An Experiment with Anthem: Semantic Equivalence of Tiling Programs

Vladimir Lifschitz

University of Texas at Austin

Abstract. ANTHEM is a proof assistant designed for verifying several conditions that play an important role in answer set programming. In this note we show that ANTHEM can help us verify equivalence of logic programming solutions to the same problem that have been independently developed by different programmers.

1 Introduction

ANTHEM (`https://potassco.org/anthem/`) is a proof assistant designed for verifying a number of conditions that play an important role in answer set programming (ASP). One of these conditions is *external equivalence* of ASP programs. External equivalence means, informally speaking, that the programs exhibit the same external behavior for all permissible inputs. The word "external" refers to the idea that two programs with the same output predicates may be equivalent even if their auxiliary ("private") predicates are different. This can be made precise by defining equivalence with respect to a "user guide," which is a formal expression specifying the set of permissible inputs and the set of output predicates [2].

The article mentioned above motivates studying the external equivalence relation by the fact that improving a correct but inefficient ASP encoding amounts to replacing a program by another program that is equivalent to it with respect to a user guide. ANTHEM does not tell us how to improve a program, but it can help us verify that two versions of a program have the same functionality as far as their output predicates are concerned.

Michael Gelfond has observed (personal communication) that verifying external equivalence can be used also for another purpose: for investigating the relationship between independently developed ASP encodings of the same domain. As an experiment, we investigate here the relationship between two encodings written by students for a class taught at the University of Texas in 2005. The assignment was to solve a tiling puzzle: covering the 8×8 square by twenty–one 3×1 tiles and a single 1×1 tile [1]. The puzzle has 1424 solutions, as can be demonstrated by running the answer set solver CLINGO (`https://potassco.org/clingo/`) on either encoding.

To make the question more interesting, we generalize the problem as follows:

> *Given a positive integer n, find all covers of the $n \times n$ square by 3×1 tiles and a single 1×1 tile.*

Such a cover exists whenever n is different from 2 and is not a multiple of 3 (Krzysztof Apt, personal communication).

The programs written by students have been generalized accordingly. (That was straightforward: it was enough to replace `8` in the programs by `n`, and `6` by `n-2`.) The programs have been also edited to make them more concise, and adapted to the syntactic restrictions required by ANTHEM.

Listing 1.1. Program A

```
1  % T = 1: a 1x1 tile.
2  % T = 2: a horizontal 3x1 tile with the
3  % leftmost square at (X,Y).
4  % T = 3: a vertical 3x1 tile with the
5  % topmost square at (X,Y).
6
7  {place(X,Y,T)} :- X = 1..n, Y = 1..n, T = 1..3.
8
9  :- place(X,Y,T1), place(X,Y,T2), T1 != T2.
10 :- place(X,Y,1), place(X1,Y1,1), X != X1.
11 :- place(X,Y,1), place(X1,Y1,1), Y != Y1.
12
13 % filled(X,Y) means that (X,Y) is covered by
14 % one of the tiles.
15 filled(X,Y) :- place(X,Y,1).
16 filled(X+I,Y) :- place(X,Y,2), I = 0..2.
17 filled(X,Y+I) :- place(X,Y,3), I = 0..2.
18
19 :- not filled(X,Y), X = 1..n, Y = 1..n.
20 :- place(X,Y,2), X > n-2.
21 :- place(X,Y,3), Y > n-2.
22 :- place(X,Y,2), place(X+I,Y,T), I = 1..2.
23 :- place(X,Y,3), place(X,Y+I,T), I = 1..2.
24 :- place(X,Y,2), place(X+I,Y-J,3),
25     I = 1..2, J = 1..2.
26 :- place(X,Y,3), place(X-I,Y+J,2),
27     I = 1..2, J = 1..2.
```

2 Two encodings

The programs, shown in Listings 1.1 and 1.2, are designed in accordance with the same general principles, common in applications of ASP to search problems. A solution is represented by a set of ground atoms that are formed using "output predicates" of the program. Both programs include choice rules that describe "potential solutions." They include also constraints, which weed out potential

solutions that are not fully satisfactory, and define auxiliary predicates, which are used to express the constraints.[1]

Listing 1.2. Program B

```
1  % h(R,C) means that there is a tile
2  % at (R,C), (R,C+1), (R,C+2).
3  % v(R,C) means that there is a tile
4  % at (R,C), (R+1,C), (R+2,C).
5
6  {h(1..n,1..n-2)}.
7  {v(1..n-2,1..n)}.
8
9  square(1..n,1..n).
10
11 % covered(R,C) means that (R,C) is covered
12 % by a 3x1 tile.
13 covered(R,C+I) :- h(R,C), I = 0..2.
14 covered(R+I,C) :- v(R,C), I = 0..2.
15
16 :- square(R1,C1), square(R2,C2),
17    not covered(R1,C1), not covered(R2,C2),
18    R1 != R2.
19
20 :- square(R1,C1), square(R2,C2),
21    not covered(R1,C1), not covered(R2,C2),
22    C1 != C2.
23
24 :- h(R,C), h(R,C+(1..2)).
25 :- v(R,C), v(R+(1..2),C).
26 :- h(R,C), v(R-(0..2),C+(0..2)).
```

But the two programs have different output predicates.

In Program A, the output predicate is `place/3`. Its last argument shows whether the tile is 1×1 or 3×1, and, in the latter case, whether it is placed horizontally or vertically. The other two arguments are the coordinates of the "head" of the tile. The choice rule in Line 7 allows us to choose the number of tiles and their positions arbitrarily, as long as the head of every tile is within the $n \times n$ square. The constraint in Line 19, expressing that the entire square is covered, uses the auxiliary predicate `filled/2`, which is defined in Lines 15–17.

In Program B, on the other hand, two output predicates describe the positions of 3×1 tiles placed horizontally (`h/2`) and vertically (`v/2`), and there is no symbol for the position of the 1×1 tile. Instead of the constraint expressing that

[1] This perspective on the structure of search programs is useful for writing them, but the search algorithms implemented in answer set solvers do not involve generating all potential solutions—their number can be astronomical. These algorithms are similar to those used in the design of satisfiability solvers [3].

the entire square is covered, in Listing 1.2 we see a pair of constraints requiring that 3×1 tiles miss at most one position in the $n\times n$ square (Lines 16–22). [2] The auxiliary predicate `covered/2`, defined in Lines 13 and 14, is slightly different from `filled/2` from Program A: `filled(R,C)` means that the position in row `R` and column `C` is covered by one of the 3×1 tiles.

The programs differ also by the way they order the arguments denoting the coordinates of a position: the row number `R` in Program B corresponds to the `Y`-coordinate in Program A, and the column number `C` corresponds to the `X`-coordinate. Thus the atom `h(5,6)` in the output of Program B corresponds to `place(6,5,2)` in the output of Program A.

3 Combining output predicates

Because of the difference between their output predicates, Programs A and B are not externally equivalent in the sense of the theory behind ANTHEM. They are only equivalent in the weaker sense discussed by Pearce and Valverde [5], who observe that two knowledge descriptions can be semantically equivalent even if they are expressed in different languages or vocabularies.

But we can make these programs externally equivalent if we extend Program A by definitions of `h/2` and `v/2` (Listing 1.3), and extend Program B by a definition of `place/3` (Listing 1.4). Adding these rules makes the programs equivalent with respect to the user guide that classifies `n` as a placeholder for an integer, and the symbols `h/2`, `v/2`, `place/3` as output predicates (Listing 1.5).

This claim can be verified by ANTHEM as described in the next section.

Listing 1.3. Rules to be added to Program A

```
1  h(R,C) :- place(C,R,2).
2  v(R,C) :- place(C,R,3).
```

Listing 1.4. Rules to be added to Program B

```
1  place(X,Y,1) :- square(Y,X), not covered(Y,X).
2  place(X,Y,2) :- h(Y,X).
3  place(X,Y,3) :- v(Y,X).
```

Listing 1.5. User guide for the extended programs

```
1  input: n -> integer.
2  output: h/2.
3  output: v/2.
4  output: place/3.
```

[2] The original program employed the `#count` aggregate to express this condition more concisely. The versions of ANTHEM available at the time of this writing are not applicable to aggregate expressions.

4 Operation of anthem

ANTHEM has been designed and implemented by researchers at the University of Potsdam, the University of Nebraska Omaha, and the University of Texas at Austin. It verifies a claim about external equivalence of programs by reducing it to proving certain formulas in a first-order theory and invoking the theorem prover VAMPIRE [4] to find a proof. Formulas are written in a language with terms of two sorts, *general* and its subsort *integer*. General variables are similar to variables in ASP programs; their domain includes both integers and symbolic constants. Integer variables are distinguished from general variables by the symbol $ as their last character. The syntax of the language does not allow general variables in the scope of an arithmetic operation.

The user can help VAMPIRE organize search by providing a *proof outline*—a list of first-order sentences ("lemmas") to be proved consecutively before attempting to prove the main goal. Some lemmas are used in only one half of the proof of equivalence, "forward" (left-to-right) or "backward" (right-to-left).

The output of ANTHEM describes each reasoning task given to VAMPIRE by listing the axioms and the conjecture that VAMPIRE is instructed to derive from them. The user of ANTHEM usually approaches a verification task by checking first whether VAMPIRE can succeed within reasonable time—say, 5 minutes—without help. For nontrivial tasks, this first attempt usually fails. The next step is to find a lemma that can be derived by VAMPIRE from the axioms without help and that is likely to facilitate achieving the goal when added to the list of axioms. Several steps of this kind may be required. Some lemmas can be stated more concisely using explicitly defined predicates, and such definitions can be included in the proof outlines along with the statements of lemmas.

The proof outline used in the verification of the equivalence claim from Section 3 is shown in Listing 1.6. Given this proof outline, the ANTHEM-VAMPIRE team, invoked with 6 cores on a machine running Ubuntu 20.04.6, 8 Intel(R) Xeon(R) E3-1271 CPUs, 16 GB RAM, terminated within 326 seconds.

Listing 1.6. Proof outline

```
1  definition: forall I$ J$ (filled2(I$,J$) <->
2    place(I$,J$,2) or place(I$-1,J$,2)
3    or place(I$-2,J$,2)).
4
5  definition: forall I$ J$ (filled3(I$,J$) <->
6    place(I$,J$,3) or place(I$,J$-1,3)
7    or place(I$,J$-2,3)).
8
9  lemma(forward):
10 filled2(I$,J$) ->
11   h(J$,I$) or h(J$,I$-1) or h(J$,I$-2).
12
13 lemma(forward): filled2(I$,J$) -> covered(J$,I$).
14
```

```
15  lemma(forward):
16  filled3(I$,J$) ->
17    v(J$,I$) or v(J$-1,I$) or v(J$-2,I$).
18
19  lemma(forward): filled3(I$,J$) -> covered(J$,I$).
20
21  lemma(forward):
22  filled(I$,J$) ->
23    place(I$,J$,1)
24    or filled2(I$,J$) or filled3(I$,J$).
25
26  lemma(forward):
27  square(I$,J$) ->
28    place(I$,J$,1)
29    or filled2(I$,J$) or filled3(I$,J$).
30
31  lemma(forward):
32  square(I$,J$) -> place(I$,J$,1) or covered(J$,I$).
33
34  lemma(backward): not(h(R,C) and v(R,C)).
35
36  lemma(backward):
37  not(h(R$,C$) and h(R$,C$+I$) and 1 <= I$ <= 2).
38
39  lemma(backward):
40  not(h(R$,C$) and v(R$,C$+I$) and 0 <= I$ <= 2).
41
42  lemma(backward):
43  not(v(R$,C$) and h(R$+I$,C$) and 0 <= I$ <= 2).
44
45  lemma(backward):
46  not(v(R$,C$) and h(R$+I$,C$-J$)
47    and 0 <= J$ <= 2 and 1 <= I$ <= 2).
48
49  lemma(backward):
50  square(I$,J$) ->
51    place(I$,J$,1)
52    or filled2(I$,J$) or filled3(I$,J$).
53
54  lemma(backward): filled2(I$,J$) -> filled(I$,J$).
55
56  lemma(backward): filled3(I$,J$) -> filled(I$,J$).
```

5 Conclusion

Programs A and B describe the same domain in the same dialect of answer set programming, and they represent the input in the same way. But the output is represented in them by different predicates. After extending each encoding by rules defining the output predicates of the other we obtained a pair of programs that are equivalent with respect to the user guide in which the output predicates of A and B are combined.

The proof outline required for completing verification in the example above includes a large number of lemmas. Inventing these lemmas in the process of interaction with ANTHEM involved a long series of experiments; it was challenging and time consuming. This example can serve as a benchmark for evaluating future versions of ANTHEM and VAMPIRE. Making the number of lemmas smaller without increase in runtime will be a sign of progress.

Acknowledgements

Many thanks to Michael Gelfond, who suggested to me the possibility of using ANTHEM for comparing alternative encodings of the same domain, to Zachary Hansen for sharing with me the best available version of ANTHEM, to Pedro Cabalar, Jorge Fandinno, Martin Gebser, Roland Kaminski and anonymous referees for comments on drafts of this note, and to Krzysztof Apt and Tobias Stolzmann for useful discussions related to its topic.

References

1. Dijkstra, E.W.: Seemingly on a problem transmitted by Bengt Jonsson (1989), https://www.cs.utexas.edu/~EWD/ewd10xx/EWD1039.PDF
2. Fandinno, J., Hansen, Z., Lierler, Y., Lifschitz, V., Temple, N.: External behavior of a logic program and verification of refactoring. Theory and Practice of Logic Programming (2023)
3. Gomes, C., Kautz, H., Sabharwal, A., Selman, B.: Satisfiability solvers. In: van Harmelen, F., Lifschitz, V., Porter, B. (eds.) Handbook of Knowledge Representation, pp. 89–134. Elsevier (2008)
4. Kovaćs, L., Voronkov, A.: First-order theorem proving and Vampire. In: International Conference on Computer Aided Verification. pp. 1––35 (2013)
5. Pearce, D., Valverde, A.: Synonymous theories and knowledge representations in answer set programming. Journal of Computer and System Sciences **78**, 86–104 (2012)