Answer Set; Programming? ⋆

Pedro Cabalar

Dept. Computación,
University of Corunna (Spain)
cabalar@udc.es

Abstract. Motivated by a discussion maintained by Michael Gelfond and other researchers, this short essay contains some thoughts and refections about the following question: should ASP be considered a programming language?

1 Introduction

During a break in the Commonsense[1] 2007 Symposium, sat around a table in some cafeteria inside the Stanford campus, an interesting and solid debate between Michael Gelfond and some well-known researcher (call him A.B.) in the area of Logic Programming and Nonmonotonic Reasoning was initiated. The discussion started when A.B. claimed at some point that Answer Set Programming (ASP) was a *programming* paradigm. Although at that moment, this seemed a quite obvious, redundant and harmless assertion, surprisingly, Gelfond's reaction was far from agreement. With his usual kind but firm, rational style, he proceeded to refute that argument, defending the idea that ASP was exclusively a logical knowledge representation language, *not* a programming language. This short essay contains some disconnected, personal thoughts and reflections motivated by that discussion.

2 Programming as implementing algorithms

Of course, the keypoint relies on what we consider to be a *programming language* or more specifically which is the task of *programming*. In its most frequent use, this term refers to *imperative* programming, that is, specifying sets of instructions to be executed by a computer in order to achieve some desired goal. This is opposed to *declarative* programming, which is frequently defined as specifying *what* is to be considered a solution to a given problem, rather than the steps describing *how* to achieve that solution. ASP would perfectly fit in this second definition, had not it been the case that declarative programming actually involves much more than this ideal goal of pure formal specification.

⋆ This work was partially supported by Spanish MEC project TIN2009-14562-C05-04 and Xunta de Galicia project INCITE08-PXIB105159PR.

[1] Eighth International Symposium on Logical Formalizations of Commonsense Reasoning.

In practice, despite of the big differences between imperative and declarative programming languages, we can generally identify *programming* as *implementing algorithms*. We actually expect that a programming language allows us a way to execute algorithms on a computer. When a (declarative) programmer designs a program, either in Prolog or in some functional language, she must not only write a formal specification (using predicates, functions, equations, etc), but also be aware of how this specification is going to be processed by the language interpreter she is using, so that both things together become the algorithm implementation. This idea was clearly expressed by Kowalski's equation [7] "Algorithm = Logic + Control" or $A = L + C$, meaning that a given algorithm A can be obtained as the sum of a logical description L (our formal specification) plus a control strategy C (top-down, bottom-up, SLD, etc). The same algorithm A can be the result of different combinations, say $A = L_1 + C_1$ or $A = L_2 + C_2$. Efficiency issues and even termination will strongly depend on the control component C we choose. If we use an interpreter like Prolog, where the control component is *fixed*, we will face the typical situation where the logic component L must be adapted to the control strategy. A simple change in the ordering among clauses or literals in L may make the program to terminate or not.

Now, back to our concern for ASP, despite of being an LP paradigm, it seems evident that the algorithmic interpretation for Predicate Logic [16] (which was in the very roots of the LP area) is not applicable here. In fact, in an ASP program, there is no such a thing as a "control strategy." In this sense, we reach a first curious, almost paradoxical, observation:

Observation 1 *ASP is a purely declarative language, in the sense that it exclusively involves formal specification, but cannot implement an algorithm, that is, cannot execute a program!*

Thus, ASP fits so well with the ideal of a declarative programming language (i.e., telling *what* and not *how*) that it does not allow programming at all.

3 Programming as temporal problem solving

The previous observation saying that ASP cannot implement an algorithm may seem too strong. One may object that ASP has been extensively used for solving temporal problems in transition systems, including planning, that is, obtaining a sequence of actions to achieve a desired goal. This may look close to algorithm implementation. Let us illustrate the idea with a well-known example.

Example 1 (Towers of Hanoi). We have three vertical pegs a, b, c standing on a horizontal base and a set of n holed disks, numbered $1, \ldots, n$, whose sizes are proportional to the disk numbers. Disks must always be stored on pegs, and a larger disk cannot rest on a smaller one. The HANOI problem consists in moving a tower formed with the n disks in peg a to peg c, by combination of individual movements that can carry the top disk of some peg on top of another peg. □

Figure 1 shows an ASP program for solving HANOI in the language of `lparse`[2]. Constant `n` represents the number of disks and constant `pathlength` the number of transitions, so that, we must make iterative calls to the solver varying `pathlength=0,1,2,...` until a solution is found. For instance, for `n=2` the first solution is found with `pathlength=3` and consists in the sequence of actions `move(a,b,0) move(a,c,1) move(b,c,2)`; for `n=3`, the minimal solution is `pathlength=7` obtaining the sequence:

```
move(a,c,0) move(a,b,1) move(c,b,2) move(a,c,3) move(b,a,4)
move(b,c,5) move(a,c,6)
```

whereas for `n=3` with `pathlength=15` we get:

```
move(a,b,0) move(a,c,1) move(b,c,2) move(a,b,3) move(c,a,4)
move(c,b,5) move(a,b,6) move(a,c,7) move(b,c,8) move(b,a,9)
move(c,a,10) move(b,c,11) move(a,b,12) move(a,c,13) move(b,c,14)
```

Although the ASP program correctly solves the HANOI problem, it is far from being an algorithm. In particular, *we would expect that an algorithm told us how to proceed in the general case for an arbitrary number n of disks.* This means finding some general process from which the above sequences of actions can be extracted once we fix the parameter n. For instance, in the sight of the three solved instances above, it is not trivial at all how to proceed for $n = 4$.

In the case of HANOI, such a general process to generate an arbitrary solution does exist. In fact, the HANOI problem is a typical example extensively used in programming courses to illustrate the idea of a recursive algorithm. We can divide the task of shifting n disks from peg X to peg Y, using Aux as an auxiliary peg, into the general steps

1. Shift $n-1$ disks from X to Aux using Y as auxiliary peg;
2. Move the n-th disk from X to Y;
3. Shift $n-1$ disks from Aux to Y using X as auxiliary peg.

This recursive algorithm is encoded in Prolog in Figure 2, where predicate `hanoi(N,Sol)` gets the number of disks `N` and returns the solution `Sol` as a list of movements to be performed.

Note now the huge methodological difference between both programs. On the one hand, the ASP program exclusively contains a formal description of HANOI but cannot tell us how to proceed in a general case, that is, cannot yield us a general pattern for the sequences of actions for an arbitrary n. On the other hand, the Prolog program (or any implementation[3] of the recursive algorithm)

[2] `http://www.tcs.hut.fi/Software/smodels/lparse.ps`

[3] We could perfectly use instead any programming language allowing recursive calls. In the same way, Prolog can also be used to implement a planner closer to the constraint-solving spirit of the ASP program.

```
%---- Types and typed variables
disk(1..n).                    peg(a;b;c).
transition(0..pathlength-1).  situation(0..pathlength).
location(Peg) :- peg(Peg).    location(Disk) :- disk(Disk).
#domain disk(X;Y).     #domain peg(P;P1;P2).  #domain transition(T).
#domain situation(I).  #domain location(L;L1).

%---- Inertial fluent: on(X,L,I) = disk X is on location L at time I
on(X,L,T+1) :- on(X,L,T), not otherloc(X,L,T+1).   % inertia
otherloc(X,L,I) :- on(X,L1,I), L1!=L.
:- on(X,L,I), on(X,L1,I), L!=L1.                   % on unique location

%---- Defined fluents
%   inpeg(L,P,I) = location L is in peg P at time I
%   top(P,X,I) = location L is the top of peg P. If empty, the top is P
inpeg(P,P,I).
inpeg(X,P,I) :- on(X,L,I), inpeg(L,P,I).
top(P,L,I) :- inpeg(L,P,I), not covered(L,I).
covered(L,I) :- on(X,L,I).

%---- State constraint: no disk X on a smaller one
:- on(X,Y,I), X>Y.

%---- Effect axiom
on(X,L,T+1) :- move(P1,P2,T), top(P1,X,T), top(P2,L,T).

%---- Executability constraint
:- move(P1,P2,T), top(P1,P1,T).     % the source peg cannot be empty

%---- Generating actions
movement(P1,P2) :- P1 != P2.        % valid movements
1 {move(A,B,T) : movement(A,B) } 1. % pick one at each transition T

%---- Initial situation
on(n,a,0).         on(X,X+1,0) :- X<n.

%---- Goal: at last situation, all disks in peg c
onewrong :- not inpeg(X,c,pathlength).
:- onewrong.
```

Fig. 1. An ASP program to solve the HANOI problem.

```
hanoi(N,Sol) :- shift(N,a,c,b,Sol).

shift(0,_,_,_,[]) :- !.
shift(N,X,Y,Aux,Sol) :- N1 is N-1,
                        shift(N1,X,Aux,Y,Pre),
                        append(Pre,[move(X,Y)|Post],Sol),
                        shift(N1,Aux,Y,X,Post).
```

Fig. 2. A Prolog program implementing a recursive algorithm to solve HANOI.

tells us the steps that must be performed in a compact (and in fact, much more efficient) way, but contains *no information at all* about the original problem. The recursive algorithm is just a "solution generator" or if preferred, a regular way of describing the structure of sequences of actions that constitute a solution.

This naturally leads to the following topic: *verification*. How can we guarantee that the actions recursively generated for some n actually move our tower to the desired target without violating the problem constraints? Verifying the recursive algorithm is a crucial work, since it contains no information on the original puzzle. In the case of the ASP encoding, since it already constitutes a formal specification, verification is a much more subtle task. It would require providing a second, different enough, formal specification (perhaps using mathematical objects closer to the original problem) and proving afterwards that the answer sets we obtain are in one-to-one correspondence to the solutions of our second representation.

Together with verification, another typical issue in algorithm analysis is *complexity*. The recursive algorithm can be easily used to prove that a solution to HANOI requires $2^n - 1$ steps. Obtaining this complexity result from the ASP program (plus the iteration of `pathlength`) is far from trivial. Although the complexity of arbitrary ASP is well-known, a different open and interesting question is *how to use ASP for complexity analysis of a given encoded temporal problem.*

One final comment that stresses the difference between both programs is that, obviously, the recursive algorithm approach is not elaboration tolerant at all. A simple variation of HANOI that allowed a fourth peg would lead to shorter solutions (for instance, with $n = 4$ a solution is found in 9 steps). Note that the only change in the ASP representation would just require adding the fact `peg(d)`. It is easy to think about simple variations that would even make the Prolog program to become incorrect.

4 Programming as implementing a Turing machine

One of the usually desirable properties of programming languages is *Turing completeness*, that is, the capability of capturing any computation that can be performed by a Turing machine. It is well-known [15] that Prolog (in fact Horn clauses with functions) is Turing complete. Figure 3 shows one possible encoding of a generic Turing machine in Prolog. It just consists of five rules for predicate

`tm(S,L,X,R)` which represents a machine configuration. The current state is represented by argument `S` and the tape is fragmented into three portions: `L` is the (reversed) list of symbols to the left of the machine head; `X` is the tape symbol currently pointed by the head; and `R` is the list of symbols to the right of the head. Constant `0` stands for the blank symbol. Predicate `nexttm` is just a recursive call to `tm` preceded by a display output of the new configuration.

```
tm(S, _, _, _)      :- final(S).
tm(S,[Y|L],X,  R  ) :- t(S,X,S1,X1,l), nexttm(S1,L,Y,[X1|R]).
tm(S,  L,  X,[Y|R]) :- t(S,X,S1,X1,r), nexttm(S1,[X1|L],Y,R).
tm(S,  L,  X,  [] ) :- t(S,X,S1,X1,r), nexttm(S1,[X1|L],0,[]).
tm(S,  [], X,  R  ) :- t(S,X,S1,X1,l), nexttm(S1,[],0,[X1|R]).

nexttm(S,L,X,R) :- write(tm(S,L,X,R)),nl,tm(S,L,X,R).
```

Fig. 3. A general implementation of a Turing machine in Prolog.

In order to implement a particular machine, we just have to include facts for the predicates `t/5` (the transition table), and `final/1` (which states are final), assuming that no transition is given for a final state. As an example, the following facts would specify a 3 state, 2 symbol busy beaver:

```
t(a,0,b,1,r).  t(b,0,a,1,l).    t(c,0,b,1,l). t(a,1,c,1,l).
t(b,1,b,1,r).  t(c,1,halt,1,r). final(halt).
```

The execution of the busy beaver on an empty tape beginning with state '`a`' would correspond to the query call shown in Figure 4.

Back again to ASP, under its most accepted understanding as a logic programming paradigm [8,13], one of its characteristic features is the complexity class it can cover: NP-completeness for normal logic programs [9]; Σ_2^P-completeness for disjunctive programs [4]. Both results refer to existence of a stable model for a *propositional* program. The use of variables as abbreviations of all their possible ground instances to obtain a finite propositional program is possible thanks to another fundamental feature of the traditional ASP paradigm: forbidding function symbols. These complexity bounds point out that, although useful for solving many constraint-like problems, ASP is far from being a Turing-complete programming language.

However, this "traditional" picture needs to be revised in the sight of recent results obtained during the last years. First, in the theoretical field, the classical definition of stable models [6], which was only applicable to propositional programs, has been extended for covering any arbitrary first order theory, thanks to the definition of *First Order Equilibrium Logic* [14], a nonmonotonic formalism relying on a monotonic intermediate logic, or the equivalent *General Theory of*

```
?- nexttm(a,[],0,[]).
tm(a, [], 0, [])
tm(b, [1], 0, [])
tm(a, [], 1, [1])
tm(c, [], 0, [1, 1])
tm(b, [], 0, [1, 1, 1])
tm(a, [], 0, [1, 1, 1, 1])
tm(b, [1], 1, [1, 1, 1])
tm(b, [1, 1], 1, [1, 1])
tm(b, [1, 1, 1], 1, [1])
tm(b, [1, 1, 1, 1], 1, [])
tm(b, [1, 1, 1, 1, 1], 0, [])
tm(a, [1, 1, 1, 1], 1, [1])
tm(c, [1, 1, 1], 1, [1, 1])
tm(halt, [1, 1, 1, 1], 1, [1])
Yes
```

Fig. 4. A query and its corresponing display output for the 3 state, 2 symbol busy beaver machine.

Stable Models [5], a syntactic construct very close to Circumscription [12]. Under these extensions, we can even remove the restriction to Herbrand models, so that assumptions like, for instance, domain closure or unique names are now optional. Thus, at least as a theoretical device, the new generalisations of ASP can now perfectly deal[4] with (Herbrand models) of the encoding of a Turing machine we presented in Figure 3.

But this capability is not limited to the theoretical field. A second important recent breakthrough has been the introduction of functions in ASP [2] and its implementation with solver `DLV-complex` [3]. When a program with functions, disjunction and negation satisfies a given property, so-called being *finitely-ground*, it has nice computational features: brave and cautious reasoning become decidable, and its answer sets are computable. An interesting result is that finitely-ground programs can encode any computable function. This was proven by *encoding a Turing machine as an ASP program* so that a function computation stops in the machine iff its ASP encoding is finitely recursive (and the answer set will contain the execution steps). As it can be expected, checking whether a program is finitely recursive is undecidable. In fact, the encoding of Figure 3 is practically the same one[5] recently used in [1] to prove Turing-completeness of ASP with functions. In practical terms, this means that we can feed a program similar to the one in Figure 3 to `DLV-complex` and, as the machine halts, obtain one answer set containing all the facts for `tm` shown in Figure 4.

At the sight of this new scenario, we must reconsider our main question: should ASP be considered now a programming language? Capturing a Turing

[4] A correctness proof of this program with respect to any of these two first order formalisms would be interesting.

[5] The original encoding considered a version of Turing machine with a left-ended tape.

machine looks an undeniable proof. However, we can still find a subtle distinction between ASP and Prolog behaviours. The ASP encoding is being used to *represent* a Turing machine and its computations, whereas the Prolog encoding actually *executes* those computations. We can associate a time stamp to each of the ordered lines that the computer prints to solve the Prolog query: time is "real." Contrarily, the answer set would contain the same set of `tm` facts, but with *no associated ordering*. In fact, if we wanted to reconstruct the order among the steps followed by the machine, we would have to include one more parameter for an explicit representation of time. Thus, in ASP time is "virtual."

Observation 2 *Although ASP and Prolog can encode a Turing machine, ASP* represents *the machine computations using virtual, reified time, whereas Prolog* executes *the machine computations using real time.*

Under our point of view, this observation reaffirms the idea of seeing ASP as a formal specification language rather than a programming language. Note how the ASP orientation could be used to *analyse* a real-time application or a reactive system, but not to *implement* it in a real scenario.

5 Programming as a craft

Finally, one more meaning we may sometimes associate to the idea of programming to distinguish it from formal specification is that the former involves some kind of craft work whereas the latter is frequently seen as an accurate task and free from efficiency issues. In the particular case of Prolog programs, we all know that practical programming involves capabilities like a reasonable use of the cut predicate or an efficient list construction strategy. This usually means important sacrifices in program readability and declarativeness (the simple inclusion of a cut predicate may easily change the program semantics).

Unfortunately, efficiency considerations are also present in practical ASP (mostly related to reduce grounding) and, as happens with Prolog, they frequently introduce a sacrifice with respect to the program quality too. As ASP is a formal specification language, this sacrifice does not have to do with a lack of declarativeness – we can say that ASP programs are always declarative. The cost to pay may come as a lack of *elaboration tolerance* [10]. As defined by John McCarthy:

> *"A formalism is* elaboration tolerant *to the extent that it is convenient to modify a set of facts expressed in the formalism to take into account new phenomena or changed circumstances."*

The quest for elaboration tolerance was present in ASP from its very beginning. Stable models and their use of default negation allowed establishing a fruitful connection between Logic Programming and Nonmonotonic Reasoning (NMR). We can even say that ASP constitutes nowadays one of the most successful tools for practical NMR. However, the fact that a language is elaboration

tolerant (up to some degree) does not necessarily mean that any program built with that language is elaboration tolerant too. When we talk about elaboration tolerance of ASP, we mean that there exists a flexible way of representing a problem, not that *any* encoding of that problem in ASP will have the same degree of elaboration tolerance, or that it will have that property at all.

For instance, back to our HANOI problem, we will have few actions (and so, few choice points) if we consider, as we did, movements from one peg to another: for three pegs, this always means six possible actions. On the contrary, if we take, for instance, movements from a given disk to a given location (a peg base or another disk), the number of possible actions blows up as n increases. The peg-to-peg representation is, however, less elaboration tolerant, as it is more focused on the given problem we try to solve rather than the physical possibilities of the scenario. As an example, assume now that some marked disks could be carried with all the disks they have above at a time. The disk-to-location encoding would still be valid, but the peg-to-peg representation no, as it implicitly assumes that we always take the top disk. Decisions like this arise in almost any ASP problem solving project.

To further illustrate this dilemma, think about the Second ASP Solver Competition[6], where efficiency plays a preeminent role, as expected. In this edition, the competition just defined the input and output format of the benchmark problems, and the final encoding was left to the competitors. This idea had the important advantage of opening the competition to solvers that accepted *different languages*, so they could compete altogether for a faster solution. The disadvantage, however, is obvious: we are measuring not only the performance of a given solver, but also the craft or experience of the ASP programmer to obtain a more "efficient representation" (usually, a less grounding-consuming one). Furthermore, if we look at the three ASP encodings available in the competition site for the HANOI problem we will find out that elaboration tolerance sacrifices have been considerable. None of the three solutions contain the default rule of *inertia* to avoid the frame problem [11], which has been the cornerstone of the NMR area of Reasoning about Actions and Change. In fact, the use of negation is quite limited (no defaults are really used) and two of the encodings contain an automaton-style description, which is probably one of the less elaboration tolerant ways of describing a problem (any slight variation usually means rebuilding the whole automaton).

Although we recognize the crucial importance of an efficiency competition for an active improvement of the available solvers (as happened in the SAT area), it is perhaps worth to consider a different track including benchmarks focused on an fixed, elaboration tolerant encoding of a given problem. After all, our final goal should proving that *efficient elaboration tolerance* is feasible. We can summarize this focusing by ending up with another quote by McCarthy, when talking about the use of chess as a drosophila for AI:

> *"Unfortunately, the competitive and commercial aspects of making computers play chess have taken precedence over using chess as a scientific*

[6] `http://dtai.cs.kuleuven.be/events/ASP-competition/index.shtml`

domain. It is as if the geneticists after 1910 had organized fruit fly races and concentrated their efforts on breeding fruit flies that could win these races."

6 Conclusions

After examining several aspects of the idea of programming, we claim that ASP is not a programming paradigm in a strict sense, but a formal specification language instead. Still, we find that the term Answer Set *programming* can be more vaguely used to refer to the craft (and perhaps in the future, to the methodologies) for developing an efficient and elaboration tolerant formal representation of a given problem.

Acknowledgements Special thanks to Michael Gelfond for his always inspiring and enlightening discussions - hearing or reading him always means extracting new valuable ideas. Also thanks to Ramón P. Otero and Alessandro Provetti, who introduced me to Dr. Gelfond and brought my attention to his work some years ago.

References

1. Mario Alviano, Wolfgang Faber, and Nicola Leone. Disjunctive ASP with functions: Decidable queries and effective computation. *Theory and Practice of Logic Programming*, 10(4-6):497–512, 2010.
2. Francesco Calimeri, Susanna Cozza, Giovambattista Ianni, and Nicola Leone. Computable functions in ASP: Theory and implementation. In *24th Intl. Conf. on Logic Programming*, volume 5366 of *Lecture Notes in Computer Science*, pages 407–424. Springer-Verlag, 2008.
3. Francesco Calimeri, Susanna Cozza, Giovambattista Ianni, and Nicola Leone. An ASP system with functions, lists, and sets. In *10th Intl. Conf. on Logic Programming and Nonmonotonic Reasoning*, volume 5753 of *Lecture Notes in Computer Science*, pages 483–489. Springer-Verlag, 2009.
4. Thomas Eiter and Georg Gottlob. Complexity results for disjunctive logic programming and application to nonmonotonic logics. In *Proceedings of the International Logic Programming Symposium (ILPS*, pages 266–278. MIT Press, 1993.
5. P. Ferraris, J. Lee, and V. Lifschitz. A new perspective on stable models. In *Proc. of the International Joint Conference on Artificial Intelligence (IJCAI'07)*, pages 372–379, 2004.
6. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R. A. Kowalski and K. A. Bowen, editors, *Logic Programming: Proc. of the Fifth International Conference and Symposium (Volume 2)*, pages 1070–1080. MIT Press, Cambridge, MA, 1988.
7. R. Kowalski. Algorithm = logic + control. *Communications of the ACM*, 22(7):424–436, 1979.
8. V. Marek and M. Truszczyński. *Stable models and an alternative logic programming paradigm*, pages 169–181. Springer-Verlag, 1999.

9. W. Marek and M. Truszczyński. Autoepistemic logic. *Journal of the ACM*, 38:588–619, 1991.
10. J. McCarthy. Elaboration tolerance. In *Proc. of the 4th Symposium on Logical Formalizations of Commonsense Reasoning (Common Sense 98)*, pages 198–217, London, UK, 1998. Updated version at `http://www-formal.stanford.edu/jmc/elaboration.ps`.
11. J. McCarthy and P. Hayes. Some philosophical problems from the standpoint of artificial intelligence. *Machine Intelligence Journal*, 4:463–512, 1969.
12. John McCarthy. Circumscription - a form of non-monotonic reasoning. *Artif. Intell.*, 13(1-2):27–39, 1980.
13. I. Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25:241–273, 1999.
14. David Pearce and Agustín Valverde. Towards a first order equilibrium logic for nonmonotonic reasoning. In *Proc. of the 9th European Conf. on Logics in AI (JELIA'04)*, pages 147–160, 2004.
15. Sten-Ake Tärnlund. Horn clause computability. *BIT*, 16(2):215–226, 1977.
16. Maarten H. van Emden and Robert A. Kowalski. The semantics of predicate logic as a programming language. *J. ACM*, 23(4):733–742, 1976.