented by nonmonotonic logic programs, such that the *stable models* (or *answer sets*) [19] of the program represent the solutions to a given problem instance. ASP has many applications in knowledge representation and problems in artificial intelligence including planning, diagnosis, and configuration.

A natural way to design software for solving problems is to identify easier to handle subproblems that can be solved independently from each other, and then based on this analysis to craft corresponding software components that solve the subproblems: the modules. The combination of these components then gives an implementation for the whole problem. Most general-purpose programming languages have their own way to introduce modularity, a key concept that helps developing software artifacts. Techniques like information hiding, abstraction, and structured programming are well-established principles for breaking down sub-tasks in an imperative program, and essentially any standard programming language has amenities that allow to define input/output interfaces to modules for easy code-reuse in implementations of possibly unrelated problems. Testing software greatly benefits from structured programs, since it involves defining well-suited interfaces to the components, which in turn assists writing testcases. When many programmers are working on a project, the strict component-wise building of software is the only way to success. In contrast, it is customary to view logic programs as monolithic entities, i.e., one program is tailored to solve a particular problem without a clear separation of the sub-tasks, albeit the same principle of creating manageable pieces will help users of logic programming systems building knowledge bases. Having an explicit way to modularize knowledge in logic programs is thus needed and adding modularity principles to ASP has several advantages like easy knowledge base reuse by clean input/output interfaces and helping to model complex problem domains by focusing

* This research has been supported by the Austrian Science Fund project P20841, by the Vienna Science and Technology Fund project ICT 08-020, and by the EC project OntoRule (IST-2009-231875).

on smaller parts first. This issue has been identified and various notions for modularizing logic programs have been proposed to support testing logic programs, reusing and abstracting components, and maintaining program code.

However, there are obstacles that impede to bring such characteristics to ASP. Traditional answer set semantics has no module concept and there is no straightforward way that would allow that. It is not clear how a semantics should be defined that caters for modules, as the declarative nature of ASP does not distinguish between knowledge stored in different logic programs (when viewed as modules). Another issue is to allow for cyclic module systems, i.e., when modules mutually refer to each other. Modules that have such cyclic dependencies may bring in semantic issues like unfounded models that would not be present when viewing logic programs as single unit. Both of these problems are related to the declarative nature of ASP, and any prospective model-theoretic semantics for modular ASP has to deal with unwanted semantic deficits. Methods that bring modularity aspects closer to ASP have not yet stood the test of time, and no single semantics has gained general acceptance.

The aim of this paper is to recall existing approaches in modular logic programming and to present work and results on a novel formalism to modular ASP: Modular Nonmonotonic Logic Programs (MLPs) [10]. We pinpoint peculiar issues that exist in modular frameworks for ASP and highlight how the MLP formalism addresses them. We conclude with prospective future work and open research issues.

2 Background and Overview of Existing Literature

There is a long history of research in investigating modularity principles in logic programming. A good overview provides [5, 7], which studies modularity in the context of traditional definite Horn logic programming. In general, they identify two directions for investigating modularity aspects in logic programming: (i) *Programming-in-the-large*, which introduce compositional operators to combine separate and independent modules; and (ii) *Programming-in-the-small*, which builds upon abstraction and scoping mechanisms. Early influential work on modularity in logic programming include [17] and [18], where the former can be seen as an approach for (ii), while the latter is a prototypical instance of (i).

In the context of answer set semantics, whose focus lies in the treatment of negation-as-failure and disjunctive rules, several important proposals have been put forward. Representatives for (i) are DLP-functions [21] and modular SMODELS programs [25], which has recently been generalized to a module-based framework for multi-language constraint modeling [22], and to modular P-log programs [9] that combines probabilistic reasoning with logic programs. Another proponent [28] is concerned with operator splitting similar in the vein of splitting sets [24]. Exponents in (ii) are modular logic programs with generalized quantifiers [15], macros [3], templates [8], and web rule bases [1]. On a broader scale, multi-agent scenarios with logic programs has been studied in social logic programs [6] and communicating ASP [4].

3 Goal of the Research

As described above, several semantics exist that deal with modularity in ASP. Virtually all semantics are defined such that mutual recursion between modules is disallowed. While this helps to simplify the definitions of a semantics for modular ASP, in general this may bring issues when different, possibly independently developed modules are combined. Many natural problems exist that have an inherent cyclic flavor, and ruling out the chance to model problems using modules that depend on each other may be too restrictive in practice, or

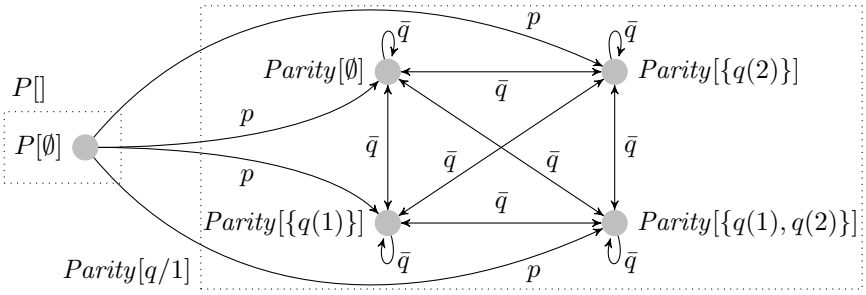

Figure 1 Call graph of instantiated modules in Example 1

even force to use unintuitive encodings. We aim at defining a model-theoretic semantics that caters for this situation, investigate its semantic properties and computational complexity, and develop novel evaluation algorithms for such modular nonmonotonic logic programs. The next example illustrates cycles in modular logic programming using Modular Nonmonotonic Logic Programs (MLP) as defined in [10], a formalism that admits arbitrary non-ground disjunctive nonmonotonic logic programs as modules. MLPs can be seen as a proponent of the programming-in-the-small approach to modular programming, as it is using module atoms as a language construct to access knowledge encoded in other modules. We sketch the basic building blocks of MLPs and refer to [10] for proper formal definitions.

▶ **Example 1.** Consider the following recursive module $Parity[q/1]$ consisting of four rules, which determines whether a set has an even respectively odd number of elements:

$$\bar{q}(X) \vee \bar{q}(Y) \leftarrow q(X), q(Y), X \neq Y \qquad odd \leftarrow skip(X), Parity[\bar{q}].even$$
$$skip(X) \leftarrow q(X), not\ \bar{q}(X) \qquad even \leftarrow not\ odd$$

Here, $q/1$ is a (formal) unary input predicate that stores the set. The first two rules on the left have the effect, by stability of answer sets, that q becomes $\bar{q}$ with one element randomly removed (for which *skip* is true, as defined in the lower left rule). The third rule top right determines recursively whether q stores an odd number of elements using the *module atom* $Parity[\bar{q}].even$, while the last rule bottom right defines *even* as the complement of *odd*. Intuitively, if we call the module *Parity* with a predicate p for input, then *even* is computed true, which is expressed by $Parity[p].even$, whenever p stores an even number of elements. Note that *Parity* is recursive, and for empty input p it calls itself with the same input.

We demonstrate the use of *Parity* in an MLP with the (main) module $P[]$ with empty input, which calls *Parity* with a set p of two elements:

$$p(1) \leftarrow \qquad p(2) \leftarrow \qquad pev \leftarrow Parity[p].even$$

The combination of both modules gives the cyclic MLP $\mathbf{P} = (P[], Parity[q/1])$. On the surface, $\mathbf{P}$ can be seen as an "uninstantiated" modular program, whose semantics is given by characterizing models at modules which have been instantiated with a set of input facts: the *value calls*. Figure 1 depicts the *call graph* (the principle dependencies) of $\mathbf{P}$ with value calls as nodes and edges labeled with input predicates; e.g., value call $P[\emptyset]$ calls $Parity[\{q(1), q(2)\}]$ on input p. The dotted boxes highlight the modules from which the value calls on the inside have been generated. Loosely speaking, MLPs encode schematic dependencies between modules, and instantiated modules then can be used to define a semantics that takes module input into account which is defined over possibly cyclic modules. Different interpretations

of an MLP select different subgraphs of its call graph, and answer sets are defined based on the selected subgraphs. For instance, **P** has two answer sets in which *pev* is true at the main instantiation $P[\emptyset]$ and *even* is true at $Parity[\{q(1), q(2)\}]$ and $Parity[\emptyset]$, whereas *odd* is satisfied at $Parity[\{q(1)\}]$ and $Parity[\{q(2)\}]$. Both answer sets are symmetric on the guess of $\bar{q}$ at $Parity[\{q(1), q(2)\}]$, but otherwise equal.

4 Current Status

We have an advanced understanding of peculiar issues that arise when we allow for module cycles in MLPs. One key aspect is the use of the FLP-reduct [16] instead of the traditional GL-reduct [19] to cure semantic issues when dealing with negation-as-failure over potential nonmonotonic module atoms. Roughly, given an interpretation of a program, the GL-reduct first removes each rule whose negative body is false in the interpretation, and then cut offs the negative literals from remaining rules. On the other hand, the FLP-reduct just removes rules whose body is unsatisfied in a given interpretation, which leaves negative literals in the result of this translation. Applied to traditional answer set programs, both reducts are equivalent, but FLP-semantics is beneficial for language extensions of ASP such as logic programs with aggregates. In the context of MLPs, the FLP-semantics guarantees that models are minimal, thus we retain groundedness of the semantics and prohibit unfounded answer sets. Another aspect of MLP is to contextualize module instantiation. Here, relevant instantiations are a concept to concentrate on the important part of all instantiated modules. In general, module instantiation plays a key role for the definition of a semantics for MLPs. Akin to the call semantics of imperative programming languages, the module instantiation employed in MLPs can be seen as *call-by-value* mechanism, where module instantiation calls other instantiations with explicit input facts. This is in contrast to the module framework of DLP-functions [21], which can be classified as *call-by-reference* mechanism; input here is given implicitly by the models of each module.

Further results show that MLPs have an increase in computational complexity compared to standard ASP: propositional Horn-MLPs with unrestricted cyclic input over modules are EXP-complete, and non-ground ones are 2EXP-complete. If we restrict propositional MLPs such that modules have no input predicates, we obtain for instance that checking satisfiability of normal propositional MLPs is NP-complete, and for disjunctive MLP it is Σ_2^p-complete. In general, checking answer set existence of arbitrary normal non-ground MLPs is 2NEXP-complete, and 2NEXP^{NP}-complete for the disjunctive case.

5 Preliminary Results

The work in [10] devised a novel semantics for MLPs that allows for mutual recursion between modules. We have studied the semantic properties of MLPs, their computational complexity, and compared it to DLP-functions [21]; interestingly, DLP-functions can be seen as MLPs that have no module input parameters. MLPs conservatively extend ordinary logic programs, and many semantic properties of answer set programs generalize to MLPs. For instance, the important property that every answer set of an MLP is a minimal model implies that answer sets in the MLP setting are grounded (see discussion above).

In [13], we investigated the relationships between various semantics for modular logic programs and other nonmonotonic formalisms. We have provided a more systematic view of approaches in combining nonmonotonic knowledge bases and classified formalisms based on the program reduct and on the environment view, i.e., whether their semantics is defined

in terms of local models for each individual knowledge base that implicitly converge to a semantics for the combined system, or whether the formalism has a global state using a collection of explicitly accessible local models.

We developed a novel evaluation algorithm for MLPs in [11]. Here, we concentrated on an MLP fragment called input- and call-stratified MLPs, whose stratification can be evaluated in a top-down fashion starting from uninstantiated modules. This way we could generalize the splitting sets technique to MLPs and develop an evaluation algorithm that traverses the call graph and instantiates modules on-the-fly. Example 1 above is input-call-stratified, and the techniques developed in [11] are applicable to it.

We worked on two characterizations of MLPs in terms of classical models by investigating the notions of loop formulas [23] and ordered completion [2] in MLPs [12]. The results include (i) *modular loop formulas* based on loops over module instantiations, and (ii) *ordered completion for MLPs* without using explicit loop formulas. We generalized Clark's completion and positive dependency graph to MLPs with respect to different module instantiations. Based on these results, we defined modular loop formulas that capture MLP semantics. The second contribution was to explore ordered completion in the realm of MLPs. Here, fresh predicates ensures a derivation order, and program completion is only active for those predicates that do not participate in a positive loop, possibly involving module instantiations.

6 Open Issues

Future work includes to find further useful fragments of MLPs and characterize their computational complexity. Based on first results on loop formulas and ordered completion for MLPs [12] we seek to develop new algorithms that interweave conflict-driven model building with module instantiation. Related to this is to investigate first-order theorem proving techniques in the context of MLPs. Another line of research is to improve the understanding of MLP semantics and give it a logical foundation using (generalized) equilibrium logic [26] and applying results on FLP-semantics in [27]. Furthermore, we want to relax the restriction to minimal models in non-relevant instantiations and use semi-equilibrium models [14] instead. As a prospective application we want to investigate dl-programs with Datalog-rewritable Description Logics [20]. Intuitively, the Description Logic knowledge base can be rewritten to a module, and dl-atoms that appear in the logic program of the dl-program can be rewritten as module atoms that refer to this module. Moreover, we are currently developing a prototype implementation to evaluate input-call stratified MLPs.

Acknowledgments. I would like to thank my supervisor Prof. Dr. Thomas Eiter and Dr. Michael Fink for their ongoing support, as well as my fellow co-author Minh Dao-Tran.

References

1 A. Analyti, G. Antoniou, and C. V. Damásio. MWeb: a principled framework for modular web rule bases and its semantics. *ACM Trans. Comput. Logic*, 12(2):17:1–17:46, 2011.

2 V. Asuncion, F. Lin, Y. Zhang, and Y. Zhou. Ordered completion for first-order logic programs on finite structures. In *AAAI'10*, pp. 249–254. AAAI Press, 2010.

3 C. Baral, J. Dzifcak, and H. Takahashi. Macros, macro calls and use of ensembles in modular answer set programming. In *ICLP'06*, pp. 376–390. Springer, 2006.

4 K. Bauters, S. Schockaert, D. Vermeir, and M. De Cock. Communicating ASP and the polynomial hierarchy. In *LPNMR'11*, pp. 67–79. Springer, 2011.

5 A. Brogi, P. Mancarella, D. Pedreschi, and F. Turini. Modular logic programming. *ACM Trans. Prog. Lang. Syst.*, 16(4):1361–1398, 1994.

6 F. Buccafurri and G. Caminiti. Logic programming with social features. *Theo. Pract. Logic Progr.*, 8(5-6):643–690, 2008.
7 M. Bugliesi, E. Lamma, and P. Mello. Modularity in logic programming. *J. Logic Prog.*, 19/20:443–502, 1994.
8 F. Calimeri and G. Ianni. Template programs for disjunctive logic programming: An operational semantics. *AI Comm.*, 19(3):193–206, 2006.
9 C. V. Damásio and J. Moura. Modularity of P-Log programs. In *LPNMR'11*, pp. 13–25. Springer, 2011.
10 M. Dao-Tran, T. Eiter, M. Fink, and T. Krennwallner. Modular nonmonotonic logic programming revisited. In *ICLP'09*, pp. 145–159. Springer, 2009.
11 M. Dao-Tran, T. Eiter, M. Fink, and T. Krennwallner. Relevance-driven evaluation of modular nonmonotonic logic programs. In *LPNMR'09*, pp. 87–100. Springer, 2009.
12 M. Dao-Tran, T. Eiter, M. Fink, and T. Krennwallner. First-order encodings of modular nonmonotonic logic programs. In *Datalog 2.0*. Springer, 2011. `http://datalog20.org/`, to appear.
13 T. Eiter, G. Brewka, M. Dao-Tran, M. Fink, G. Ianni, and T. Krennwallner. Combining nonmonotonic knowledge bases with external sources. In *FroCos'09*, pp. 18–42. Springer, 2009.
14 T. Eiter, M. Fink, and J. Moura. Paracoherent answer set programming. In *KR'10*, pp. 486–496. AAAI Press, 2010.
15 T. Eiter, G. Gottlob, and H. Veith. Modular logic programming and generalized quantifiers. In *LPNMR'97*, pp. 290–309. Springer, 1997.
16 W. Faber, N. Leone, and G. Pfeifer. Semantics and complexity of recursive aggregates in answer set programming. *Artif. Intell.*, 175(1):278–298, 2011.
17 M. Fitting. Enumeration operators and modular logic programming. *J. Logic Prog.*, 4(1):11–21, 1987.
18 H. Gaifman and E. Shapiro. Fully abstract compositional semantics for logic programs. In *POPL'89*, pp. 134–142, ACM, 1989.
19 M. Gelfond and V. Lifschitz. Classical negation in logic programs and deductive databases. *New Generat. Comput.*, 9:365–385, 1991.
20 S. Heymans, T. Eiter, and G. Xiao. Tractable reasoning with dl-programs over datalog-rewritable description logics. In *ECAI'10*, pp. 35–40. IOS Press, 2010.
21 T. Janhunen, E. Oikarinen, H. Tompits, and S. Woltran. Modularity aspects of disjunctive stable models. *J. Artif. Intell. Res.*, 35:813–857, 2009.
22 M. Järvisalo, E. Oikarinen, T. Janhunen, and I. Niemelä. A module-based framework for multi-language constraint modeling. In *LPNMR'09*, pp. 155–168. Springer, 2009.
23 F. Lin and Y. Zhao. ASSAT: computing answer sets of a logic program by SAT solvers. *Artif. Intell.*, 157(1-2):115–137, 2004.
24 V. Lifschitz and H. Turner. Splitting a logic program. In *ICLP'94*, pp. 23–37. MIT, 1994.
25 E. Oikarinen and T. Janhunen. Achieving compositionality of the stable model semantics for smodels programs. *Theo. Pract. Logic Prog.*, 8(5–6):717–761, 2008.
26 D. Pearce. A new logical characterisation of stable models and answer sets. In *Non-Monotonic Extensions of Logic Programming*, pp. 57–70. Springer, 1997.
27 M. Truszczynski. Reducts of propositional theories, satisfiability relations, and generalizations of semantics of logic programs. *Artif. Intell.*, 174(16–17):1285–1306, 2010.
28 J. Vennekens, D. Gilis, and M. Denecker. Splitting an operator: algebraic modularity results for logics with fixpoint semantics. *ACM Trans. Comput. Logic*, 7(4):765–802, 2006.