Aalto University

OPEN ACCESS

This is an electronic reprint of the original article.
This reprint may differ from the original in pagination and typographic detail.

Janhunen, Tomi

Cross-Translating Answer Set Programs Using the ASPTOOLS Collection

Published in:
KUENSTLICHE INTELLIGENZ

DOI:
10.1007/s13218-018-0529-9

Published: 14/05/2018

Document Version
Peer reviewed version

Please cite the original version:
Janhunen, T. (2018). Cross-Translating Answer Set Programs Using the ASPTOOLS Collection. *KUENSTLICHE INTELLIGENZ*, *32*(2-3), 183-184. https://doi.org/10.1007/s13218-018-0529-9

This material is protected by copyright and other intellectual property rights, and duplication or sale of all or part of any of the repository collections is not permitted, except that material may be duplicated by you for your research use or educational purposes in electronic or print form. You must obtain permission for any other use. Electronic or print copies may not be offered, whether for sale or otherwise to anyone who is not an authorised user.

Künstliche Intelligenz manuscript No.
(will be inserted by the editor)

Cross-Translating Answer Set Programs

Using the ASPTOOLS Collection

Tomi Janhunen

Received: December 18, 2017 / Accepted: date

Abstract One viable way of implementing answer set programming (ASP) is to compile (ground) logic programs into other formalisms and to use existing solver technology to compute answer sets. In this article, we present an overview of translators used for such compilations, targeting at other solving paradigms such as Boolean satisfiability checking, satisfiability modulo theories, and mixed integer programming. Borrowing ideas from modern compiler design, such translators can be systematically developed in stages so that the details of the target formalism can be incorporated at the last step of the translation. In this way, the resulting translators realize a cross-compilation framework for answer set programs, coined as *cross-translation* in this article.

Keywords Cross Compilation · Rules · Answer Sets · Constraints · Satisfiability

1 Introduction

Answer set programming (ASP) [2] is a declarative programming paradigm where computation is organized in two main steps. The first concerns the instantiation of first-order rules in the input program. The goal of the second step is to compute *answer sets* for the *ground* logic program that resulted from the first step. As regards the search for answer sets, native answer set *solvers* have been implemented for the task. The other mainstream approach counts on translating ground rules into other kinds of constraints and harnessing off-the-shelf solver technology to perform the search. In this way, it is possible to use ASP for *knowledge representation* as before—encoding problems as logic programs—but a wide variety of back-end solvers can be responsible for the actual *reasoning* phase. We expect that this scenario is also beneficial for the development of native answer set solvers in the long run, since any breakthroughs in other solver technology can readily boost ASP via translations.

Over the years, we have developed a number of translations from ASP to *satisfiability checking* (SAT) [4, 5], SAT *modulo theories* (SMT) [6, 9], and *mixed integer programming* (MIP) [7]. Some of these target formalisms presume the *normalization* of extended rule types before translation [1, 5], if such rules are not supported by back-end solvers. These translation and normalization schemes have been implemented as stand-alone translators, known as the LP2* family under the ASPTOOLS[1] collection. The translators of this family operate quite similarly but vary when it comes to producing the actual translation. This can be deemed as an issue from development perspective since any new translational ideas have to be incorporated into a number of tools. The situation can be relieved using a more recent SMT-style translation of ASP, namely into SAT *modulo acyclicity* [3], which defers the details of constraints supported by back-end solvers until the last step of translation.

In this article, we use traditional compiler architectures and the idea of *cross compilation* as starting points when presenting a similar framework for the translation of an-

Research supported by the Academy of Finland (under grant #251170).

T. Janhunen
Department of Computer Science, Aalto University, Finland
E-mail: Tomi.Janhunen@aalto.fi

[1] `http://research.ics.aalto.fi/software/asp/`

swer set programs (Section 2). In particular, cross compilers may generate executable code for *multiple platforms* from essentially the *same source*. They can also support multiple programming languages. For instance, the GNU C compiler GCC[2] has an architecture enabling all of these features:

1. The **Front End** of GCC forms an *abstract syntax tree* for an input program written either in C, C++, or Java.
2. The **Middle End** of the compiler is responsible for *code optimization* where the program is treated in a structured *intermediate representation* called GIMPLE [8].
3. The **Back End** generates *object code* for the respective processor platform such as x68-64, ARM, or z8000.

2 Cross-Translation Framework for ASP

Following the structure above, we present an analogous architecture for the *cross translation* of answer set programs, as illustrated in Fig. 1. It is natural that grounding plays the role of **front-end** translation that can be achieved by grounders like GRINGO or LPARSE. As a result, the input program is represented in an *intermediate format* such as the historical SMODELS format. The *interoperability* of various tools builds heavily on the existence of such a format, enabling the representation of the results of various *translation steps*. In the **middle-end** phase, some rewriting of the logic program may be in order. Most notably, the *normalization* of extended rule types is needed for any translations towards pure SAT [1], as implemented by the tool LP2NORMAL2.

The final **back-end** stage of translation is based on a lightweight SMT-style extension of SAT, viz. SAT *modulo acyclicity* (ACYC) [3], which lends itself for the formalization of various of topological structures such as trees, forests, DAGs, and chordal graphs. The back-end translation consists of three steps. The first *instruments* a logic program with special (acyclicity) atoms based on its dependency graph. The second forms Clark's completion for the program, capturing *supported models* modulo acyclicity. The third step, as implemented in two separate translators called LP2SAT and ACYC2SOLVER, emits the required *formalism-specific* constraints. Many output formats are covered: DIMACS, SMT-LIB 2.0 (QF_IDL and QF_BV), or CPLEX, to be fed to the respective back-end solvers (see Fig. 1).

3 Conclusion

Translation-based ASP combines the expressive power of ASP with the performance of existing solver technology. In

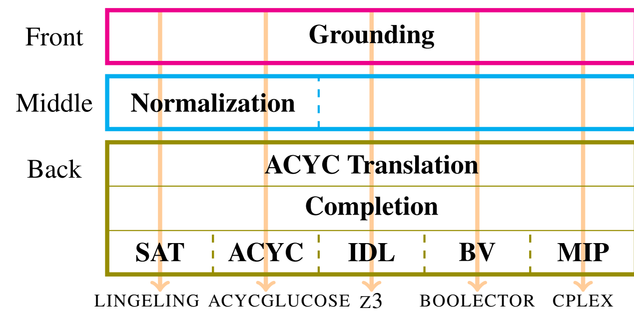

Fig. 1 Architecture for Cross-Translating Answer Set Programs

this setting, cross translation means that formalism-specific aspects are realized at the last step translation. This sets special requirements on intermediate formats to enable the interoperability and integration of tools. As regards future work, we plan to introduce the ASPIF format, deployed by the latest GRINGO and CLASP versions, in our framework.

[2] en.wikibooks.org/wiki/GNU_C_Compiler_Internals

References

1. Bomanson J, Janhunen T (2013) Normalizing cardinality rules using merging and sorting constructions. In: Proceedings of LPNMR 2013, pp 187–199
2. Brewka G, Eiter T, Truszczyński M (2011) Answer set programming at a glance. ACM Comm 54(12):92–103
3. Gebser M, Janhunen T, Rintanen J (2014) Answer set programming as SAT modulo acyclicity. In: Proceedings of ECAI 2014, pp 351–356
4. Janhunen T (2004) Representing normal programs with clauses. In: Proceedings of ECAI 2004, pp 358–362
5. Janhunen T, Niemelä I (2011) Compact translations of non-disjunctive answer set programs to propositional clauses. In: Gelfond Festscrift, LNCS 6565, pp 111–130
6. Janhunen T, Niemelä I, Sevalnev M (2009) Computing stable models via reductions to difference logic. In: Proceedings of LPNMR 2009, pp 142–154
7. Liu G, Janhunen T, Niemelä I (2012) Answer set programming via mixed integer programming. In: Proceedings of KR 2012, pp 32–42
8. Merrill J (2003) GENERIC and GIMPLE: A new tree representation for functions. In: GCC Developers Summit, pp 171–180
9. Nguyen M, Janhunen T, Niemelä I (2013) Translating answer-set programs into bit-vector logic. In: Proceedings of INAP 2012, pp 95–113