ASGL: Argumentation Semantics in Gecode and Lisp

Kilian Sprotte

University of Hagen,
Knowledge-Based Systems,
Universitätsstr. 1,
58097 Hagen, Germany
kilian.sprotte@gmail.com

Abstract. ASGL is a solver for argumentation semantics, capable of answering queries with respect to grounded, complete, preferred, and stable semantics. It is built based on GECODE, a generic CSP solver. ASGL itself is mainly written in Common Lisp. This paper presents a description of its system components and, for a selection of the computational tasks, provides details on how ASGL approaches them.

1 Introduction

ASGL [6] is a solver for argumentation semantics. Given an argumentation framework $AF = (Ar, att)$, answers to queries with respect to grounded, complete, preferred, and stable semantics can be computed. Specifically, for each semantics, ASGL allows to enumerate one or all extensions and to report on the status of an argument, while taking either a credulous or a skeptical point of view [4].

ASGL is mainly written in Common Lisp and CLOS [7], with some parts written in C++, which is used for low-level parsing of the input file and, more importantly, to interface to GECODE [3], a generic CSP solver library, which is used in ASGL as a backend.

The plan of the paper is as follows. Section 2 presents GECODE and ECL [2], the Common Lisp implementation on which ASGL is built. Then, the realization of the computational tasks is exemplified with respect to grounded semantics in Section 3 and preferred semantics in Section 4. Finally, Section 5 details reductions of queries performed by ASGL.

2 System Components

GECODE, the generic constraint development environment, is a toolkit for developing constraint-based systems and applications. It is an efficient, generic CSP solver, featuring among others finite domain integer variables and finite set variables and constraints (see Section 4). Conceived as a C++ library, it can be easily integrated with other systems and it is designed to be open for extension

by user code. For instance, it is possible to program new search engines that can be regarded *en par* to the already built-in ones, such as depth-first search and branch-and-bound search, without requiring the user to dive into low-level code.

The abstraction that new search engines can be programmed with is called computation space [5]. A computation space, which is a first-class citizen, encapsulates a speculative computation involving constraints. New constraints can be posted on a space. Another space operation is to wait for it to become stable, i.e. for propagation to reach a fixpoint (see Section 3).

ECL is an implementation of Common Lisp that features a byte-code compiler, but can also compile to C code, which is then further compiled by the host's native compiler. This allows for easy integration with C or C++ libraries. While bindings to such libraries are usually generated with a tool such as SWIG [1], ECL has a feature that makes this unnecessary. It allows the user by the use of an inline construct *ffi:c-inline* to directly emit fragments of C or C++ code as part of a Lisp function definition. These fragments are placed (almost) verbatim within the generated C code.

As an example, this allows for the definition of a Lisp function *space-status* for the aforementioned operation on a space, which calls the underlying GECODE method *status*.

```
(defun space-status (space)
  (let ((status
         (ffi:c-inline (space) (:pointer-void) :int
                       "{
// wait for space to become stable, then retrieve status
Gecode::SpaceStatus status = (((Gecode::Space*)(#0))->status());

switch (status) {
case Gecode::SS_FAILED: @(return 0) = 1; break;
case Gecode::SS_SOLVED: @(return 0) = 2; break;
case Gecode::SS_BRANCH: @(return 0) = 3; break;
default: @(return 0) = 100; break;
}
}")))
    (ecase status
      (1 :failed)
      (2 :solved)
      (3 :branch))))
```

ASGL makes use of typical Lisp features, such as CLOS, the Common Lisp Object System, macros (see Section 5) and first-class functions to orchestrate on a high level the parsing of the command-line arguments and input file, the creation and appropriately constraining of a computation space, the invocation of a search engine and then, finally, the formatting of the output.

3 Grounded Semantics

Exactly one grounded extension exists. It contains all the arguments which are not defeated, as well as that arguments which are directly or indirectly defended by non-defeated arguments. An algorithm to compute this extension in linear time is given by Caminada [4]; it progresses iteratively until a fixpoint is reached.

In ASGL, no special purpose algorithm for grounded semantics has been implemented. A computation space is created as for the other semantics with an array of $|Ar|$ boolean variables: ASGL uses an extension-based encoding for solutions. Constraints are then posted on the space and the status is queried by *space-status*, which first waits for the space to become stable, i.e. propagation to reach a fixpoint. Subsequently, the grounded extension can be read from the space by including all arguments whose corresponding variables are instantiated to *true*.

4 Preferred Semantics

The preferred extensions are all those complete extensions that are maximal with respect to set inclusion. In general, one or more preferred extensions may exist.

The task of computing some preferred extension is implemented in ASGL like a classical optimization problem with branch-and-bound search. As soon as one solution has been found, all further solutions are constrained to be better than the current solution. If no more solutions can be found, the current solution is maximal.

In the case of preferred extensions, an extension is better than another iff it is a proper superset of the other. In order to allow this kind of constraint to be posted, ASGL makes use of an additional set variable that represents the extension as a set. For consistency, the set variable is connected to the array of boolean variables by a channeling constraint. A branch-and-bound search engine is already part of standard GECODE. This allows for a straightforward implementation of this strategy. The search for a maximal solution is further supported by a value heuristic: When a choice needs to be made, an argument is considered to be included first, before it is considered to be excluded.

No built-in search engine in GECODE can be used to efficiently enumerate all preferred extensions. In the development of ASGL, an attempt has been made to implement a *multi-bab-engine* for this purpose. This engine essentially keeps a master computation space that stays unmodified by individual searches. Only a clone of the master is passed to the built-in branch-and-bound search engine. Whenever this engine finds a preferred extension, the master is constrained not to be a subset of this extension and the process repeats.

Unfortunately, this strategy turned out to be slower than filtering all complete extensions for maximality – at least for small input graphs. In the current version of ASGL, this work has therefore been abandoned in favor of this more simple approach. We expect that repeatedly restarting from the master space incurred too much overhead by repeating work already performed in previous invocations. This effect could possibly be mitigated by utilizing no-good learning or by employing more sophisticated variable ordering heuristics, such as accumulated failure count or activity that build on information gained from previous searches.

5 Reductions

ASGL allows the product of grounded, complete, preferred, and stable semantics and enumeration of some or all extensions, credulous and skeptical inference, in total 16 different problems to be solved. In this problem space numerous reductions are possible. ASGL currently makes use of the following rules, which are – thanks to Lisp macros – written exactly as given here in the source code:

1. `(translate (:se :co) -> (:se :gr))`
2. `(translate (:ds :co) -> (:ds :gr))`
3. `(translate (:dc :pr) -> (:dc :co))`

The first and second rule are quite simple: 1. When asked for some complete extension, one could simply compute the grounded extension. 2. a) If an argument is included in all complete extensions, it is also included in the grounded extension. b) The grounded extension is a subset of all complete extensions, therefore an argument included in the grounded extension is included in all complete extensions.

3. The third rule is more subtle. It states that whether an argument is included in some preferred extension can be reduced to the question of whether the argument is included in some complete extension. This can be shown like this: a) If an argument is included in some preferred extension, it is also included in some complete extension. b) If an argument is included in some complete extension, this extension is either maximal – a preferred extension or a preferred extension must exist that is a superset of the complete extension, hence it includes the argument.

References

1. Beazley, D.M.: Swig: An easy to use tool for integrating scripting languages with c and c++. In: Proceedings of the 4th Conference on USENIX Tcl/Tk Workshop, 1996 - Volume 4. pp. 15–15. TCLTK'96, USENIX Association, Berkeley, CA, USA (1996)
2. Garcia-Ripoll, J.J., Kochmański, D.: Ecl: Embeddable common-lisp (2001), available from `http://ecls.sourceforge.net`
3. Gecode Team: Gecode: Generic constraint development environment (2006), available from `http://www.gecode.org`
4. Modgil, S., Caminada, M.W.: Proof theories and algorithms for abstract argumentation frameworks. In: Rahwan, I., Simari, G. (eds.) Argumentation in Artificial Intelligence, pp. 105–129. Springer Publishing Company, Incorporated (2009)
5. Schulte, C.: Programming constraint inference engines. In: Smolka, G. (ed.) Proceedings of the Third International Conference on Principles and Practice of Constraint Programming. Lecture Notes in Computer Science, vol. 1330, pp. 519–533. Springer-Verlag, Schloß Hagenberg, Austria (Oct 1997)
6. Sprotte, K.: Asgl: Argumentation semantics in gecode and lisp (2015), available from `https://github.com/kisp/asgl`
7. Steele, Jr., G.L.: Common Lisp: the language. Second edn. (1990)