Specifying Optimisation Problems for Declarative Programs in Precise Natural Language

Rolf Schwitter
School of Computing
Macquarie University
Sydney NSW 2109
Rolf.Schwitter@mq.edu.au

Abstract

We argue that declarative programs can be written in precise natural language and back this claim by using a complex optimisation problem. The problem specification is expressed in natural language and automatically translated into an executable answer set program, and an answer set solver is then used to find (optimal) solutions for natural language questions. Our approach enables subject matter experts to express their knowledge in a natural and truly declarative notation without the need to encode this knowledge in a formal way.

1 Introduction

Declarative programming involves stating *what* is to be computed, but not *how* it is to be computed; it is a programming paradigm that expresses the logic of computation without describing its control flow (Lloyd, 1994). Logic programming languages (Körner et al., 2022) and functional programming languages (Hu et al., 2015) belong to the declarative programming paradigm and result in code that is characterised by a high level of abstraction. One of the main benefits of declarative programming languages is their ability to describe problems with less code than imperative programming languages. Furthermore, declarative languages are known to be elaboration-tolerant, precise, and easier to optimise than imperative languages. In the context of logic programming, the development of the stable model semantics for logic programs (Gelfond and Lifschitz, 1988) has led to answer set programming (Janhunen and Nimelä, 2016), a powerful model-based language for knowledge representation and non-monotonic reasoning with industrial applications (Falkner et al., 2018). Writing an answer set program involves identifying objects and the relations between them and formally encoding this information as facts, rules and constraints. However, the resulting formal notation may be difficult to understand by subject matter experts who have detailed knowledge about the application domain but do not have a background in formal logic. Instead of encoding a problem specification in answer set programming notation, we suggested in previous work to express such a specification in a precise subset of natural language (Schwitter, 2018). Such a precise subset of natural language is also known as a controlled natural language (Kuhn, 2014). It has been shown that the writing of a specification in controlled language can be supported by a predictive authoring tool like the writing of code with the help of a code editor (Schwitter, 2020; Schwitter et al., 2003). That means the authoring tool instructs the user about the language constructs that can be used to construct a textual specification. As in the case of writing code, writing a specification in controlled language is a process that requires careful planning and an understanding of the application domain. In this paper, we focus on an extension of our controlled language PENG^{ASP} and illustrate how this controlled language can be used to specify a complex optimisation problem that goes beyond our previous work but uses similar unification-based techniques for the translation into an answer set program (Guy and Schwitter, 2017). The novelty is the use of choice rules, aggregates, and optimisation statements that can be expressed directly on the level of the controlled language. A state-of-the-art answer set solver like *clingo*[1] can then be used to find the (optimal) solutions to the problem.

2 Answer Set Programming

Answer set programming (ASP) has its roots in the fields of logic programming and non-monotonic reasoning (Lifschitz, 2019; Gelfond and Kahl,

[1] https://potassco.org/clingo/

2014) and has been applied to a wide range of areas in artificial intelligence, among them also to natural language processing tasks (Mitra et al., 2019; Sharma, 2019; Schüller, 2018; Dzifcak et al., 2009). ASP is supported by powerful reasoning tools and offers a rich representation language that allows for recursive definitions, (strong and weak) negation, (strong and weak) constraints, aggregates, optimisation statements, and external functions (Gebser et al., 2019). An ASP program consists of a set of rules of the following form:

```
l1 ;...; lm :-
  lm+1,..., ln, not ln+1,..., not lo.
```

Here each l_i is a literal. A literal is either a positive atom of the form $p(t_1,\ldots,t_k)$ or its strong negation $-p(t_1,\ldots,t_k)$, where p is a predicate name and all t_i are terms that are composed of function symbols and variables. The symbol `:-` ("if") separates the head of a rule from its body. The symbol `;` in the head of a rule stands for an (epistemic) disjunction, and the symbol `not` in the body for default negation (aka negation as failure). A rule is called a fact if $m = o = 1$, normal if $m = 1$, and an integrity constraint if $m = 0$. Semantically, the rule above states that if $l_{m+1},\ldots,l_n$ are true and there is no reason to believe that $l_{n+1},\ldots,l_o$ are true, then at least one of $l_1,\ldots,l_m$ is believed to be true.

To ease the use of ASP for practical applications, several simplifying notations and extensions have been developed (see (Gebser et al., 2019) for details). The most notable ones in our context are: choice rules, aggregates, and objective functions.

A choice rule has the following form:

```
s { e1 ;...; em } t :- body.
```

Here e_i is a choice element of the form $a:L_1,\ldots,L_k$, where a is an atom; L_i are possible default-negated literals; and s and t are integers which express lower and upper bounds on the cardinality of elements. Intuitively, a choice rule means that if the body of the rule is true, then an arbitrary number of elements can be chosen as true as long as this number complies with the lower and upper bounds. Note that if $e_i \geq 2$ and $s = t = 1$, then a choice rule implements an exclusive disjunction; similarly, if $s = 1$, $t = nil$, then it implements an inclusive disjunction. Aggregates are functions that apply to sets and can be used to calculate for example the number of elements of a set. For instance, the expression:

```
#count { X, Y : p(X, Y) }
```

represents the number of elements of the set `p/2`. Expressions like this can be used in the body of an ASP rule as one side of a comparison, with a variable on the other side, for example:

```
number_of_elements(N) :-
  N = #count { X, Y : p(X, Y) }.
```

When an ASP program has several answer sets, we may be interested in finding the best possible one, according to some measure of quality. Objective functions can be used in this case to minimise or maximise the sum of a set of weighted tuples (w_i, t_i) that are subject to some conditions c_i. These objective functions are expressed in ASP as directives of the following form:

```
#minimize { w1@l1, t1 :  c1 ;...;
            wn@ln, tn :  cn }.
```

Note that w_i is a numerical constant, l_i is an optional (lexicographically ordered) priority level, t_i is a sequence of terms, and c_i is a sequence of possibly default-negated literals. Alternatively, optimisation statements can be implemented as weak constraints. In contrast to integrity constraints that weed out answer sets as solutions, weak constraints rank solutions.

3 Finding an Optimal Accommodation

Let us assume a traveller wants to choose the best one among three different accommodations (Aloe, Metro, Oase); each of them comes with a star rating and a weekly room rent. Furthermore, one of the accommodations is located on the main street and known to be noisy. Considering the available options, the traveller faces the following optimisation problem: (a) minimising noise has the highest priority; (b) minimising the cost per star has the second highest priority; and (c) maximising the number of stars of an accommodation that is otherwise not distinguishable has the lowest priority.[2]

We can start expressing the factual information about these three different accommodations in a precise way in controlled language:

1. *The bedroom apartment Oase is rated three stars and costs 240 dollars.*
2. *The bedroom apartment Aloe is rated two stars and costs 160 dollars.*

[2]This example is inspired by (Gebser et al., 2019).

3. *The studio apartment Metro that is located on the main street is rated three stars and costs 200 dollars.*

Furthermore, we specify ontological statements that are necessary to solve the problem directly in controlled language and describe what counts as an accommodation (4 and 5); as a noisy accommodation (6); as the cost per star of an accommodation (7); and as the star rating of an accommodation (8):

4. *Every studio apartment is an accommodation.*
5. *Every bedroom apartment is an accommodation.*
6. *If an accommodation is located on a main street then the accommodation is noisy.*
7. *If an accommodation costs N dollars and is rated M stars then N / M is the cost per star of the accommodation.*
8. *If an accommodation is rated N stars then N is the start rating of the accommodation.*

Next, we specify that one of the accommodations is the optimal one, using an exclusive disjunction in controlled language:

9. *Either one of Aloe or Metro or Oase is optimal.*

The relevant optimisation statements are expressed with the help of predefined key phrases *Minimise/Maximise with a priority of I* that include a priority level, where a higher integer (*I*) indicates a higher priority:

10. *Minimise with a priority of 3 that an optimal accommodation is noisy.*
11. *Minimise with a priority of 2 that C is the cost per star of an optimal accommodation.*
12. *Maximise with a priority of 1 that S is the star rating of an optimal accommodation.*

Finally, the questions to be answered can be expressed as follows:

13. *How many accommodations are there?*
14. *Which accommodation is optimal?*

This entire specification can be translated automatically into an executable ASP program. The three factual statements (1-3) result in a number of ASP facts. In our case, these facts are based on a reified notation that relies on a small number of predefined predicates. Constants that start with `c` followed by a positive integer `I` are Skolem constants and replace existentially quantified variables.

```
class(c1, bedroom_apartment).      % 1
named(c1, oase).
prop(c1, c2, rated).
data_prop(c2, 3, cardinal).
class(c2, star).
pred(c1, c3, cost).
data_prop(c3, 240, cardinal).
class(c3, dollar).

class(c4, bedroom_apartment).      % 2
named(c4, aloe).
prop(c4, c5, rated).
data_prop(c5, 2, cardinal).
class(c5, star).
pred(c4, c6, cost).
data_prop(c6, 160, cardinal).
class(c6, dollar).

class(c7, studio_apartment).       % 3
named(c7, metro).
prop(c7, c8, located_on).
class(c8, main_street).
prop(c7, c9, rated).
data_prop(c9, 3, cardinal).
class(c9, star).
pred(c7, c10, cost).
data_prop(c10, 200, cardinal).
class(c10, dollar).
```

The ontological statements (4-8) result in five ASP rules that define classes and properties:

```
class(A, accommodation) :-         % 4
  class(A, studio_apartment).

class(B, accommodation) :-         % 5
  class(B, bedroom_apartment).

prop(C, noisy) :-                  % 6
  class(C, accommodation),
  prop(C, D, located_on),
  class(D, main_street).

prop(E/F, G, cost_per_star) :-     % 7
  class(G, accommodation),
  pred(G, H, cost),
  data_prop(H, E, cardinal),
  class(H, dollar),
  prop(G, I, rated),
  data_prop(I, F, cardinal),
  class(I, star).

prop(J, K, star_rating) :-         % 8
  class(K, accommodation),
  prop(K, L, rated),
  data_prop(L, J, cardinal),
  class(L, star).
```

The two rules for the ontological statements 7 and 8 are interesting since both of them contain an atom as rule head that has been derived from a relational noun (*cost per star* and *star rating*) and introduce properties. The property that represents *cost per star* contains an arithmetic function (`E/F`) with two variables as its first argument. This arithmetic function picks up two cardinal numbers and evaluates their ratio during grounding. The prop-

erty that represents *star rating* picks up a cardinal number (J) as its first argument.

The statement 9 is translated into a choice rule. The integers before and after the expression in the braces express lower and upper bounds on the cardinality. In our case, the lower bound and upper bound is 1, meaning that exactly one accommodation is optimal:

```
1 { prop(M, optimal) :              % 9
    named(M, (aloe ;
              metro ;
              oase)) } 1.
```

The optimisation statements (10-12) are translated into weak constraints with the help of `#minimize` and `#maximize` directives. The first argument `w@l` of these directives consists of a weight (w) and priority level (l), greater levels being more significant than smaller ones. These directives instruct *clingo* to look for the best stable model of the given ASP program.

```
#minimize { 1@3,                    % 10
  N : prop(N, optimal),
      class(N, accommodation),
      prop(N, noisy) }.

#minimize { O@2,                    % 11
  P : prop(O, P, cost_per_star),
      prop(P, optimal),
      class(P, accommodation) }.

#maximize { Q@1,                    % 12
  R : prop(Q, R, star_rating),
      prop(R, optimal),
      class(R, accommodation) }.
```

Questions such as (13 and 14) are translated into an ASP rule with a specific answer literal (`answer/1`) as head. These literals will contain the answer to the question after grounding and will be displayed using the `#show` directive (15):

```
answer(T) :- T = #count {           % 13
  S : class(S, accommodation) }.

answer(V) :-                        % 14
  named(U, V),
  class(U, accommodation),
  prop(U, optimal).

#show answer/1.                     % 15
```

Note that question (13) is not necessary to find the optimal solution but illustrates the use of an aggregate construct.

4 Evaluation

If we submit our ASP program to the answer set solver *clingo*, then the solver will generate and display three answer sets (models), one for each accommodation together with the weights used for finding the optimal solution. These answer sets also contain the answers to the questions (13 and 14) that are displayed with the help of the `#show` directive (15):

```
clingo version 5.6.1
Reading from asp.lp
Solving...
Answer: 1
answer(3) answer(metro)
Optimization: 1 66 -3
Answer: 2
answer(3) answer(aloe)
Optimization: 0 80 -2
Answer: 3
answer(3) answer(oase)
Optimization: 0 80 -3
OPTIMUM FOUND

Models       : 3
  Optimum    : yes
Optimization : 0 80 -3
Calls        : 1
Time         : 0.037s (...)
CPU Time     : 0.000s
```

We can see in the output that the accommodation Metro is noisy (1) and therefore not optimal with respect to the most important optimisation statement. The accommodations Aloe and Oase are not noisy (0) and are both optimal with respect to the cost per star ratio (80); the second most important optimisation statement. This tie is broken by the least important optimisation statement that looks at the number of stars. Note that since we maximise the number of stars, the values are displayed as negative integers (-2 and -3). In summary, the optimal accommodation is Oase, since it is not noisy, has a cost per star ratio of 80 and is rated three stars.

5 Conclusion

We showed in this paper that complex optimisation statements can be expressed directly in precise natural language. The resulting specification can then be automatically translated into an executable ASP program. The writing of such a specification in controlled language is usually supported by a predictive authoring tool that has similar features as a code editor for a programming language. As in the case of writing declarative code, writing a textual specification in controlled language needs to be carefully planned and will never be a fast process, but our approach has the potential to close the gap between a (seemingly) informal textual specification and a declarative program. In this sense, programming in controlled language is the most extreme form of declarative programming.

References

Juraj Dzifcak, Matthias Scheutz, Chitta Baral, and Paul W. Schermerhorn. 2009. What to do and how to do it: Translating natural language directives into temporal and dynamic logic representation for goal management and action execution. *2009 IEEE International Conference on Robotics and Automation*, pages 4163–4168.

Andreas Falkner, Gerhard Friedrich, Konstantin Schekotihin, Richard Taupe, and Erich C. Teppan. 2018. Industrial applications of answer set programming. *KI - Künstliche Intelligenz*, 32(2):165–176.

Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Marius Lindauer, Max Ostrowski, Javier Romero, Torsten Schaub, Sven Thiele, and Philipp Wanko. 2019. *Potassco User Guide, Version 2.2.0*.

Michael Gelfond and Yulia Kahl. 2014. *Knowledge Representation, Reasoning, and the Design of Intelligent Agents, The Answer-Set Programming Approach*. Cambridge University Press.

Michael Gelfond and Vladimir Lifschitz. 1988. The stable model semantics for logic programming. In *Proceedings of International Logic Programming Conference and Symposium*, pages 1070–1080. MIT Press.

Stephen Guy and Rolf Schwitter. 2017. The PENG^{ASP} system: Architecture, language and authoring tool. *Journal of Language Resources and Evaluation, Controlled Natural Language*, 51:67–92.

Zhenjiang Hu, John Hughes, and Meng Wang. 2015. How functional programming mattered. *National Science Review*, 2(3):349–370.

Tomi Janhunen and Ilkka Nimelä. 2016. The answer set programming paradigm. *AI Magazine*, 37(3):13–24.

Tobias Kuhn. 2014. A survey and classification of controlled natural languages. *Computational Linguistics*, 40(1):121–170.

Philipp Körner, Michael Leuschel, João Barbosa, Vítor Santos Costa, Verónica Dahl, Manuel V. Hermenegildo, Jose F. Morales, Jan Wielemaker, Daniel Diaz, Salvador Abreu, and Giovanni Ciatto. 2022. Fifty years of prolog and beyond. *Theory and Practice of Logic Programming*, page 1–83.

Vladimir Lifschitz. 2019. *Answer Set Programming*. Springer, Cham.

John W. Lloyd. 1994. Practical advantages of declarative programming. In *Proceedings of GULP-PRODE'94*, volume I, pages 3–17, Peñíscola, Spain.

Arindam Mitra, Peter Clark, Oyvind Tafjord, and Chitta Baral. 2019. Declarative question answering over knowledge bases containing natural language text with answer set programming. In *Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence*, pages 3003–3010, Honolulu, Hawaii, USA. AAAI Press.

Rolf Schwitter. 2018. Specifying and verbalising answer set programs in controlled natural language. *Journal of Theory and Practice of Logic Programming*, 18:691–705.

Rolf Schwitter. 2020. Lossless semantic round-tripping in PENG^{ASP}. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI-20*, pages 5291–5293.

Rolf Schwitter, Anna Ljungberg, and David Hood. 2003. Ecole: A look-ahead editor for a controlled language. In *Controlled Language Translation*, pages 141–150. Dublin University.

Peter Schüller. 2018. Answer set programming in linguistics. *KI - Künstliche Intelligenz*, 32(2):151–155.

Arpit Sharma. 2019. Using answer set programming for commonsense reasoning in the winograd schema challenge. *Theory and Practice of Logic Programming*, 19(5-6):1021–1037.