Translating Definitions into the Language of Logic Programming: A Case Study

Vladimir Lifschitz

University of Texas at Austin, USA
vl@cs.utexas.edu

Abstract
In the process of creating a declarative program, the programmer transforms a problem specification expressed in a natural language into an executable specification. We study the case when the given specification is expressed by a mathematically precise definition, and the goal is to write a program for an answer set solver.

Keywords
declarative programming, programming methodology, infinite stable models, safety

1. Introduction

Transition from a problem specification to a declarative program is essentially a translation: the programmer reformulates a specification expressed in a natural language (intermixed with mathematical notation) as an executable specification. In the discussion of answer set programming (ASP), Gelfond and Kahl [6] observe that

> as in any translation, the translation of even simple English sentences to statements in ASP may be a nontrivial task . . . Any type of translation is an art, and translation into ASP is no exception.

In this paper, we examine the case when the given specification is expressed by an informal, but mathematically precise, definition, and the declarative program that has been constructed is written in the core language of answer set programming [1]. Representing definitions is not a typical use of answer set programming languages if ASP is viewed as a method for solving combinatorial search problems. But definitions are an integral part of the generate-define-test approach to search [9]. Furthermore, ASP is not only about search; it is a general purpose knowledge representation paradigm.

Translating mathematical definitions is relatively easy, for a number of reasons. Mathematics is easier to formalize than commonsense ideas, such as beliefs and causation. Since incomplete information is not involved, there is no need to distinguish between negation as failure and classical negation. The rules produced by translating a definition have an atom in the head; disjunctive rules, choice rules, and constraints are not used in the translation.

Nevertheless, translating mathematical definitions is nontrivial and deserves careful study. Specifically, we consider here the definition of prime numbers:

A prime is a natural number greater than 1
that is not a product of two smaller natural numbers. (1)

2. Expressing the definition in formal notation

The most conspicuous feature of programming languages, both procedural and declarative, is that they use formal notation. Many mathematical assertions are easy to express in the formal language of first-order logic—this is what that language was invented for. In particular, definition (1) can be written as the first-order formula

$$p(n) \leftrightarrow n > 1 \ \wedge \ \neg\exists ij(n = i \times j \ \wedge \ i < n \ \wedge \ j < n), \tag{2}$$

where the variables range over nonnegative integers. The universal closure of this formula has a unique model that is *standard* in the sense that its universe consists of all nonnegative integers, and that the symbols

$$1 \quad \times \quad > \quad <$$

are interpreted in the usual way. In this model, the extent of p is the set of all primes.

In logic programming, the same idea can be expressed by the rule obtained from (2) by replacing the equivalence sign by a left arrow:

$$p(n) \leftarrow n > 1 \ \wedge \ \neg\exists ij(n = i \times j \ \wedge \ i < n \ \wedge \ j < n). \tag{3}$$

This expression is not very close to executable code, but it is similar to rules discussed in many theoretical publications. For instance, Lloyd and Topor [10] defined an *extended program clause* as a first-order formula of the form $A \leftarrow W$, where A is an atom and W is an arbitrary first-order formula. They observed that program completion [2] can be generalized to extended clauses in a straightforward way. The relationship between formulations (2) and (3) can be described by saying that the universal closure of the former is the completed definition of p in the latter.

This example illustrates the fact that definitions in the sense of logic programming are not so different from definitions in the sense of first-order logic, provided that we allow bodies of rules to be arbitrarily complex. The view accepted in logic programming—a definition is a set of rules with the same predicate symbol in the head—is actually more general. Equivalences like (2) correspond to the logic programming definitions consisting of a single non-recursive rule such that the arguments of the atom in its head are distinct variables. If a definition in a logic program consists of several rules, and/or the arguments in the heads are not distinct variables, then transforming that definition into an equivalence becomes more involved. If the definition is recursive (as for instance, the definition of the transitive closure) then transforming it into a first-order equivalence may be impossible.

The stable model semantics [7] can be extended to arbitrary first-order sentences using the syntactic transformation SM [3]. Under some conditions, the result of applying that operator

is equivalent to completion [3, Theorem 11]. In particular, the result of applying SM_p to the universal closure of implication (3) is equivalent to the universal closure of (2). The subscript p in the symbol SM_p indicates that the predicate symbol p is treated as intensional, and the other predicate symbols occurring in the formula (in this case, $>$ and $<$) are considered extensional.

The definition of SM is based on the view that the negation sign represents negation as failure; in equivalence (2), this sign is understood as classical negation. This difference is inessential, however, because the negated formula in this case does not contain intensional symbols.

In the rest of this paper, variables in rules will be capitalized, as usual in logic programming. In particular, we will write rule (3) as

$$p(N) \leftarrow N > 1 \wedge \neg\exists IJ(N = I \times J \wedge I < N \wedge J < N). \tag{4}$$

3. Making the answer finite

Standard Prolog systems answer queries about stable models of logic programs. A standard answer set solver, on the other hand, is designed to display the entire model of the program, or at least the extents of the predicates that it is instructed to show. For this reason, stable models of an ASP program are expected to be finite [1, Section 5]. Since the set of all primes is infinite, rule (4) violates this condition, and we need to replace it by the definition of a finite set.

Let a, b be object constants ("placeholders") representing integers such that $b \geq a > 1$. Consider the rule obtained from (4) by replacing $N > 1$ with $a \leq N \leq b$:

$$p(N) \leftarrow a \leq N \leq b \wedge \neg\exists IJ(N = I \times J \wedge I < N \wedge J < N). \tag{5}$$

It defines the set of all primes in the interval $\{a, \dots, b\}$. Since this interval can be arbitrarily large, the modified definition is as general as the definition (4) of the set of all primes.

4. Simplifying the body

The body of rule (5) is syntactically more complex than expressions allowed in the core language of answer set programming. We can get around this difficulty using an auxiliary predicate.[1] The atomic formula $aux(N)$ will represent the subformula

$$\exists IJ(N = I \times J \wedge I < N \wedge J < N). \tag{6}$$

Rule (5) will be rewritten as

$$p(N) \leftarrow a \leq N \leq b \wedge \neg aux(N), \tag{7}$$

[1] The input language of the answer set solver CLINGO includes conditional literals, which correspond to universally quantified implications in the body of a rule. A conditional literal can be used to convert (5) into a CLINGO rule without auxiliary predicates, because formula (6) can be rewritten as

$$\forall IJ(I < N \wedge J < N \rightarrow N \neq I \times J).$$

Conditional literals are not part of the core language and are not available in the current version of the solver DLV.

and *aux* can be defined by the rule

$$\mathit{aux}(N) \leftarrow \ \exists IJ(N = I \times J \ \wedge \ I < N \ \wedge \ J < N).$$

Furthermore, the meaning of the last rule will not change if we drop the existential quantifier from its body:

$$aux(N) \leftarrow \ N = I \times J \ \wedge \ I < N \ \wedge \ J < N. \tag{8}$$

For any choice of a and b, program (7), (8) has a unique standard stable model, and the extent of p in that model is the set of all primes in the interval $\{a, \ldots, b\}$.

The transformation described in this section is an example of the general process implemented in the translator F2LP [8]. That process is applicable to first-order formulas that contain any propositional connectives and quantifiers.

Program (7), (8) looks pretty much like a typical piece of ASP code, modulo simple syntactic changes, such as replacing $\leftarrow$ with `:-`, $\wedge$ with a comma, and $\neg$ with `not`. A little more work is needed, however, to convert it into an input for an answer set solver.

5. Making the entire stable model finite

One of the reasons why program (7), (8) is not completely satisfactory is that the problem of infinite stable models, discussed in Section 3, has reappeared with the introduction of *aux*. The extent of this predicate is an infinite set—it includes 0, 1, and all composite numbers. The extent of the predicate p that we would like to see displayed on the screen has not changed; it is the same finite set as before. But the entire stable model of program (7), (8) is infinite.

We can deal with this difficulty by modifying the auxiliary predicate. In the body of rule (7), $\neg \mathit{aux}(N)$ is conjoined with the condition $a \leq N \leq b$. Consequently the meaning of the rule will not be affected if we understand $\mathit{aux}(N)$ in a different way:

N is a number *between* a *and* b that equals 0 or 1 or is composite.

This can be accomplished by adding $a \leq N \leq b$ as a conjunctive term to the body of rule (8):

$$aux(N) \leftarrow \ a \leq N \leq b \wedge N = I \times J \ \wedge \ I < N \ \wedge \ J < N. \tag{9}$$

The stable model of program (7), (9) is finite, and the extent of p in that model is again the set of all primes in the interval $\{a, \ldots, b\}$.

6. Allowing variables to take general values

In the rules above, the symbols I, J, N are understood as variables for nonnegative integers. On the other hand, the set of values that can be assigned to variables by a global substitution includes not only integers, but also symbolic constants and other expressions [1, Section 3]. The only restriction is that the evaluation of arithmetic subterms is required to be well-defined; for instance, substituting non-integer values for I, J in a rule containing the term $I \times J$ is not allowed.

Because of this difference, the meaning of program (7), (9) in the core language of ASP can be different from its understanding adopted in the discussion above. Rule (7) is not problematic in this sense, because the condition $a \leq N \leq b$ in its body restricts the values of N to integers that are greater than or equal to a, and a is expected to be a positive integer (Section 3). But the body of rule (9) can be satisfied for substitutions that make I and J negative, for instance

$$N = 7,\ I = -1,\ J = -7.$$

We will make such substitutions harmless by saying explicitly in the body of the rule that I and J are nonnegative:

$$aux(N) \leftarrow\ a \leq N \leq b \wedge N = I \times J\ \wedge\ 0 \leq I < N\ \wedge\ 0 \leq J < N. \tag{10}$$

7. Helping the grounder to ground

Rules (7) and (10) need to be further modified, because they are not safe.

The definition of safety [1, Section 5] is recursive and rather complicated; but violating the safety requirement usually causes the solvers CLINGO and DLV to terminate with an error message.[2] In a safe rule, every global variable has at least one occurrence in the body that either belongs to a nonnegated atom or is the left-hand side of an equality. A condition of this kind is needed to facilitate the process of grounding. An ASP solver calculates a finite set of "essential values" of each variable in each rule of the program such that substituting inessential values is not needed for generating stable models. This is a prerequisite for transforming a program with variables into a finite ground program.

Rule (7) is not safe, because one of the two occurrences of N in its body is within an inequality, and the other is part of a negated atom. Rule (10) is not safe because of the variables I, J.

The interval construct $(..)$ provides a way out.[3] The expression $N = a..b$ ("the value of N belongs to the interval $\{a, \ldots, b\}$") has almost the same meaning as the inequality $a \leq N \leq b$; the difference is that the latter can be satisfied even for non-integer values of a, N, and b.[4] The other two inequalities in the body of (10) can be replaced by interval expressions in a similar way.

Now we are done. The program

[2] There are exceptions, however. CLINGO does not complain, for example, about the unsafe rule

```
p(N) :- 3*N+1 = 10.
```

Rules (7) and (10) will be groundable by Version 5.6.0 of CLINGO, which is in the works at the time of this writing (Roland Kaminski, personal communication, June 16, 2022).

[3] This construct is available in the input languages of CLINGO and DLV, but it is not mentioned in the ASP-Core-2 document [1]. The authors of the document view this now as an oversight (Martin Gebser, personal communication, April 10, 2022).

[4] In this context, the set membership symbol $\in$ between N and $a..b$ would look more natural than the equality sign. But the general view adopted in the input language of CLINGO is that a ground term may have several values [4, Section 4.2], and a comparison $t_1 \prec t_2$ in the body of a rule expresses that the relation $\prec$ holds between some value of t_1 and some value of t_2. For instance, each of the comparisons $2 = 1..3$, $1..3 = 2$ expresses that the only value of 2 equals one of the three values of $1..3$. The comparison $1..3 = 3..5$ in the body of a rule is true because its left-hand side and right-hand side have a common value.

```
#const a = 20.
#const b = 30.
  p(N) :- N = a..b, not aux(N).
aux(N) :- N = a..b, N = I*J, I = 0..N-1, J = 0..N-1.
#show p/1.
```

can be successfully grounded by CLINGO, and the answer

```
p(23) p(29)
```

will be displayed on the screen.

8. Conclusion

The process of translating the English language definition of primes into executable code described in this note involved

- rewriting the definition in formal notation,
- modifying it to make the answer finite,
- introducing an auxiliary definition to make the body simpler,
- making the entire stable model finite,
- allowing variables to take arbitrary values, and
- helping the grounder to ground the program.

This is somewhat similar to the process of stepwise refinement in procedural programming, in the sense that each step took us closer to executable code. There is a difference, however. Stepwise refinement makes the outline of the algorithm longer and longer, as the programmer breaks down steps into substeps. Our transformations, on the other hand, had almost no effect on the size of the specification.

The final result is the first version of executable code; we did not talk here about the subsequent process of improving it in the sense of making the program more efficient. An example of the latter is the topic of Section 3.1 of the paper by Gebser *et al.* [5], where a series of encodings of the n-queens problem is presented, each of them an improvement of the one before. Case studies of both kinds contribute to developing and clarifying the methodology of answer set programming.

> The detailed elaborations on the development of even a short program form a long story, indicating that careful programming is not a trivial subject. If this paper has helped to dispel the widespread belief that programming is easy as long as the programming language is powerful enough and the available computer is fast enough, then it has achieved one of its purposes [11].

Acknowledgements

Many thanks to Jorge Fandinno, Michael Gelfond, Roland Kaminski, Yuliya Lierler, Jayadev Misra and the anonymous referees for comments on a draft of this paper. Jessica Zangari and Martin Gebser helped me by answering questions about the input language of DLV and about the core language of answer set programming.

References

[1] Francesco Calimeri, Wolfgang Faber, Martin Gebser, Giovambattista Ianni, Roland Kaminski, Thomas Krennwallner, Nicola Leone, Marco Maratea, Francesco Ricca, and Torsten Schaub. ASP-Core-2 input language format. *Theory and Practice of Logic Programming*, 20:294–309, 2020.

[2] Keith Clark. Negation as failure. In Herve Gallaire and Jack Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, New York, 1978.

[3] Paolo Ferraris, Joohyung Lee, and Vladimir Lifschitz. Stable models and circumscription. *Artificial Intelligence*, 175:236–263, 2011.

[4] Martin Gebser, Amelia Harrison, Roland Kaminski, Vladimir Lifschitz, and Torsten Schaub. Abstract Gringo. *Theory and Practice of Logic Programming*, 15:449–463, 2015.

[5] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. Challenges in answer set solving. In *Logic programming, knowledge representation, and nonmonotonic reasoning. Essays Dedicated to Michael Gelfond on the Occasion of His 65th Birthday*, pages 74–90. Springer, 2011.

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

[7] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In Robert Kowalski and Kenneth Bowen, editors, *Proceedings of International Logic Programming Conference and Symposium*, pages 1070–1080. MIT Press, 1988.

[8] Joohyung Lee and Ravi Palla. System F2LP — computing answer sets of first-order formulas. In *Proceedings of 10th International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 515–521, 1988.

[9] Vladimir Lifschitz. What is answer set programming? In *Proceedings of the 23rd AAAI Conference on Artificial Intelligence*, pages 1594–1597. MIT Press, 2008.

[10] John Lloyd and Rodney Topor. Making Prolog more expressive. *Journal of Logic Programming*, 1:225–240, 1984.

[11] Niklaus Wirth. Program development by stepwise refinement. *Communications of the ACM*, 14(4):221–227, 1971.