EVOLOG
Actions and Modularization in Lazy-Grounding Answer Set Programming

Master Thesis Proposal

Michael Langowski
Advisors: Thomas Eiter, Antonius Weinzierl

April 2022

1 Motivation and Problem Statement

Answer Set Programming (ASP) [6] is a widespread and mature formalism for declarative problem solving. It has been used to solve search- and planning problems with great success in numerous application areas. Traditionally, software systems employing an ASP-based component would use a mixture of programming languages to obtain input data, either from user input or data storage such as files or databases, transform this data into ASP *facts*, trigger an ASP *solver* and transform the *answer sets* calculated by the solver back into a (typically object-oriented) representation the application frontend can work with. The language(s) used for frontend and data transformation tasks are typically established general-purpose programming languages such as Java, Python, etc.

This kind of architecture is necessary because ASP, as defined in the current version of the ASP-Core2 Standard [3], lacks facilities to directly interact with the "external world" such as reading and writing files, reacting to user input, or sending data over a network. Despite some systems offering features to trigger actions from ASP code, e.g. ACTHEX [1] or newer clingo versions, it is still very hard to develop a complete application using only ASP as an implementation language.

The aim of this thesis is to render complex multi-language application architectures unnecessary by developing a more generally useable language (working title "EVOLOG") based on the Answer Set Programming formalism, in which it is possible to not only write the reasoning component of a larger application, but the whole application. To that end, we introduce a conservative extension of the Stable Model Semantics [10] that allows programmers to

- interact with the outside world using *actions* and *external atoms*,
- easily group and re-use partial programs using a lightweight library and modularization system.

A prototypical reference implementation of the above mentioned features should be realized as an extension of the lazy-grounding ASP solver *Alpha*.

2 Goals of the Thesis

The goal of this work is to address the following issues with as little "semantics overhead" as possible and deliver an easy-to-use programming language that can be used in most software engineering contexts.

- **Triggering actions from programs** Most program flows follow a chain of events, each a consequence of its predecessor, e.g. "If there exists a file A, read it. If reading was successful, do something with the content. If the operation succeeds, write the result to file B". It is highly desirable to be able to write this kind of program in a declarative, logic-based language that can leverage the strengths of ASP for the "business logic" part. We aim to show that our proposed action semantics can achieve this as a conservative extension of the existing stable model semantics.

- **Program Modularization** While not formally connected, triggering actions from programs and modularization (i.e. plugable and re-usable sub-programs), intuitively complement each other in our current high-level design. Introducing a simple, easy-to-use module system is therefore the second goal of this work. It is, however, secondary in priority to definition and prototypical implementation of action support and may be reduced to a technical design draft if required due to time constraints.

- **Incremental Evaluation and Lazy Grounding** Experiences from existing systems for ASP application development such as ASAP [15] or ACTHEX [1] show that, in order to achieve the evaluation performance necessary for use in real-world applications, ASP application code needs to be evaluated in an incremental fashion (rather than iteratively re-evaluating the whole program) whenever possible. The lazy-grounding architecture employed by ASP systems such as Alpha [17] offers an intuitive solution.

In addition to the three cornerstone goals outlined above, care should be taken to achieve a language suited to writing *concise, easy-to-debug programs.* A major strength of ASP is that every result (i.e. atom in an answer

set) of a program is *explainable* by a sequence of rules firing, i.e. it is (theoretically) easy to ascertain *why* something is in an answer set. This should be equally applied to actions triggered by a program, thereby offering an easy means of debugging and error recovery. Furthermore, to ease development, we aim to *provide a foundation for a rich standard library*. One of the strengths of Java is its large standard library which provides a set of tools for a number of frequently occurring scenarios. As part of an effective toolset, EVOLOG should provide a similar range of natively implemented library modules to avoid users spending time repeatedly re-implementing routine patterns.

3 Methodological Approach

To achieve the best possible results, the following methodology will be employed:

- Literature Review: Comprehensive background information needs to be gathered in order to neatly fit the language extension developed in this thesis into the existing knowledge body on logic programming.
- Precise formal definition of EVOLOG syntax and semantics
- Technical design of an EVOLOG implementation based on the existing Alpha ASP solver
- Implementation of an EVOLOG interpreter based on Alpha following said design
- Formal proof of EVOLOG being a conservative extension of the traditional Stable Model Semantics
- Evaluation of implementation based on a small application

4 Thesis Outline

4.1 Introduction

This chapter gives a high-level overview of the Answer Set Programming paradigm as well as typical application architectures involving ASP reasoning components. We describe issues with logic-based general-purpose programming languages such as Prolog (language-inherent procedurality, etc.) and introduce an EVOLOG code sample performing a simple I/O-related task, e.g. read graph data from an input file, find minimum vertex cover, write result(s) to a file.

4.2 Preliminaries

In the Preliminaries chapter, we give a detailed description of the Stable Model Semantics as defined by Gelfond and Lifschitz [10]. Since the idea for EVOLOG is to run in the context of a lazy-grounding solver, we next introduce the notion of a computation sequence [4] and the lazy-grounding solver architecture with an emphasis on the Alpha system. We move on to cover extensions of ASP relevant to the topic of this thesis, namely aggregates, external atoms and select modularization approaches. The section is concluded by a discussion of "evaluation-shortcuts" based on stratifiable program parts and splitting sets, which provide the formal foundation for EVOLOGs action semantics.

4.3 EVOLOG Language Specification

4.3.1 Syntax

Before defining the formal semantics for each language feature, this chapter defines syntax constructs accompanied by short and intuitive explanations and code samples. The items covered are:

- Rule heads that trigger actions
- Literals that instantiate modules (i.e. solver-calls to solve sub-programs) with a set of facts. We will explore mechanisms to pass sets of data

items (rather than just single terms) into modules on instantiation - Adding a list data type to the language seems a viable option.

- Rule heads that trigger runtime errors ("exceptions") as an easy way of reporting and reacting to errors arising from actions.

4.3.2 Semantics

The cornerstone of this chapter is a sound definition of how actions work in EVOLOG. The current vision is to permit rules deriving actions only in contexts, i.e. program parts, where the head of an "action rule" is *guaranteed to be in an answer set*. This ensures that execution (i.e. solving) of unsatisfiable programs has no effect on the outside world as well as traceability of program execution. This (compared to ACTHEX) restrictive stance on semantics also has the additional advantage that actual execution of actions can happen directly during program evaluation, while still preserving correct execution sequence.

Modules are to be treated as a specialized type of external atom, i.e. oracle functions that map from an input set of ground terms to multi-sets of atoms (the answer sets of the respective sub-program with its input as a fact).

More generally, we aim to characterize the notion of an EVOLOG model. Loosely inspired by the concept of Monads in functional languages, we view actions as functions mapping one (i.e. previous or input) world state to another (resulting or output) world state, thereby making effects on the outside world transparent to and accessible from the program. Based on that intuition, an EVOLOG model is a classical model accompanied by an interpretation function ("frame") for all action functions used in the program. Furthermore, we define the semantics of modules as a special form of external atom (i.e. oracle function).

4.4 Implementation

A prototypical reference implementation of the EVOLOG language standard should be implemented as an extension of the existing Alpha ASP

Solver. Existing static program analysis needs to be enhanced to facilitate correct handling of actions (see Semantics). It seems reasonable to assume that actions can only exist in program parts that are stratified. This assumption needs to be verified and an appropriate program evaluation logic implemented. Note that Alpha's existing CDNL-driven solving approach must be kept for non-stratified program parts to avoid a negative impact on solving performance.

4.5 Verification and Evaluation

Since EVOLOG is meant to be a conservative extension of the ASP Core2 language, it needs to be verified that EVOLOG programs that do not use any of the newly introduced features comply to the ASP-Core2 standard.

Furthermore, to demonstrate the practicability of EVOLOG, a small application example should be developed. We currently envision a music composition support system where EVOLOG is used to generate accompanying chords to a given melody similar to [2]. In contrast to similar existing systems, this application should be written exclusively in EVOLOG, i.e. without using procedural languages for input- and output transformation tasks like XML parsing- and writing.

5 State of the Art

This work aims to blend action support with modularization in the context of lazy-grounding ASP solving - all three of these areas have seen a substantial amount of research in the past.

Both Clingo [8] and DLVHEX, through the ACTHEX [1] extension, offer their own flavours of support for triggering actions from programs. While Clingo does not directly support actions as a dedicated feature - and therefore offers no strictly enforced semantics for this - similar behavior can be achieved using external functions and the reactive solving features first introduced in oClingo [7]. ACTHEX has thoroughly defined semantics for actions. In the ACTHEX model, answer set search and action execution are separate steps, where executability of actions is only determined after an-

swer sets are calculated. While this gives users a high degree of flexibility in working with actions, it does not directly lend itself to the idea of a general purpose language where program behavior may be influenced by continuous two-way communication between a program and its environment.

With regards to Modularity, i.e. the process of "assembling" an ASP program from smaller building blocks (i.e. modules), a comprehensive semantics for so-called *nonmonotonic modular logic programs* has been introduced in [12] and [5]. While it does not impose any restrictions on language constructs used in modules and recursion within and between modules, it also comes with rather high computational complexity and no easily available implementations so far. A more "lightweight" approach to modularization are *Templates* [11]. As the name implies, this purely syntactic approach aims to define isolated sub-programs that can be generically used to avoid code duplication throughout an application and is conceptually similar to the well-known generics in object-oriented languages such as C++ and Java. Templates are rewritten into regular ASP rules using an "explosion" algorithm which basically "instantiates" the template by generating the needed body atoms (and rules deriving them) wherever templates are used. While easy to implement and flexible, a potential disadvantage of this concept is that - due to its purely syntactical nature - programmers need to be on the watch for potential bugs arising from unintended cyclic dependencies or recursive use of templates (leading to potential non-termination of the explosion algorithm) themselves. Yet another powerful toolset for modular application development is provided by Clingo's *multi-shot-solving* [9] features. Clingo allows for parameterized sub-programs which are then repeatedly grounded in a process that is conceptually similar to the notion of module instantiation in [14] and solved during solving of the overall program. However, as this "contextual grounding" needs to be programmatically controlled by an external application through Clingo's API, the inherent flexibility and usefulness for incremental solving of this approach is counterweighed by a high level of proficiency with and knowledge of the Clingo system necessary to leverage these capabilites. The concept of *lazy grounding*, i.e. interleaving of the - traditionally sequential - grounding and solving steps present in most prevalent ASP solvers, is relatively new. It has been spearheaded by the GASP [4] and ASPERIX [13] solvers

which avoid calculating the full grounding of an input program by performing semi-naive bottom-up evaluation along the input's topologically sorted (non-ground) dependency graph. While efficient in terms of memory use, this approach cannot stand up to the solving performance of systems like DLV or Clingo which employ their knowledge of all possible ground rules to perform conflict-driven nogood learning (CDNL) as part of their solving algorithm to great effect. Alpha [17], a more recent lazy-grounding solver, aims to bridge this gap in performance by employing CDNL-style solving techniques [16] incrementally on partially ground program parts as part of its central ground-and-solve loop.

Conceptually, the common ingredient linking the - on first glance not directly connected - areas of actions in ASP, program modularization, and lazy grounding is a need for detailed static program analysis prior to solving, be it to detect potentially invalid action sequences, calculate module instantiation orders, or for up-front evaluation of stratified program parts in a lazy-grounding context. In addition, both actions and modularization can greatly benefit from - or even depend on - incremental evaluation facilities of a solver for efficient operation. Since lazy-grounding by its very definition embodies an incremental evaluation approach, it seems only natural to incorporate actions and modularization into a lazy-grounding solver's input language in order to provide ASP programmers with a powerful tool for application development. Alpha, with its good solving performance compared to other lazy-grounding systems, support for a large part of the current ASP-Core2 language standard, and active development status, appears the natural choice as the technical backbone of this work.

6 Relevance to the Logic and Computation Curriculum

This work contributes to the areas of Knowledge Representation and Reasoning, Logic Programming, and Software Engineering, all of which are at the heart of the Logic and Computation Curriculum.

- The topic of this proposed thesis is most directly connected to the

courses of the curriculum modules "Knowledge-based Systems" and "Knowledge Representation and Artificial Intelligence", where the area of Answer Set Programming is covered in courses such as "184.730 Knowledge-based Systems", "184.205 Processing of declarative Knowledge" and "192.084 Practical Applications of Answer Set Programming"

- From a Software Engineering point of view, this thesis - having the goal of furthering development of a programming language - naturally relates to courses from the module "Programming Languages and Verification", especially courses like "184.703 Program Analysis", "188.409 Requirements Engineering and Specification" or "183.290 Software Testing".

- Last but not least, also foundational courses like "185.291 Formal Methods in Computer Science" and "104.271 Discrete Mathematics" are worth mentioning here, since the general concepts taught there will be needed for the more formal parts of this thesis.

References

[1] Selen Basol, Ozan Erdem, Michael Fink, and Giovambattista Ianni. Hex programs with action atoms. In *Technical Communications of the 26th International Conference on Logic Programming*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2010.

[2] Pedro Cabalar and Rodrigo Martín. haspie-a musical harmonisation tool based on asp. In *EPIA Conference on Artificial Intelligence*, pages 637–642. Springer, 2017.

[3] Francesco Calimeri, Wolfgang Faber, Martin Gebser, Giovambattista Ianni, Roland Kaminski, Thomas Krennwallner, Nicola Leone, Marco Maratea, Francesco Ricca, and Torsten Schaub. Asp-core-2 input language format. *Theory and Practice of Logic Programming*, 20(2):294–309, 2020.

[4] Alessandro Dal Palu, Agostino Dovier, Enrico Pontelli, and Gianfranco Rossi. Gasp: Answer set programming with lazy grounding. *Fundamenta Informaticae*, 96(3):297–322, 2009.

[5] Minh Dao-Tran, Thomas Eiter, Michael Fink, and Thomas Krennwallner. Modular nonmonotonic logic programming revisited. In *International Conference on Logic Programming*, pages 145–159. Springer, 2009.

[6] Thomas Eiter, Giovambattista Ianni, and Thomas Krennwallner. Answer set programming: A primer. In *Reasoning Web*, volume 5689 of *Lecture Notes in Computer Science*, pages 40–110. Springer, 2009.

[7] Martin Gebser, Torsten Grote, Roland Kaminski, and Torsten Schaub. Reactive answer set programming. In *International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 54–66. Springer, 2011.

[8] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. Clingo= asp+ control. *arXiv preprint arXiv:1405.3694*, 2014.

[9] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. Multi-shot asp solving with clingo. *Theory and Practice of Logic Programming*, 19(1):27–82, 2019.

[10] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *ICLP/SLP*, pages 1070–1080. MIT Press, 1988.

[11] Giovambattista Ianni, Giuseppe Ielpa, Adriana Pietramala, Maria Carmela Santoro, and Francesco Calimeri. Enhancing answer set programming with templates. In *NMR*, pages 233–239, 2004.

[12] Thomas Krennwallner. *Modular nonmonotonic logic programs*. PhD thesis, Technical University of Vienna, 2018.

[13] Claire Lefevre and Pascal Nicolas. The first version of a new asp solver: Asperix. In *International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 522–527. Springer, 2009.

[14] Emilia Oikarinen and Tomi Janhunen. Achieving compositionality of the stable model semantics for smodels programs. *arXiv preprint arXiv:0809.4582*, 2008.

[15] Peter Schüller and Antonius Weinzierl. Answer set application programming: a case study on tetris. In *ICLP (Technical Communications)*, 2015.

[16] Antonius Weinzierl. Blending lazy-grounding and cdnl search for answer-set solving. In *International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 191–204. Springer, 2017.

[17] Antonius Weinzierl, Bart Bogaerts, Jori Bomanson, Thomas Eiter, Gerhard Friedrich, Tomi Janhunen, Tobias Kaminski, Michael Langowski, Lorenz Leutgeb, Gottfried Schenner, et al. The alpha solver for lazy-grounding answer-set programming. 2019.