Writing Correct Prolog Programs

David S. Warren(✉)

Stony Brook University, Stony Brook, USA
warren@cs.stonybrook.edu

Abstract. This article describes a somewhat new way of thinking about Prolog programming. It was motivated by a video and presentation by Leslie Lamport [6] in which he argued for a simple model of computation in which, to develop a program, one uses conventional mathematical language with the necessary invariants being front and center. He used as a motivating example the problem of finding the greatest common divisor (GCD) of two positive integers. I felt his model of computation was too simple to be useful for complex programs, but I liked his essential idea. I thought I'd like to apply it to the computational model(s) of logic programming, in particular to Prolog. It led to a somewhat different way of thinking about how to develop Prolog programs that takes advantage of both bottom-up and top-down thinking. This article explores this program development strategy using the GCD problem as a motivating example.

1 Inductive Definitions

Prolog is basically a language of inductive definitions. (See M. Denecker's work including [3,8].) We all learn about inductive definitions early in our mathematics education. The first definition I remember learning was of the factorial function. Factorial is described informally as a function of natural numbers where $n! = n*(n-1)*(n-2)*...*1$. Even though this seemed pretty clear to me (at least for positive integers), I was told that the "..." in this purported definition isn't precise enough. A better way to define factorial is needed, and an inductive definition does the job:

```
n! = 1 if n=0
n! = n*(n-1)! if n>0
```

The first clause specifies the value of factorial of 0 directly; the second clause specifies the values for all natural numbers greater than 0. It's clear how one can use this definition to find the value of $n!$ for any n. For example, say we want the value of 4!. We know $0! = 1$ from the first clause; from the second clause, we know $1! = 1*0!$, and since we've established that $0! = 1$, then $1! = 1*1 = 1$; again from the second clause $2! = 2*1! = 2*1 = 2$; again $3! = 3*2! = 3*2 = 6$; and finally $4! = 4*3! = 4*6 = 24$. In the same way we can find the value of factorial for *any* natural number by starting with 0 and computing the values of factorial

D. S. Warren et al. (Eds.): Prolog: The Next 50 Years, LNAI 13900, pp. 62–70, 2023.
https://doi.org/10.1007/978-3-031-35254-6_5

for all numbers up to and including the one of interest. This is ensured by the fact that all natural numbers can be reached by starting with 0 and adding 1 some (finite) number of times.

Here we have defined a function inductively. We can also (or more generally) defines sets inductively. To define a set inductively, one first specifies a universe of elements. Then one explicitly gives some subset of them as members of the desired set, and provides a set of rules. Each rule says that if certain elements of the universe are in the desired set, then some other element(s) must be in the set. This defines a subset of the universe: the smallest set that contains the explicit elements and is closed under the rules.

As another, perhaps slightly more interesting, example of an inductive definition, we consider GCD, the Greatest Common Divisor relation. The GCD of two non-zero natural numbers is the largest natural number that evenly divides them both. E.g., the GCD of 18 and 24 is 6. We'll want to define $gcd(n, m, d)$ to mean that d is the GCD of the non-zero natural numbers n and m. An inductive definition of this set is:

```
gcd(n,n,n) for n > 0
gcd(n+m,m,d) if gcd(n,m,d)
gcd(n,n+m,d) if gcd(n,m,d)
```

The first (base) clause says that the GCD of a number and itself is that number. The second clause, a rule, says that if d is the GCD of n and m, then it is also the GCD of $n + m$ and m. And the third is similar. We'll leave it to the reader to compute a few of these triples. For example, starting from the single basic element $gcd(1, 1, 1)$ we see that it generates pairs with GCD of 1, i.e., pairs that are relatively prime.

There are two nice properties of inductive definitions that use well-defined and computable conditions:

1. They come with a mechanism to compute their values, as we have seen, by starting with the base clauses, which give the first (unconditional) set members; and then continuing to apply the other clauses until we get the answer we want (or maybe all the answers.) This iterative process can serve as a computational model for inductive definitions.
2. They provide a ready-made structure for proving properties that hold of all members of the inductively defined sets: we just need to show that the desired property holds for the unconditional members defined by the base clauses, and that if the property holds for the set elements used in the conditions of an inductive clause and the condition itself holds, then the property holds for the newly specified member. Intuitively our reasoning for why this is true can follow exactly the process we used to add members to the set: we see that each *initial* member and each *added* member must have the desired property, so that *all* members of the set must have it. The Induction Principle guarantees that the property holds for *every* member of the defined set.

As an example of a proof, say we want to prove that if $gcd(n, m, d)$ is in the set defined inductively above, then d is indeed the greatest common divisor of n

and m. For the base clause, clearly the greatest common divisor of two identical numbers is that number itself. For the inductive clauses, if d divides n and m, then it clearly divides $n+m$. And if there were a greater divisor of $n+m$ and m, that greater divisor would have to divide n, contradicting the assumption that d is the GCD of n and m. And similarly, for the other inductive clause. So. we have proved the desired property. In fact our (perhaps implicit) recognition of this property was what led us to write this definition in the first place.

The computational model and the proof method are fundamentally intertwined. When we wrote the inductive definition we had in mind the property we wanted the set to have, and ensured that each rule preserved that property. I.e., we had the proof of correctness directly in mind when we wrote the definition.

2 From Inductive Definition to Prolog Program

We now have an inductive definition of the GCD relation, which has been proved to be correct. But we want a Prolog program for finding GCD. How do we turn this inductive definition into a correctly running Prolog program?

We take the inductive definition that was written in English using conventional mathematical notation:

```
gcd(n,n,n) for n > 0
gcd(n+m,m,d)) if gcd(n,m,d)
gcd(n,n+m,d)) if gcd(n,m,d)
```

and we directly convert it to a Prolog program. We note that there have been many extensions to the Prolog language, so it makes a difference which dialect of Prolog we are working with. For our purposes here, we will assume a relatively primitive Prolog, essentially ISO Prolog. But there are Prolog systems that support functional notation and general numeric constraints. In these Prologs different transformations, perhaps including none, would be necessary.

ISO Prolog doesn't support functional notation, so we need to introduce new variables (identifiers starting with upper-case letters) for the sums in the atomic formulas. And we can convert the sums to use Prolog's general arithmetic construct, `is/2`:

```
gcd(N,N,N) :- N > 0.
gcd(NpM,M,D)) :- gcd(N,M,D), NpM is N + M.
gcd(N,NpM,D)) :- gcd(N,M,D), NpM is N + M.
```

This form now is a Prolog program in that it satisfies Prolog syntax. However, to determine if it will correctly execute to solve a problem, we have to consider the queries we will ask. In our case, we want to provide two integers and have Prolog determine the GCD of those two integers. This mode of the queries we will ask is denoted by `gcd(+,+,-)`, where `+` indicates a value is given and `-` indicates a value is to be returned by the computation. Here we intend to give the first two arguments and expect the have the third one returned. And we also need to

know the mode of each subquery in order to determine if the definition can be executed by the Prolog processor.

Correct Prolog evaluation depends on subquery modes in two ways: 1) subqueries must have only finitely many answers, and 2) Prolog's predefined predicates (like `is/2`) often work only for particular modes and so the modes for calls to those predicates must be acceptable.

We must check that the modes of our Prolog program for GCD are correct. The first requirement for correct evaluation of a query (or subgoal) is that it must have only finitely many answers. Since Prolog uses backward-chaining, it poses a number of subgoals during its computation starting from an initial goal. We must be sure that every one of those subgoals has only finitely many answers. Otherwise, Prolog would go into an infinite loop trying to generate all infinitely many answers to such a subgoal. For example, the goal `gcd(27,63,D)` has only finitely many instances in the set defined by the `gcd` program; actually only one instance, with `D=9`. But the goal `gcd(N,63,9)` has infinitely many instances in the defined set; including all those with `N` being a multiple of `9`. The lesson here is that we need to understand the *modes* of all subgoals generated by a Prolog computation. The *mode* of a subgoal describes where variables appear in it, and this affects whether it matches infinitely many set members or not.

We note that since Prolog calls goals in the body of a rule in a left-to-right order, the mode of a subgoal is determined by the success of goals to its left in the rule body, as well as by the mode of this initial call. We assume that the initial goal that we pose to this program will have the first two fields as numbers and the third field as a variable. I.e., we'll be asking to find the GCD of two positive integers. This mode is expressed as `gcd(+,+,-)`.

We next explore the modes of the subgoals in our Prolog program above. Under this mode assumption, since any call to `gcd` will have its first two arguments bound to values, the variable `N` in the subgoal `N > 0` in the first clause will have a value as required by Prolog so this condition can be checked.

In the second clause `NpM` and `M` will be integers (by the mode assumption), so in the first subgoal, `NpM is N + M`, `NpM` and `M` will have values, but `N` will be a variable. That means that the call to `gcd(N,M,D)` will have mode `(-,+,-)`, and this subgoal will have infinitely many answers. So, this Prolog program will not execute correctly. (The third rule suffers from a similar problem.) We need `N` to get a value before the call to `gcd(N,M,D)`. `N` appears in the second subgoal of that condition, so let's try evaluating that subgoal first. To this end we move the `is/2` goals earlier and get a new program:

```
gcd(N,N,N) :- N > 0.
gcd(NpM,M,D)) :- NpM is N + M, gcd(N,M,D).
gcd(N,NpM,D)) :- NpM is N + M, gcd(N,M,D).
```

Now in the call to `is/2`, `NpM` and `M` will have values (because of the mode assumption for calls to `gcd/3`). Prolog, however, requires `is` to have the mode of `is(?,+)`. (The "?" indicates either "+" or "-" is allowed.) Since `N` will not have a value, the second argument to is/2 will not have a value and this rule will

result in an error when evaluated with the expected values. But we can change the `is/2` to compute `N` from `MpN` and `M` by writing `N is MpN - M`. This imposes the equivalent constraint among `N`, `M` and `MpN` and is correctly modded for Prolog to evaluate it. Similarly fixing the third clause gives us a program:

```
gcd(N,N,N) :- N > 0.
gcd(NpM,M,D)) :- N is NpM - M, gcd(N,M,D).
gcd(N,NpM,D)) :- M is NpM - N, gcd(N,M,D).
```

In the second clause now, the first body subgoal causes `N` to get a value, so the second subgoal is called with `N` and `M` with values, and thus in the same mode as the original mode, that is `gcd(+,+,-)`. Similarly for the third clause. Thus all calls to `gcd/3` (and `is/2`) will be correctly moded.

However, there is still an issue with this program: Prolog computes with integers, not natural numbers, and so the subtraction operations might generate negative integers. But we want only positive integers. So we must add this constraint explicitly as follows:

```
gcd(N,N,N) :- N > 0.
gcd(NpM,M,D)) :- N is NpM - M, N > 0, gcd(N,M,D).
gcd(N,NpM,D)) :- M is NpM - N, M > 0, gcd(N,M,D).
```

Only when we generate a new integer do we need to check that it is positive. And we must do the check *after* the number variable gets a value to satisfy the mode requirement of `</2`; immediately after it gets that value is best. This is now a good Prolog program for computing the GCD of two integers. You might recognize this as the Euclidean algorithm for GCD. I.e., the Euclidean algorithm is the top-down evaluation (i.e., Prolog evaluation) of this inductive definition. Actually, we can make this algorithm slightly more efficient, and maybe make it look a bit more like Euclid's algorithm by noting in the second clause (and analogously in the third) that `N` will be greater than `0` only if `NpM` is greater than `M`, so we can make that check before taking the difference, getting:

```
gcd(N,N,N) :- N > 0.
gcd(NpM,M,D)) :- NpM > M, N is NpM - M, gcd(N,M,D).
gcd(N,NpM,D)) :- NpM > N, M is NpM - N, gcd(N,M,D).
```

(Renaming the variables in the third clause might make it look even more familiar.) Now we can see that only one of the three clauses can ever satisfy its comparison condition for a (correctly moded) subgoal, and so the Prolog computation is deterministic.

See Kowalski et al. [5] for another development of Euclid's algorithm, there in a more English-like dialect of Prolog.

Let's recap how we approached Prolog programming. We followed a sequence of steps, which we will describe in some generality here. They are a generalization of the steps we just used in our development of the `gcd` Prolog program.

1. Use inductive clauses to define the relation, a set of tuples, that characterizes the solution to the problem of interest. Use names for the relations to organize

and name sets of tuples. Above we used the name `gcd` as the predicate symbol to remind us that it is a set of triples that define the Greatest Common Divisor function. Use whatever mathematical notation is convenient. Similarly, define and use whatever sets of tuples are useful as subsidiary definitions.
2. Convert this mathematical definition into Prolog clauses, using the necessary Prolog built-ins.
3. Consider the mode of each subgoal that will be invoked during top-down evaluation of the clauses. Ensure that the subset of each relation required to answer each moded subgoal is finite. Ensure that all built-ins are invoked in modes that they support. Ensure that defined subgoals are all invoked in desired modes, by ordering the subgoals of rule bodies so their left-to-right evaluation is well-moded and efficient.

Notice that in developing this program, we did not consciously think about recursive programming. We thought about an inductive definition. Recursion is the procedural mechanism used to evaluate inductive definitions top-down. The procedural mechanism to evaluate inductive definitions bottom-up is iteration.

To understand the correctness of an inductive definition we can think iteratively; just iterate the application of the inductive rules, starting from empty relations. Intuitively, this is easier to understand than recursive programming. The recursion showed up in the final program because of the form of the rules of the inductive definition. We know the recursive evaluation will give the correct answer because top-down evaluation computes the same answer (when it terminates) as the bottom-up iterative evaluation.

Many others have noted the importance of bottom-up thinking. The deductive database community (see, e.g., [7]) looks at programs with only constant variable values, so-called Datalog programs, exclusively bottom up. And teachers of Prolog teach bottom-up evaluation, and sometimes provide bottom-up evaluators for students to use to understand the programs they write, [4].

3 The Claim

Let's look at what needs to be done to write a correct program. First we have to determine what it means for a correct program to be correct. For that we need to have an idea in our heads of what our program should do, i.e., a property the program should have, i.e., for a logic program a property that must hold of every tuple in the defined relation. And we must ensure that it contains every tuple it should. Then we have to write a program that defines a relation that has that property.

We could require that this whole process be done formally, i.e., do a formal verification of our program. In that case, we would specify the correctness property, a.k.a. the program specification, in some formal language. Then we would generate a formal proof that the program we created satisfied that specification. This turns out to be rather complicated and messy, and for large practical programs essentially undoable. Almost no large programs in practice are ever proved correct in this way. (See [2] for a discussion of relevant issues.)

In lieu of formal verification we might use informal methods that won't guarantee exact program correctness but might provide some confidence that our programs do what we want them to do. We argue that the informal program development strategy that we have described above does just that.

When we develop an inductive definition of a relation that satisfies our intuitive correctness criteria (what we want the program to do), we are thinking of bottom-up generation of a satisfying relation. And the bottom-up generation must generate correct tuples at every step. And seeing that rules always generate correct tuples is exactly what a proof of correctness requires. Indeed, the name given to the predicate is naturally a shorthand for its correctness property. We named our predicate `gcd` because of the Greatest Common Divisor property that we wanted its tuples to have. So, when generating an inductive definition of a relation, one has directly in mind the property all its tuples must have, and so writes rules that guarantee that property. This is exactly the thinking necessary to formulate a proof of correctness. In this way the thinking required to formulate the program is exactly the thinking require to formulate a proof of its correctness. Even if the proof is not formally carried out, the intuitive ideas of how it would be created have already been thought through.

Note, however, that such an inductive proof is not a formal proof of correctness for a Prolog program. That would require formal consideration of modes, occur-check, termination, and other important details of actual Prolog evaluation. Discussions of formal proofs of total correctness of Prolog programs in the literature tend to focus on these issues, e.g., [1].

Of course, most Prolog programs are more complicated than a single inductive definition. Most require multiple subsidiary relations to be defined and then used in more complex definitions. But each such subsidiary relation is also inductively defined and a similar methodology can be used for them. For procedural programs with `while` and `for` constructs, one needs to generate an invariant for each loop; the corresponding logic program requires a relation, with its own name and inductive definition, for each loop, thus laying bare the correctness criteria that may be hidden for iterative programs with loops.

4 Caveats

What is proposed here is an idealized methodology for developing Prolog programs. Of course, it won't always work this way. Prolog programmers learn (and are encouraged) to think about top-down execution and recursion in a procedural way. Indeed, to develop definitions that evaluate efficiently top-down, it is often necessary to think in this way. So almost always an experienced Prolog programmer will develop a program without ever thinking about how it would evaluate bottom up. My suggestion is that programmers should initially be taught this bottom-up-first methodology, and then as they advance and develop their top-down intuitions, they should always go back and look at the bottom-up meanings of their programs. As a practicing top-down Prolog programmer, I've found it often enlightening to think of my programs in a bottom-up way. Sometimes efficient top-down programs are infinite, or ridiculously explosive, as bottom-up

programs. But experience can make them intuitively understandable, and thinking that way provides insight. It deepens an understanding of what the program defines and can sometimes uncover problems with it. It is also worth noting that bottom-up evaluation is independent of problem decomposition, which is a good development strategy, independent of any evaluation strategy.

Bottom-up evaluation is generally easier to understand not only because it is based on iteration instead of recursion as is top-down. Every state reachable in a bottom-up construction satisfies the desired correctness property. But in top-down, many reachable states may not satisfy the correctness property; they may be states only on a failing derivation path. This means that for top-down one must distinguish between states that satisfy the desired property and those encountered along a failing path towards a hoped-for justification. It's more to keep track of in one's head. Perhaps another way to say it is that bottom-up evaluation is intuitively simpler in part because it needs no concept of failure.

5 Conclusion

I would claim that mathematical induction provides the formal foundation of all algorithmic computation, i.e., computation intended to terminate[1]. Prolog asks programmers to give inductive definitions directly. The form of definition is particularly simple, being straightforward rules for adding members to sets. Since the programmer creates definitions thinking directly in terms of the mathematical foundations of computation, there is less of a distance between programming and proving. This makes for programs more likely to do what the programmer intends.

References

1. Apt, K.R.: Program verification and Prolog. In: Börger, E. (ed.) Specification and Validation Methods, pp. 55–95. Oxford University Press, Oxford (1993)
2. DeMillo, R., Lipton, R., Perlis, A.: Social processes and proofs of theorems and programs. In: Tymoczko, T. (ed.) New Directions in the Philosophy of Mathematics: An Anghology, pp. 237–277. Birkhauser Boston Inc., Boston (1986)
3. Denecker, M., Warren, D.S.: The logic of logic programming. CoRR, cs.LO/2304.13430, arXiv:2304.13430 (2023)
4. Hermenegildo, M.V., Morales, J.F.: Some thoughts on teaching (and preaching) Prolog. In: Warren, D.S., Dahl, V., Eiter, T., Hermenegildo, M., Kowalski, R., Rossi, F. (eds.) Prolog - The Next 50 Years. LNCS, vol. 13900, pp. 107–123. Springer, Cham (2023)
5. Kowalski, R., Quintero, J.D., Sartor, G., Calejo, M.: Logical English for law and education. In: Warren, D.S., Dahl, V., Eiter, T., Hermenegildo, M., Kowalski, R., Rossi, F. (eds.) Prolog - The Next 50 Years. LNCS, vol. 13900, pp. 287–299. Springer, Cham (2023)

[1] Clearly quantum computation has a different foundation. And infinite computations have their foundations in co-induction.

6. Lamport, L.: If you're not writing a program, don't use a programming language. In: LPOP 2020, November 2020. https://www.youtube.com/watch?v=wQiWwQcMKuw
7. Maier, D., Tekle, K.T., Kifer, M., Warren, D.S.: Declarative logic programming. Chap. Datalog: Concepts, History, and Outlook, pp. 3–100. Association for Computing Machinery and Morgan & Claypool, New York, NY, USA (2018). https://doi.org/10.1145/3191315.3191317
8. Vennekens, J., Denecker, M., Bruynooghe, M.: FO(ID) as an extension of dl with rules. Ann. Math. Artif. Intell. **58**(1–2), 85–115 (2010)