CSE 505: Computing with Logic – Project Report

Eugenia Soroka – 111494044 – ysoroka@cs.stonybrook.edu

December 2018

1 Paper Overiew

1.1 Selected Paper

The paper I selected is **"Specifying and Verbalising Answer Set Programs in Controlled Natural Language" by Rolf Schwitter** (Department of Computing, Macquarie University, Sydney, Australia), published in Volume 18, Special Issue 3-4 (34th International Conference on Logic Programming) in July 2018.
The paper shows how a bi-directional grammar can be used to specify and verbalise answer set programs in controlled natural language (CNL), namely in PENG (**P**rocessable **Eng**lish). The point is to start from a program specification in CNL and translate this specification automatically into an executable answer set program. The resulting answer set program can be modified following certain naming conventions and the revised version of the program can then be verbalised in the same subset of natural language that was used as specification language.

1.2 Motivation

There exist a number of CNLs that have been designed as high-level interface languages to knowledge systems. However, none of the underlying grammars of these controlled natural languages is bi-directional in the sense that a specification can be written in controlled natural language, translated into a formal target representation, and then - after potential modifications of that target representation - back again into the same subset of natural language.

1.3 Bi-directional Grammar

Example (slightly changed version of the example given in the paper):

Program specification in CNL:

1. Every student who works is successful.
2. Every student who studies at SBU works or parties.
3. It is not the case that a student who is enrolled in CS parties.
4. Bob is a student.
5. Bob studies at SBU and is enrolled in CS.
6. Who works?

Program converted to ASP:

1. `successful(A) :- student(A), work(A).`
2. `work(B); party(B) :- student(B), study_at(B, sbu).`

3. `:- student(C), enrolled_in(C, cs), party(C).`
4. `student(bob).`
5. `study_at(bob, sbu). enrolled_in(bob, cs).`
6. `answer(D) :- work(D).`

To implement the bi-directional grammar, the definite clause grammar (DCG) formalism is used. The implementation of the grammar relies on four components that are necessary to build an operational system:

1. A **tokeniser** that splits a specification written in controlled natural language into a list of tokens for each sentence.

2. A **writer** that translates the internal format constructed during the parsing process into the final answer set program.

3. A **reader** that reads a (potentially modified) answer set program and converts this program into the internal format.

4. A **sentence planner** that tries to apply a number of aggregation strategies, reorganizes the internal format for this purpose, before it is used by the grammar for the generation process.

The bi-directional DCG uses an internal format for answer set programs and a special feature structure that distinguishes between **a processing and a generation mode in the grammar rules**. Depending on this mode, the same difference list is used as a data structure to either construct the internal format for an input text or to deconstruct the internal format to generate a verbalization.

2 My Project

No ready-to-use code or implementation of this system is available, and even control natural language used (PENG) is not publicly available, which means I had to specify and implement this system from scratch.
For this project, I decided to recreate what was done in the paper but on a smaller scale. Since the main motivation of the paper was to create a **bi-directional** grammar suitable for both processing and generation of program specifications in CNL, I focused on the bi-directionality aspect, as opposed to defining a huge and diverse vocabulary to work with. Even though this means that the functionality is somewhat constrained, I am positive that the main point of the project was achieved.

2.1 System Input and Output

1. **Input sentences:** sentences written in controlled natural language by a user.
   Input sentences must follow the format of the example sentence `['every',kid,dreams,'.']`, i.e. all words should be separated by a comma, sentence must end with the special symbol `'.'`, and a sentence must start with the system word `'every'` or have an agent in it, e.g. like in `[bob,dreams,'.']`, where `bob` is an agent. Some words are also system words, e.g. `'does not'`. For the the full list of reserved (system) words, see the program.

2. **Internal representation:** special internal format to represent input sentences as logic rules or clauses.

The internal format is basically the input sentences converted to First-Order Logic. This internal format already contains terms with arguments, as extracted from the input sentences, and it then used to create an ASP program from it.

3. **ASP representation:** internal representation transformed into ASP format (e.g. suitable for *clingo*).
   This ASP representation outputs rules and clauses of the resulting ASP. Since ASPs can also include `not` clauses and rules can be constraints, I had to account for that. I have also included strict negations, i.e. `-term(Argument)`, which *clingo* works with.

4. **Generated (back) sentences:** sentences reconstructed from internal representation, in ideal case – the same as input sentences.
   The output sentences are generated from the internal format using the same grammar but in the generating mode, `gen`, (as opposed to the processing mode, `proc`, used when converting input sentences to internal format).

2.1.1 Example of the output of the program:

```
1. Input sentences:
[alice,is,girl,.]
[every,girl,is,happy,.]
[no,student,is,happy,.]
[bob,is,student,.]

2. Internal representation:
girl(alice)
forall(A,[girl(A)]==>[happy(A)])
none(A,[student(A)]==>[happy(A)])
student(bob)

3. ASP representation:
girl(alice).
happy(A) :- girl(A).
-happy(A) :- student(A).
student(bob).

4. Generated (back) sentences:
[alice,is,girl,.]
[every,girl,is,happy,.]
[no,student,is,happy,.]
[bob,is,student,.]
```

Using this ASP representation, we can use *clingo* to solve it, and the answer we get is:
`girl(alice) happy(alice) -happy(bob) student(bob)`

If we want to, say, query whether `alice` is a student, we may manually add a rule `yes :- student(alice)`, and `yes` will be in the list of answers, if `alice` is a student, and not, otherwise. Running this (modified) program gives the same list of answers, from which we see that `alice` is NOT a student.

2.2 Components of the program

1. **Lexicon** – vocabulary of allowed words.
   Lexicon is specified by two types of predicates: `lexicon` and `agent`. `lexicon` has the format as follows: `lexicon(cat:noun, wform:[student], arg:X, term:student(X))`, where the `cat` argument is the category of the word (noun, verb, etc.); `wform` stands for word-form, i.e. the exact word that appears in the sentence; `arg` specifies argument of the term that the word refers to; `term` denotes the actual term that will be used in the internal format and in ASP.
   `agent` has the form `agent(bob)`, and simply contains a single agent as its argument. So before a user can run the program, they should specify all the agents they are going to use.

2. **Bi-directional Definite Clause Grammar (DCG)** - DCG created using Prolog tools for grammar specification, using difference lists to find matching words in the sentence or matching structures in the internal representation.
   Basically, the grammar specifies rules of the form:
   ```
   s --> np, vp, ['.'].
   np --> det, noun.
   np --> noun.
   vp --> iv.
   ...,
   ```
   but in a more complex fashion, since I have to account for the mode, deal with terms and arguments. As a bi-directional grammar, this DCG has two modes: `proc` and `gen`, which specify different rules for the processing and generating mode, respectively.

3. **Rules for processing** – rules used to convert sentences to internal representation.
   The rules look like this:
   ```
   s(mode:proc, sem:M1-M2, st:St) -->
   np(mode:proc, arg:X, sem:M1-M0, st:pos),
   vp(mode:proc, arg:X, sem:M0-M2, st:St),
   ['.'].
   ...
   det(mode:proc, arg:X, res:[[]|M1]-M2, sco:[[]|M2]-[Sco, Res, M3|M4], sem:M1-[[
   forall(X, Res ==> Sco)|M3]|M4]) --> ['every'].
   ...
   noun(mode:proc, arg:X, sem:[M1|M2]-[[T|M1]|M2], st:pos) -->
   lexicon(cat:noun, wform:List, arg:X, term:T);
   ( agent(X), List = [X] ),
   List.
   ```

4. **Rules for converting to ASP** – rules used to convert internal representation to ASP.
   This part is relatiely simple (compared to grammar specification). Here I take the internal format and transform it to the ASP, using rules like these:
   ```
   make_ASP([forall(_,[T1]==>[T2])|Clauses]) :-
   portray_clause(T2 :- T1),
   make_ASP(Clauses).
   ...
   make_ASP([[T1|[Ts]]|Clauses]) :-
   write(T1), write(" , "),
   make_ASP([Ts|Clauses]).
   ```

5. **Rules for generation** – rules used to convert internal representation back to sentences. Just like rules for processing, these are more complicated than `make_ASP()` and took quite some time to think of and to write. The examples are:

```
s(mode:gen, sem:S, st:St) -->
np(mode:gen, arg:X, sem:S, st:pos),
vp(mode:gen, arg:X, sem:S, st:St),
['.'].
...
vp(mode:gen, arg:X, sem:S, st:pos) -->
['is'],
jj(mode:gen, arg:X, sem:S, st:pos).
vp(mode:gen, arg:X, sem:S, st:neg) -->
['is not'],
jj(mode:gen, arg:X, sem:S, st:neg).
...
```

2.3 Conclusions

While working on this project, I:

1. Learned how definite clause grammars work, how to write one in Prolog
2. Learned how to utilize natural language processing in Prolog to specify and verbalize answer set programs
3. Came up with specific format and rules for defining different parts of speech
4. Based on the paper, wrote a bi-directional grammar, which can be used for converting CNL sentences to ASP and back

2.4 Resources

Project materials: source code, presentation (Phase 3), report, usage instructions – are all stored in my GitHub repository.

References

SCHWITTER, R. (2018). Specifying and Verbalising Answer Set Programs in Controlled Natural Language. Theory and Practice of Logic Programming, 18(3-4), 691-705. doi:10.1017/S1471068418000327