The Alpha Solver for Lazy-Grounding
Answer-Set Programming

Antonius Weinzierl¹, Bart Bogaerts², Jori Bomanson³, Thomas Eiter¹, Gerhard Friedrich⁴, Tomi Janhunen⁵,⁶, Tobias Kaminski¹, Michael Langowski¹, Lorenz Leutgeb¹, Gottfried Schenner⁶, and Richard Taupe⁴,⁶

¹ Insitute of Logic and Computation, Vienna University of Technology, Austria
² Department of Computer Science, Vrije Universiteit Brussel (VUB), Belgium
³ Department of Computer Science, Aalto University, Finland
⁴ Department of Applied Informatics, Alpen-Adria-Universität Klagenfurt, Austria
⁵ Information Technology and Communication Sciences, Tampere University, Finland
⁶ Corporate Technology, Siemens AG Österreich

weinzierl@kr.tuwien.ac.at

Abstract. The grounding bottleneck is a longstanding issue of Answer-Set Programming (ASP), a well-known Logic Programming formalism widely used for declarative problem solving. Lazy grounding as realized by the recent ALPHA system avoids this grounding bottleneck but faces new challenges that are genuine to lazy grounding. This article first gives an overview of lazy-grounding ASP solving by ALPHA and provides information on how to obtain and use the system. It then presents research issues raised by lazy-grounding ASP and overviews those which have been addressed already.

1 Introduction

Answer-Set Programming (ASP) [4, 19, 28, 34, 35] is a well-known Logic Programming formalism, widely used for declarative problem solving in diverse areas, ranging from planning, optimization, and commonsense reasoning to explanation finding. The success of ASP is rooted in efficient solvers, due to which ASP is applied fruitfully in a broad range of applications, from NASA’s Space Shuttle [33, 48], to planning in the automotive industry [32], configuration [24], etc. [9, 52]. Despite this success, ASP solving has always been hampered by the grounding bottleneck, which is an inherent drawback of the traditional ground-and-solve evaluation of ASP that is employed by many of the most prominent ASP solvers till today. First, a non-ground (i.e., first-order) ASP program is fully grounded upfront to obtain a variable-free (i.e., propositional) program. In a second step, the answer sets of the grounded program are computed using efficient, propositional solving techniques adapted or inspired from SAT solving [1, 29, 41]. As the grounded program may be exponentially larger than the original ASP program, the ensuing memory blow-up renders the ground-and-solve approach widely inapplicable for entire classes of programs.
Lazy grounding avoids upfront grounding and the grounding bottleneck by interleaving the solving and the grounding phases. While ground-and-solve systems have to ground all rules that may fire in any state the solver possibly reaches, in lazy grounding only those rules are grounded that may fire in the current state of the solver. If the solver recognizes that some part of the search space contains no answer sets, lazy-grounding can omit grounding the rules that only fire in that part. There are several systems for lazy-grounding ASP solving available, namely GASP [49], ASPeriX [38–40], Omiga [13, 57], and ALPHA [58], which are all based on the notion of a computation sequence [43]. Unfortunately, lazy-grounding cannot be simply put on top of the traditional state-of-the-art techniques for efficient ASP solving, since one of their core-assumptions, that all relevant ground rules are known, does not apply to lazy-grounding. The latter thus opens many issues and novel research questions, some of which have been closed by transferring techniques from the ground-and-solve approach to the lazy-grounding setting. In particular, ALPHA is the first system to combine lazy-grounding with efficient search techniques from traditional ground-and-solve systems, such as conflict-driven learning and watched-literals propagation. Although its search performance does not yet match the best possible with ground-and-solve systems, ALPHA is a big improvement compared to other lazy-grounding systems and offers the following features:

- computation of answer sets without the need for upfront grounding, which avoids the grounding bottleneck of ASP and thus allows to solve ASP programs where traditional solvers run out of memory;
- a combination of lazy-grounding with conflict-driven learning and other techniques for efficient (propositional) ASP solving;
- first-order normalizations to evaluate aggregates with ALPHA, while its core works with normal rules;
- non-ground justifications for atoms, to avoid a genuine problem of lazy grounding where the search may get stuck if not all ground rules that potentially can derive an atom are known;
- heuristics specifically tailored for lazy grounding;
- an integration with the HEX framework;
- a free and open-source implementation in Java.

Overall, ALPHA already strikes a good balance between efficient grounding and solving, while there are several open problems with potential for future research. In this article we review the basics underlying ALPHA, present research advances to improve lazy-grounding ASP solving, and discuss open research questions. The active development of ALPHA is continuing and we would welcome new users, feedback on the software, and collaboration on open issues.

The remainder of this article is structured as follows. Section 2 shows where to get and how to run ALPHA, while Section 3 gives an overview of the system and its components. Section 4 introduces research issues related to lazy grounding that already have been addressed, whereas Section 5 presents ongoing and open issues. Section 6 considers related work and, finally, Section 7 concludes this article.
2 Obtaining and Running Alpha

The Alpha system and its source code are publicly available on GitHub.\(^1\) Alpha is actively developed and new binary releases are made available in irregular intervals. All releases can be downloaded in compiled form as Java Archive (JAR) files. These files can be run on all major operating systems via the Java runtime (version 8 or higher) by executing `java -jar alpha-bundled.jar`. Invoking Alpha without any arguments like this will print an overview of arguments and exit. Building the newest version from source is possible using the automated build system Gradle\(^2\) and detailed instructions are available on GitHub.\(^1\)

Example 1. Let `encoding.lp` be a problem encoding in ASP and `inst43.lp` contain an instance, i.e., a number of facts. Then Alpha can be instructed to search the first 5 answer sets as follows:

```
java -jar alpha-bundled.jar -n 5 -i encoding.lp -i inst43.lp
```

One can also specify parts of the input program directly from the command line, which is useful for setting parameters like domain size or bounds of an ASP program.

Example 2. Assume the following ASP program to compute Fibonacci numbers up to a given number is given in a file `fib.lp`.

```
fib(0,0). fib(1,1).
fib(N,F) :- fib(N1,F1), fib(N2,F2), N1=N-1, N2=N-2, F = F1+F2, N=0..U, upto(U).
```

The command line to start Alpha and compute Fibonacci numbers up to 22 is:

```
java -jar alpha-bundled.jar -i fib.lp -str "upto(22)."
```

The input language currently accepted by Alpha is a subset of the ASP-Core-2 input language\(^1\) including function symbols and interval terms. Since the core of Alpha works with normal rules, we employ normalizations to transform programs using richer syntax into normal rules. At the time of writing, a growing list of supported constructs includes: choice rules without bounds, sum and count aggregates with lower bounds, and arithmetic terms in comparison relations. Full support for ASP-Core-2 is planned and ongoing. As usual in ASP, rules must be safe, i.e., every variable of a rule must also occur in its positive body.

Comments and suggestions, as well as contributions via GitHub are very much encouraged and welcomed.

3 Overview of the Alpha System

Alpha follows the notion of ASP computation for lazy-grounding and combines it with the traditional techniques for efficient ASP evaluation, where a dedicated

\(^1\) https://github.com/alpha-asp/alpha
\(^2\) https://gradle.org/
solving component employs SAT-inspired techniques to find answer sets of a set of ground nogoods and another component produces ground nogoods from the input ASP program. Different from traditional ASP solving, however, these components interact in a cycle and the algorithms in the solving component are aware of the fact that they see only a part of the full grounding of the input program.

Figure 1 gives an overview of the Alpha system and its components. There are two major components: the Grounder and the Solver. The Grounder first parses and normalizes the input program and then grounds it lazily. The Solver uses techniques for efficient ASP solving, conflict-driven learning, watched-literals propagation, and heuristics as used by state-of-the-art ASP systems, but customized for lazy grounding [58]. At startup, the Grounder derives ground rules that are applicable under the given input facts, turns them into nogoods (i.e., constraints that must be satisfied) and hands the latter on to the Solver. The Solver in turn employs a CDCL-style algorithm to derive (or extend) a partial assignment, which is then given to the Grounder to obtain further applicable ground rules, whose nogood representation is again given to the Solver.

4 Research Topics Addressed

Since 2016, when work on Alpha started, we have encountered a number of research issues and we have achieved significant progress on multiple fronts that we summarize below.

Efficient Propagation. A key issue for ASP solving is propagation in CDCL-based search; as this is frequently executed, much effort has been spent to come
up with an efficient solution. The Alpha system uses three truth values (must-be-true in addition to true and false) and therefore implements a generalization of the well-known two-watched literals (2WL) scheme commonly employed in SAT and ASP solving, which we call the three-watched literals scheme [42]. The latter introduces notions of weakly unit nogoods (propagating false or must-be-true) and strongly unit nogoods (which must have a designated head literal for propagating any truth value). Here, propagation from must-be-true to true always occurs by the head of a nogood, which requires a third watched literal. The solver maintains a watch structure for every atom, which points to the nogoods where the respective atom is being watched. This watch structure then is used for propagation in computation towards a least fixpoint.

Aggregate Rewriting. Recently, the ASP-Core-2 support in Alpha broadened to include threshold conditions on the minimum, count, and sum of terms via the powerful and highly expressive language construct of aggregates. In appreciation of the integral role of aggregates in practice and research [53, 2, 23], Alpha delivered monotone count and sum aggregate support to the lazy-grounding ASP realm [7, 8]. The approach instantiates a novel framework of lazy normalization in translating aggregates to efficient first-order normal-rule encodings. The approach overcomes key challenges posed by the lazy grounding of aggregates: it hides the complexity of aggregates from the core of the solver, which already faces the tremendous challenge and complexity of both grounding and solving. Moreover, efficient lazy grounding precludes upfront counting and ordering of predicate instances. Yet, usual normalization depends crucially on both abilities [6]. This dilemma motivated the innovation of a novel language primitive that exposes a lazily determined order of atoms and enables asymptotically more concise normalizations than previously known on the first-order level [50].

Justifications. Whenever Alpha arrives in a situation where a true atom, say \( p \), is unjustified (in the terminology from above, this means that \( p \) is must-be-true but not true) and no further choices or propagations are possible, the basic Alpha algorithm resorts to chronological backtracking. This turned out to be problematic in several applications; to avoid this, we devised a method that analyses why \( p \) is not justified and subsequently learns a new clause that intuitively states that whenever \( p \) is true, the discovered reason for \( p \) being unjustified should be invalidated [5]. Formally this analysis builds on the recently introduced theory of justifications [14]. In practice, our algorithm — inspired by partial evaluation techniques [37] — performs a top-down analysis of the logic program starting from \( p \) to find all literals that (directly or indirectly) blocked \( p \) from being justified, either by falsifying a rule body of a rule known to the solver that could contribute to a justification of \( p \) or by blocking a rule from being grounded. Preliminary experiments are promising and show that up to exponential speed-ups can be gained and that on previously existing benchmarks, the justification analysis eliminates a need for addition of redundant constraints [5].
Search Heuristics. Search heuristics steer solvers through the search space and may shorten the search by a large amount. ALPHA takes ideas from domain-independent search heuristics like VSIDS [47] and BerkMin [36], which were originally developed for SAT but are also successfully employed by ASP solvers (such as clasp [31] and WASP [1]). These heuristics take into account how often and how recently a propositional variable contributed to a conflict.

A direct application of BerkMin or VSIDS to lazy-grounding systems like ALPHA is problematic because their solving algorithms are quite different from those in ground-and-solve systems. A major difference is that not all ground rules, and consequently not all ground atoms, are known to a lazy-grounding solver at any time. A further difference is that while a traditional ASP solver can choose any atom to guess on, ALPHA only guesses on atoms representing bodies of rules that almost fire in the sense that the positive body is true already and the negative body is not falsified. Therefore, for ALPHA a set of novel domain-independent heuristics has been developed [56]. First benchmarks are promising but also indicate the need for further study.

Integration with HEX. The _hex-formalism_ extends ASP by allowing a bidirectional exchange between programs and external computation sources, which are interfaced via so-called _external atoms_ [18]. For instance, an external atom \&onlineWeather[loc](X) may query an online weather service for the weather report for all locations in the extension of the loc predicate. In practice, external atoms are realized by C++ or Python plug-ins in an API-style fashion, and the formalism has been applied to a wide range of areas ranging from Semantic Web applications to route planning [20]. A longstanding issue regarding HEX-evaluation has been the grounding of programs containing nonmonotonic external atoms that introduce new constants by so-called _value invention_ as such atoms need to be evaluated under exponentially many inputs during grounding. This makes the grounding for HEX even more challenging than for ordinary ASP.

ALPHA enabled the integration of grounding and solving of HEX-programs such that new output constants of external atoms can be generated on-the-fly during solving [21]. For this, ALPHA has been integrated as a backend-solver into the DLVHEX system, which resulted in a novel algorithm for evaluating HEX-programs based on lazy grounding. Using ALPHA solved the issue with respect to grounding nonmonotonic external atoms and for grounding-intense programs, a clear advantage of the ALPHA-based algorithm could be shown in practice.

5 Ongoing and Future Work

The development of ALPHA is ongoing and researchers from multiple universities and the industry are collaborating to further improve the state of lazy-grounding ASP solving. We present some of the currently ongoing work related to ALPHA below.

Grounding Strategies. ALPHA was equipped with state-of-the-art heuristics successfully employed by other ASP solvers, namely MOMs [51] for initialization
of heuristic scores and VSIDS [47] for their dynamic modification. Both are implemented in a similar fashion as in clasp [31]. However, the performance improvement by those heuristics was much smaller than expected, because lazy grounding provides a too narrow view of the search space for such heuristics to perform adequately. This is a novel challenge for ASP solving, which traditional ground-and-solve ASP solvers did not have to face.

So far ALPHA runs a very restrictive grounding strategy in order to save maximum space, which results in non-optimal search performance as state-of-the-art search heuristics are left mostly blind because they only consider the grounded information. Thus we investigated more permissive lazy-grounding strategies that ground more than what is absolutely necessary. They produce ground rules earlier, which informs search procedures better about the problem at hand. For more details see the upcoming paper [55].

Domain-Specific Heuristics. A major advance in solving industrial configuration problems with ASP can be achieved with domain-specific heuristics (cf. [16, 26, 30]), which allow to use heuristics that are designed specifically for one given application domain. Given, e.g., a bin-packing problem, a domain-specific heuristic for the solver could be to pick the biggest item not yet placed and put it in a fitting bin with the least space remaining (i.e., a best-fit decreasing heuristic).

In [54], a novel semantics for heuristic directives in ASP is presented that allows declarative specification of domain-specific heuristics where default negation inside heuristic conditions holds for \textit{false} and \textit{currently unassigned} atoms. This allows for a natural and declarative formalisation of many domain-specific heuristics, e.g., in a bin-packing encoding to refer to items not yet placed or the total weight of items already placed in the partial assignment. The implementation of such heuristics in ALPHA is currently ongoing.

Partial Evaluation. Traditional ground-and-solve ASP systems evaluate the definite part of the given input program already in the grounding phase in order to reduce the instance of the subsequent solving phase. In ALPHA there is ongoing work to realize partial evaluation in a similar way, based on the part of the input that can be stratified [3] and which does not depend on any choices to be made by the solver. For this, ALPHA analyses the dependency graph of the input program on a predicate level, identifies strongly connected components (SCCs), and then evaluates the acyclic part that does not depend on any cycles. This approach is not as general as, e.g. the one in DLV [22] which considers dependency on ground rules; however, it enables a subsequent search strategy based on SCCs similar to ASP\textsc{erixa}.

Future Work. A medium-term goal for ALPHA is to support the full ASP-Core-2 input language, which includes weak constraints, optimization, and disjunction. A further such goal is to transfer all major techniques for efficient answer-set solving, like learned clause deletion, rapid restarts, etc., to the lazy-grounding setting. For many of them, lazy-grounding raises novel research questions and issues that do not exist if the full grounding is available. They pose completely
new challenges and open avenues for novel solving techniques that make use of
the first-order representation of the program. We will continue to investigate
these avenues and invite the community to join our efforts.

6 Related Work

Alpha is the latest in a line of lazy-grounding ASP solvers being based on the
notion of a computation sequence [43]. GASP [49] and ASPereRX [39, 40, 38]
were the first lazy-grounding ASP solvers that implemented this notion. Given
a (partial) interpretation, finding a ground instance of a non-ground rule, i.e.,
grounding the rule lazily, is a task that is well-known to be NP-complete. For
this reason, lazy grounding inside GASP was encoded as a constraint problem,
while ASPereRX used a semi-naive grounding approach. The later Omiga solver
[13, 57] used a RETE network [25] to speed-up this task but otherwise followed
the idea of a computation sequence quite closely. As noted in [38], however, semi-
naive grounding seems to be sufficiently fast for lazy-grounding ASP solving.

Computation sequences have also been implemented on top of special hard-
ware, in particular graphics cards to make use of their massive parallelization
[17]. This approach, however, uses no lazy grounding but the traditional full
upfront grounding.

Another approach to avoid grounding is based on a top-down query-driven
method to evaluate normal logic programs in a way similar to Prolog and with
negation as failure under the stable model semantics [44]. Galliwasp [45] is the
propositional stepping stone for the s(ASP) system [46], which implements this
query-driven ASP evaluation. Similar to Alpha, it does not require a finite
grounding, but the s(ASP) prototype, in contrast, is not designed to be compet-
itive in terms of speed and no benchmark results for search performance beyond
trivial problem instances have been reported.

Goal-driven lazy grounding is also realized in the Lazy-MX [12] system, which
achieves efficient solving performance for its language of \textit{FO(ID)} that essentially
corresponds to the generate-define-test fragment of ASP [15].

Another way to avoid the grounding bottleneck of ASP is to outsource
difficult-to-ground parts of a problem specification. Recent applications with
CLINGO [27] represent parts of a problem in an extra theory and employ the
ASP modulo theories approach to combine the outsourced theory reasoning with
the parts specified in ASP. This is akin to SAT modulo theories (SMT). The ob-
vious drawback of such an approach, however, is that the problem now has to
be specified in two distinct formalisms and for solving, these formalisms must
be combined again in a suitable manner.

7 Conclusion

We have presented the Alpha system for answer-set solving, which avoids the
grounding bottleneck using a lazy-grounding approach. While on programs whose
full upfront grounding easily fits into memory the solving performance of Alpha
is not yet on a par with the best performance achieved by traditional ground-and-solve ASP systems, ALPHA offers competitive solving performance on programs where grounding is demanding, because it implements the most important techniques for efficient ASP solving already. By that, ALPHA makes ASP solving feasible for entirely new classes of programs and new application domains from industry.

The system is ready to be used and ongoing development aims at supporting the full ASP-Core-2 input language as well as improved solving performance. ALPHA and its source code are freely available on GitHub.\(^3\) We are happy to receive bug reports and would very much welcome potential future contributors and collaborators.

Acknowledgements

Research on ALPHA has been supported by the Austrian Science Fund (FWF) projects P27730 and W1255, the Academy of Finland, project 251170, and the research project DynaCon (FFG-PNr.: 861263), which is funded by the Austrian Federal Ministry of Transport, Innovation and Technology (BMVIT) under the program “ICT of the Future” between 2017 and 2020 (for more information see https://iktderzukunft.at/en/).

References


\(^3\) https://github.com/alpha-asp/alpha
27. Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, Torsten Schaub, and Philipp Wanko. Theory solving made easy with clingo 5. In