Formal Methods: A First Introduction using Prolog to specify Programming Language Semantics

Lutz Hamel
Department of Computer Science and Statistics
University of Rhode Island
Kingston, Rhode Island, USA
hamel@cs.uri.edu

Abstract

An important fundamental idea in formal methods is that programs are mathematical objects one can reason about. Here we introduce students and developers to these ideas in the context of formal programming language semantics. We use first-order Horn clause logic as implemented by Prolog both as a specification and a proof scripting language. A module we have written facilitates using Prolog as a proof assistant and insures that Prolog implements a sound logic. In order to illustrate our approach we specify the semantics of a small functional language and demonstrate various proof approaches and styles.

1 Introduction

An important fundamental idea in formal methods is that programs are mathematical objects one can reason about [1]. This fundamental idea appears in many areas of software development including algorithm correctness, programming language semantics, compiler correctness, system validation, and system security. For instance, in security sensitive systems one could look at a program as a mathematical object and then formally reason about the safety of that program with respect to some metric. Given the importance of this topic every software developer should be exposed to at least the fundamental concepts and ideas of formal methods [2, 3]. In our curriculum we expose students to ideas in formal methods in the context of formal programming language semantics. Here, programs are structures with corresponding models and the idea is to be able to formally reason about the behavior of programs. The advantage of using programming language semantics as a tool for teaching formal methods is that students have an intuition of what the behavior of a program is and can bring that intuition to the construction of proofs.

After experimenting with many different formalisms including denotational semantics, algebraic semantics, and structural operational semantics we settled on using first-order logic as the formalism for specifying programming language semantics and the corresponding proofs in the context of operational semantic specifications. There are a number of advantages to using first-order logic:

1. It is a formalism most students (and developers) are already familiar with and therefore can concentrate on semantic problems rather than notational issues.
2. It can serve both as a specification language and as a language for constructing proofs.
3. It (or at least the Horn clause subset) is machine executable giving rise to executable specifications and the notion of automatic proof assistants.

We consider the last point extremely important in that students and software developers need to be exposed to automatic theorem proving ideas in the context of formal methods. There exist many first- and higher-order proof assistant systems [4, 5]. However, most of them have difficult notations and concepts of proof construction making them inaccessible for a one or two semester course in formal methods. It turns out that Prolog [6] together with a proof-module that we have developed is more than adequate for an introduction to formal specification of programming language semantics and the construction of the corresponding proofs. Here we describe the proof-module we have developed for Prolog and then we briefly step through an exercise defining the semantics of a small functional programming language together with corresponding proofs.

Using Prolog for the specification of programming language semantics is not new, *e.g.*, [7]. In particular, the work by Christiansen [8] and Mosses [9] stands out because it shares our goal of using Prolog to teach programming language semantics and uses a style of semantic specification similar to the natural semantics style we use in our approach [10]. However, none of the above works takes advantage of Prolog as a theorem prover. The work by Gupta and Pontelli [11] shares our approach by integrating language specification and the corresponding proofs all under the umbrella of logic programming. However, their approach is based on constraint logic programming as opposed to first-order

Horn clause logic [12]. Furthermore, their view of a proof is a single query showing that a particular property holds in their specification. This is very different from our view of a proof as a program over the meta-language of Prolog including queries, assertions, and retractions. It was paramount for us to stay in the confines of first-order Horn clause logic in order to satisfy our teaching goal. As far as we are aware using Prolog as a proof assistant in the context formal specifications is novel.

The remainder of this paper is structured as follows. Section 2 discusses Prolog as a theorem prover or more precisely as an automatic proof assistant. In Section 3 we discuss our approach to the semantic specification of programming languages using Prolog. Section 4 discusses proofs. As mentioned above, we view proofs as programs over the meta language of Prolog and here we showcase a number of different proof techniques applicable to semantic specifications. Finally, in Section 5 we present conclusions and further work.

2 Prolog as a Theorem Prover

2.1 The Logic

The first-order Horn clause logic Prolog implements is perhaps one of the simplest machine executable, Turing complete logics. This makes Prolog attractive as a specification language since its learning curve is not as steep as other logic implementations. Under the following considerations Prolog implements a sound but incomplete logic [13, 14]:

1. The unification algorithm implements the occurrence-check – Most Prologs omit the required occurrence-check for efficiency reasons. However, some Prolog systems such as SWI-Prolog [6] make the occurrence-check user selectable.
2. The proof search strategy is a depth-first search of the refutation proof tree – This is the standard implementation of the search strategy for Prolog due to efficiency reasons.
3. Only ground terms are negated in rule bodies and proof goals.

Our Prolog proof-module for SWI Prolog insures that the three conditions above are met.

For the last condition above it can be shown that under certain circumstances deduction will flounder when negation of non-ground terms is involved [13, 14]. Our module circumvents this problem by introducing a new negation predicate `neg/1` which checks whether the negated term is ground or not:

```
neg(G) :- ground(G),!,call(not(G)).
neg(_) :- throw('term is not ground').
```

Note that it is necessary to abort deduction if a non-ground term is found since simple failure is interpreted as a negation result. The following is a classic example where deduction flounders under negation [14],

```
on_top(X) :- not(blocked(X)).
blocked(X) :- on(Y,X).
on(a,b).
```

Now given the query of 'do there exist any objects Q on top?' Prolog returns the incorrect answer 'false',

```
?- on_top(Q).
false
?-
```

However, it does produce the correct result given the query,

```
?- on_top(a).
true
?-
```

Now, replacing the first line in the program above with the line which includes our new negation predicate,

```
on_top(X) :- neg(blocked(X)).
```

prevents Prolog from performing unsound deductions and will abort the computation.

```
?- on_top(Q).
ERROR: Unhandled exception: term is not ground
?-
```

And it still does produce the correct result given the query,

```
?- on_top(a).
true
?-
```

Even though the incompleteness of the logic is disconcerting it does not have as much an impact on our proofs as one might think due to the fact that we use Prolog as a *proof assistant* along the lines of Coq [4] and Isabelle [5][1] where proofs are composed of many small steps each verified by Prolog rather than a *fully* automatic theorem prover where the system is tasked with also finding the proof steps. That is, we view proofs as programs over the meta-language of Prolog including queries, assertions, and retractions. We refer to these programs as *proof scores*. It is our experience that it is highly unlikely to encounter problems with incompleteness of the logic in this approach. Even if one did, the problems are easily remedied by either reordering the predicates in a proof step (in the case of an infinite search) or including additional lemmas in the proof to work around incompleteness problems due to the restriction of negation to ground terms only.

2.2 Notation

Our style of specification of programming language semantics was inspired by the natural semantics of Kahn [10]. The overall structure of a semantic rule is as follows,

<context>:: <syntax> -» <value> :- <conditions>

The intended interpretation of these rules is: given a context, a piece of abstract syntax is mapped into a semantic value if the conditions hold. In Prolog the symbol `:-` represents the keyword `if`. The rules can be abbreviated to,

<syntax> -» <value> :- <conditions>

if no context is required by the rule. Our module defines this notation to make specifications and proofs more readable.

[1] Neither Coq nor Isabelle is complete due to their use of higher order logics.

2.3 Universally Quantified Queries

Queries in Prolog allow only for existentially quantified variables. However, when constructing proofs it is often necessary to have queries over universally quantified variables. We can simulate universally quantified variables in queries using the following rule from quantification theory [15]:

$$\frac{\begin{array}{l} q \in U \\ P(q) \end{array}}{\therefore \forall x \in U[P(x)]}$$

If a predicate P is true for an arbitrary object q in some domain U it follows that the predicate is true for all objects in that domain. We can use this to pose universally quantified queries in our semantics such as,

```
?- s:: plus(1,1) -->> 2.
```

where we can interpret `s` as a constant representing some state and the query poses the question whether in some state `s` the operation `plus(1,1)` evaluates to the value 2. If the query is successful then we can use the above quantification rule to conclude that the query holds for all possible states. Since this kind of reasoning is always possible we abuse notation slightly and interpret symbolic constants in queries as universally quantified variables unless it is obvious from context that a particular constant is meant, for example, `s0` for the initial state.

2.4 The `xis/2` Predicate

Prolog implements a machine executable logic. Given this we are interested in using programming language specifications both as executable prototypes as well as for proving properties of the specified language. When we use a specification as a prototype we want to appeal to Prolog's efficiency as a programming language which includes the efficient evaluation of arithmetic expressions. When we want to perform proofs we appeal to the declarative side of Prolog [14]. It turns out that these two notions clash in the evaluation of arithmetic expressions using the `is/2` predicate. The `is` predicate is very efficient for evaluating arithmetic expressions,

```
?- X is 1 + 1.
X = 2.
```

However, when performing proofs it is often necessary to write arithmetic expressions involving universally quantified variables,

```
X is k + 1.
```

and this leads to problems because `is` does not know how to handle these quantities,

```
?- X is k + 1.
ERROR: is/2: Arithmetic: 'k/0' is not a function
```

In order to accommodate proofs involving universally quantified variables our module implements the `xis/2` predicate (eXtended `is`) which behaves just like `is` but allows universally quantified variables,

```
?- X xis k + 1.
X = k+1.
```

It does perform partial evaluation of the expressions where possible,

```
?- X xis 0, Y xis k + 3 * cos(X).
X = 0,
Y = k+3.0.
```

2.5 Additional Predicates

In order to make proofs more readable and easier to follow at runtime our module defines some additional predicates. These predicates do not add new meta-language functionality to Prolog but rather act as wrappers for existing functionality that provide better self-documentation of proofs and a better runtime trace. Among the newly defined predicates are:

`assume/1` – this is the same as `asserta/1`.

`remove/1` – this is the same as `retract/1`.

`show/1` – this is the same as a Prolog query.

Each of these predicates preserves the original functionality but outputs additional information when executed. Here is an example of a very simple (and perhaps silly) proof score:

```
:- consult('preamble.pl').
:- >>> 'assume the commutative property'.
:- >>> 'of integer addition'.
:- assume equiv(A+B,B+A).

:- >>> 'show that expressions X and Y'.
:- >>> 'are related by commutativity'.
:- show
     X xis a + b,
     Y xis b + a,
     equiv(X,Y).
```

Here is the runtime trace of this proof score,

```
%   xis.pl compiled 0.00 sec, 33 clauses
%  preamble.pl compiled 0.00 sec, 45 clauses
>>> assume the commutative property
>>> of integer addition
    Assuming: equiv(_G1202+_G1203,_G1203+_G1202)
>>> show that expressions X and Y
>>> are related by commutativity
    Showing:
    _G1214 xis a+b,
    _G1262 xis b+a,
    equiv(_G1214,_G1262)
% proof-simple.pl compiled 0.03 sec, 1,343 clauses
```

Note that in queries Prolog replaces variable names with internally generated unique names. In the case above, for example, the variable `A` is replaced by `_G1202`. Also, the `consult` predicate at the beginning of the proof score loads our module `preamble.pl`. Also note the "executable" comments.

3 Semantic Specifications

In order to illustrate the use of our semantic rules we will specify the semantics of a small functional language inspired by Winskel's REC language [16]. The abstract syntax for this language is shown in Figure 1 with the concrete syntax shown in brackets.

```
E ::= X
    |  I
    |  mult(E,E)            [E * E]
    |  plus(E,E)            [E + E]
    |  minus(E,E)           [E - E]
    |  if(B,E,E)            [if B then E else E end]
    |  let(X,E,E)           [let X = E in E end]
    |  letrec(F,X,E,E)      [let rec F X => E in E end]
    |  fn(X,E)              [fn X => E]
    |  apply(E,E)           [E E]

B ::= true
    |  false
    |  le(E,E)             [E <= E]
    |  eq(E,E)             [E == E]
    |  not(E)              [not E]

I ::= <any integer digit>
X ::= <any variable name>
F ::= <any function name>
```

Figure 1: The abstract syntax of a small functional language.

As usual, we have to give at least one semantic rule for each syntactic unit in the grammar. The distinguishing feature of the semantics for this language is that it has a declaration environment for functions we call `D` and a binding environment for variables we call `S`. Therefore, a state in our semantics is a pair consisting of a declaration environment and a binding environment, *e.g.* `(D,S)`. We start our discussion by giving the rule for the arithmetic operator `mult`,

```
(D,S):: mult(E1,E2) -->> V :-
       (D,S):: E1 -->> V1,
       (D,S):: E2 -->> V2,
        V xis V1 * V2,!.
```

This rule can be paraphrased as follows:

> In the context of state `(D,S)`, the operator `mult(E1,E2)` with subexpressions `E1` and `E2` evaluates to the value `V` if under state `(D,S)` the subexpressions `E1` and `E2` evaluate to the values `V1` and `V2`, respectively, and the integer multiplication of `V1` and `V2` is the value `V`.

In Prolog commas represent the boolean connective `and`. Also, in Prolog variables start with a capital letter, that means `E1`, `E2`, `S`, *etc.* are all variables or more precisely meta-variables, *i.e.*, variables of the specification language. Also noteworthy is the cut `(!)` at the end of the rule. We can interpret this cut in one of two ways. First, from a procedural point of view each semantic rule constitutes a state transition and once a state transition was made it is not allowed to be reversed. Second, from a declarative point of view the set of semantic rules constitute an inductively defined set of rules. Therefore, once it has been shown that a rule has been successfully applied to a piece of syntax all other branches of the proof tree can be safely pruned because they will not contain another success. This holds even if there are multiple rules for a particular syntactic unit because those rules will be mutually exclusive (*e.g.*, see the if-then-else rules).

The rules for `plus` and `minus` are analogous to the rule for `mult`. Next we look at integer constants and variables. The rule,

```
I -->> I :- is_int(I),!.
```

states that integer constants are treated as integer values regardless of state. The following rules interpret variables in expressions. The first rule gives an interpretation to function variables and the second rule to variables that range over integer values,

```
(D,_):: F -->> [[X,E,S]] :-
        is_var(F),
        lookup(F,D,[[X,E,S]]),!.

(_,S):: X -->> V :-
        is_var(X),
        lookup(X,S,V),!.
```

The first rule looks up the name `F` in the function declaration environment `D` and returns the closure of a function which incorporates the formal parameter, the function body, and the binding environment in which the function was defined. We denote closures with a double bracket notation, `[[ ]]`. The second rule looks up the variable `X` in the binding environment `S` and returns the bound integer value. The predicate `is_var` insures that the variable names conform to the lexical rules. This predicate is not strictly necessary but here we are dealing with abstract syntax and we do not have a parser enforcing lexical rules. The `lookup` predicate is an auxiliary predicate defined as part of our semantics. The underscore in the rules represents an anonymous variable meaning that the corresponding structure is matched but ignored by the rule. Next, the if expression has its usual interpretation,

```
(D,S):: if(B,E,_) -->> V :-
        (D,S):: B -->> true,
        (D,S):: E -->> V,!.

(D,S):: if(B,_,E) -->> V :-
        (D,S):: B -->> false,
        (D,S):: E -->> V,!.
```

Here the first rule states that if the boolean expression evaluates to the value true within the context of state `(D,S)` then the first expression is evaluated. The second rule states that otherwise the second expression is evaluated. Let expressions allow us to bind values to variables,

```
(D,S):: let(X,E1,E2) -->> V :-
                is_var(X),
                (D,S):: E1 -->> V1,
        (D,[(X,V1)|S]):: E2  -->> V,!.
```

Here we first evaluate expression `E1` under the original state `(D,S)`. Once we have the corresponding value `V1` we extend the original binding environment `S` with the binding term `(X,V1)` making use of Prolog's list manipulation abilities and evaluate the expression `E2` under this new extended state. The resulting value `V` is the return value of the overall let expression. A special case of the let expression is the let-rec expression which allows us to define recursive functions,

```
(D,S):: letrec(F,X,E1,E2) -->> V :-
                is_var(F),
                is_var(X),
         ([(F,[[X,E1,S]])|D],S):: E2  -->> V,!.
```

The let-rec expression computes the function closure and associates the closure with the function name `F` in the function declaration environment `D`. The expression `E2` is then evaluated in this extended state.

Our programming language also supports anonymous functions envisioned in the style of ML [17]. In the abstract syntax this is denoted by the operator `fn`. As before, the semantic value of a function definition is the closure of the function,

```
(_,S):: fn(X,E) -->> [[X,E,S]] :- is_var(X),!.
```

Finally, we define function application as follows,

```
(D,S):: apply(E1,E2) -->> V :-
                        (D,S):: E1 -->> [[X,E,Sfn]],
                        (D,S):: E2 -->> V2,
              (D,[(X,V2)|Sfn]):: E -->> V,!.
```

Here we see that in order for function applications to make sense the first expression `E1` has to evaluate to a function closure. We then evaluate the second expression `E2` and its value `V2` is used to create a binding term `(X,V2)` where `X` is the formal parameter of the function. This binding term is used to extend the function binding environment `Sfn` and the body of the function `E` is evaluated under this extended state.

The semantics of boolean expressions can be specified analogously to the arithmetic expression with the big difference of course that we only have two constant values: `true` and `false`. A complete listing of all the semantic specification rules is available from the authors website.

4 Proofs

Everything in Prolog is a proof – in particular, running a logic program in Prolog is a proof. However, here we are interested in Prolog as a proof assistant in order to prove characteristics of our language specifications. Our view of proofs as programs over the meta language of Prolog seems to be novel and we explore this here. We explore three types of proofs:

- Tests - which are proofs over a particular input-output pair of a program.
- Proofs of language properties - these proofs examine features of the language such as program equivalence.
- Program correctness proofs - proofs whether a program conforms to a given requirement or not.

Here we take a look at each of these proof categories.

4.1 Tests

In testing we are interested in the behavior of language features and want to show that a certain feature behaves as expected given some particular input value. In Prolog we accomplish this by setting up a proof that relates an input to a program to its expected outcome. The following is a simple proof for integer multiplication in our functional programming language assuming that the language definition has been loaded,

```
?- show (d,[(x,10)|s]):: mult(x,10) -->> 100.
    Showing: (d,[ (x,10)|s])::mult(x,10)-->>100
true.
```

We can paraphrase this proof as follows,

> Show that for all declaration environments `d` and all binding environments `s` that contain the binding term `(x,10)` the code snippet `mult(x,10)` evaluates to the value 100.

In order to illustrate how these tests can be used to explore features let us take a look at function calls. Here is a more ambitious test proof regarding function calls,

```
:- consult('functional-rec-sem.pl').
:- assume program
       let(inc,
           fn(x,plus(x,1)),
           apply(inc,1)).
:- >>> 'we have for all states (d,s), (d,s):: P -->> 2'.
:- show
        program P,
        (d,s):: P -->> 2.
```

The above program can be rewritten in concrete syntax as follows,

```
let inc = (fn x => x + 1) in inc 1 end
```

The actual test checks whether for all possible states the program evaluates to the value 2. Here is the corresponding runtime trace of the proof score assuming that the proof score is called 'proof-inc.pl',

```
?- consult('proof-inc.pl').
%    xis.pl compiled 0.00 sec, 33 clauses
%   preamble.pl compiled 0.00 sec, 45 clauses
%  functional-rec-sem.pl compiled 0.01 sec, 68 clauses
    Assuming: program let(inc,fn(x,plus(x,1)),apply(inc,1))
>>> we have for all states (d,s), (d,s):: P -->> 2
    Showing: program _G117, (d,s)::_G117-->>2
% proof-inc.pl compiled 0.01 sec, 72 clauses
true.
```

We can also experiment with the higher-order nature of our functional programming language using currying,

```
:- >>> 'Higher order functions: curried plus'.
:- assume program
       let(add,
           fn(x,
              fn(y,plus(x,y))),
           apply(apply(add,1),1)).
:- >>> 'we have for all states (d,s), (d,s):: P -->> 2'.
:- show
        program P,
        (d,s):: P -->> 2.
```

In terms of concrete syntax the above program is written as:

```
let add = (fn x => (fn y => x + y)) in add 1 1 end
```

4.2 Proofs of Language Properties

In order to prove properties of a programming language it is convenient to define the notion of program equivalence,

$$p_1 \sim p_2 \text{ iff } \forall s, \exists v_1, v_2 [s :: p_1 \rightarrow v_1 \wedge s :: p_2 \rightarrow v_2 \wedge v_1 = v_2]$$

That is, two programs p_1 and p_2 are equivalent if and only if under all states s they produce the same semantic value. We can use this to prove that the multiplication operator in our language is commutative. Looking at the semantic rule for multiplication defined above it is clear that commutativity follows directly from the commutativity of integer multiplication but it is still nice to actually prove that this is so,

```
:- >>> 'Assume that we have expressions a and b'.
:- assume (d,s):: a -->> va.
:- assume (d,s):: b -->> vb.

:- >>> 'Integer multiplication is commutative'.
:- assume equiv(A*B,B*A).

:- show
     (d,s):: mult(a,b) -->> V1,
     (d,s):: mult(b,a) -->> V2,
     equiv(V1,V2).
```

Next we prove that our functional language implements by-value parameter passing. We show this by proving that function application is equivalent to an appropriate let-expression,

```
:- >>> 'By-value parameter passing'.

:- assume          (d,s):: a -->> va.
:- assume (d,[(x,va)|s]):: e(x) -->> ve.

:- show
        (d,s):: let(x,a,e(x)) -->> V1,
        (d,s):: apply(fn(x,e(x)),a) -->> V2,
        V1=V2.
```

The proof itself is straightforward with perhaps the exception of the second assumption which states that any expression `e` parameterized over the variable `x` evaluates to the value `ve` under some state whose binding environment `s` contains the variable binding `(x,va)`.

The following is a proof that in our functional language without function application all programs terminate, *i.e.*, always produce a value. The proof is by structural induction over the expressions,

```
:- >>> 'Base cases:'.

:- >>> 'Variables'.
:- >>> 'Assume that states are finite'.
:- assume lookup(x,s,vx).
:- show (d,s):: x -->> vx.
:- remove lookup(x,s,vx).

:- >>> 'Constants'.
:- assume is_int(n).
:- show (d,s):: n -->> n.
:- remove is_int(n).

:- >>> 'anonymous function definitions'.
:- assume is_var(x).
:- show (d,s):: fn(x,e) -->> [[x,e,s]].
:- remove is_var(x).

:- >>> 'Inductive cases'.

:- >>> 'Operators'.
:- >>> 'mult'.
:- assume (d,s):: a -->> va.
:- assume (d,s):: b -->> vb.
:- show (d,s):: mult(a,b) -->> va*vb.
:- remove (d,s):: a -->> va.
:- remove (d,s):: b -->> vb.

:- >>> 'the remaining operators and boolean'.
:- >>> 'expressions can be proved similarly'.

:- >>> 'programming constructs'.
:- >>> 'let-expression'.
:- assume (d,s):: a -->> va.
:- assume (d,[(x,va)|s]):: e(x) -->> ve.
:- show (d,s):: let(x,a,e(x)) -->> ve.
:- remove (d,s):: a -->> va.
:- remove (d,[(x,va)|s]):: e(x) -->> ve.

:- >>> 'similarly for the let-rec expression'.

:- >>> 'if-expression with case analysis'.
:- assume (d,s):: e1 -->> v1.
:- assume (d,s):: e2 -->> v2.

:- assume (d,s):: b -->> true.
:- show (d,s):: if(b,e1,e2) -->> v1.
:- remove (d,s):: b -->> true.

:- assume (d,s):: b -->> false.
:- show (d,s):: if(b,e1,e2) -->> v2.
:- remove (d,s):: b -->> false.
:- remove (d,s):: e1 -->> v1.
:- remove (d,s):: e2 -->> v2.
```

The structural induction argument as encoded by this proof score is pretty straight forward. Perhaps the only surprising aspects are the 'remove' statements which remove assumptions from the Prolog database. They are necessary in order to prevent assumptions from one step of the proof to "bleed" into another step of the proof.

4.3 Program Correctness Proofs

Program correctness proofs are very similar to testing as discussed above with the exception that we want to show that the program behaves as expected for *all* inputs. Here we use techniques described in [18] and [19].

We start with the correctness proof a program that computes the maximum of two values. The proof makes use of the Prolog built-in predicate `max/2` as a model for the computation of our program.

```
:- >>> 'show that program'.
:- >>> ' P = "let(z,if(le(n,m),m,n),z)"'.
:- >>> 'computes the maximum of'.
:- >>> 'the values assigned to m and n'.

:- assume program let(z,if(le(n,m),m,n),z).

:- >>> 'assume values for m and n'.
:- assume (d,s):: m -->> vm.
:- assume (d,s):: n -->> vn.

:- >>> 'case analysis on values vm and vn'.
:- >>> 'case vm = max(vm,vn)'.
:- assume vm xis max(vm,vn).
:- >>> 'this implies that'.
:- assume true xis (vn =< vm).
:- show
      program P,
      (d,s):: P -->> vm.

:- remove vm xis max(vm,vn).
:- remove true xis (vn =< vm).

:- >>> 'case vn = max(vm,vn)'.
:- assume vn xis max(vm,vn).
:- >>> 'this implies that'.
:- assume false xis (vn =< vm).
:- show
      program P,
      (d,s):: P -->> vn.

:- remove vn xis max(vm,vn).
:- remove false xis (vn =< vm).
```

The proof performs a case analysis on the values of `m` and `n` and shows that in each case our program evaluates to the correct value for all possible states `s`.

Our next proof is the correctness proof of the factorial function,

```
let
   rec fact x => if x == 1 then 1 else x * fact(x-1) end
in
   fact(1)
end
```

Here is the proof,

```
:- >>> 'Factorial: show that program P:'.
:- assume program
        letrec(fact,
```

```
              x,
              if(eq(x,1),
                 1,
                 mult(x,
                      apply(fact,
                            minus(x,1)))),
       apply(fact,i)).
:- >>> 'is correct for all inputs i > 0'.

:- >>> 'proof by induction on i'.

:- >>> 'base case: i=1'.
:- assume i -->> 1.
:- show
        program P,
       (d,s):: P -->> 1.

:- >>> 'inductive step: i=n'.
:- assume i -->> n.
:- assume false xis n==1.
:- >>> 'inductive hypothesis:'.
:- assume
   apply(fact,minus(x,1)) -->> factorial(n-1).

:- show
        program P,
        (d,s):: P -->> n*factorial(n-1).
```

The proof is by induction over the input to the `fact` function. As a model for the computation we use the factorial operator defined in the standard recursive way for $k > 0$,

$$\text{factorial}(k) = \begin{cases} 1 & \text{if } k = 1 \\ k * \text{factorial}(k-1) & \text{otherwise} \end{cases}$$

5 Conclusions

Every software developer should be exposed to the fundamental idea in formal methods that programs are mathematical objects one can reason about. We introduce this idea in the context of formal programming language semantics. Here, programs are structures with corresponding models and the idea is to be able to formally reason about the behavior of programs. We have shown that the first-order Horn clause logic as implemented by Prolog is a suitable framework to introduce these ideas. Using the specification of a small functional language we have shown that a variety of proof types and styles can be implemented using Prolog as a proof assistant, from simple implication based proofs to induction based arguments. In our view proofs are programs over the meta-language of Prolog and our custom module assists in writing these proofs. Our module also insures that Prolog deduction is sound and allows the use of universally quantified variables in proofs. The advantages of using Prolog is that it is a straightforward language to learn and the underlying logic is likely a formalism most students and software developers have already encountered.

In the future we interested in developing bisimulation and co-inductive techniques using Prolog which would prove useful when proving compilers and translators correct.

This paper is dedicated to Angel.

References

[1] E. M. Clarke and J. M. Wing, "Formal methods: State of the art and future directions," *ACM Computing Surveys (CSUR)*, vol. 28, no. 4, pp. 626–643, 1996.

[2] S. Skevoulis and V. Makarov, "Integrating formal methods tools into undergraduate computer science curriculum," in *Frontiers in Education Conference, 36th Annual*, pp. 1–6, IEEE, 2006.

[3] A. Zamansky and E. Farchi, "Exploring the role of logic and formal methods in information systems education," in *Software Engineering and Formal Methods*, pp. 68–74, Springer, 2015.

[4] Y. Bertot and P. Castéran, *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. springer, 2004.

[5] L. C. Paulson, *Isabelle: A generic theorem prover*, vol. 828. Springer, 1994.

[6] J. Wielemaker, T. Schrijvers, M. Triska, and T. Lager, "SWI-Prolog," *Theory and Practice of Logic Programming*, vol. 12, no. 1-2, pp. 67–96, 2012.

[7] B. R. Bryant and A. Pan, "Rapid prototyping of programming language semantics using prolog," in *Computer Software and Applications Conference, 1989. COMPSAC 89., Proceedings of the 13th Annual International*, pp. 439–446, IEEE, 1989.

[8] H. Christiansen, "Using prolog as metalanguage for teaching programming language concepts," *Issues in Information Technology, EXIT, Warszawa*, pp. 59–82, 2000.

[9] P. D. Mosses, "Modular structural operational semantics," *The Journal of Logic and Algebraic Programming*, vol. 60, pp. 195–228, 2004.

[10] G. Kahn, "Natural semantics," in *4th Annual Symposium on Theoretical Aspects of Computer Science (STACS 87)*, pp. 22–39, Springer-Verlag, 1987.

[11] G. Gupta and E. Pontelli, "Specification, implementation, and verification of domain specific languages: a logic programming-based approach," in *Computational Logic: Logic Programming and Beyond*, pp. 211–239, Springer, 2002.

[12] T. Swift and D. S. Warren, "Xsb: Extending prolog with tabled logic programming," *Theory and Practice of Logic Programming*, vol. 12, no. 1-2, pp. 157–187, 2012.

[13] J. Lloyd, *Foundations of Logic Programming*. Berlin: Springer-Verlag, 1987.

[14] U. Nilsson and J. Małuszyński, *Logic, programming and Prolog*. Wiley Chichester, 1990.

[15] I. Copi, "Introduction to logic (6th ed)," 1982.

[16] G. Winskel, *The formal semantics of programming languages: an introduction*. MIT press, 1993.

[17] R. Milner, M. Tofte, R. Harper, and D. B. MacQueen, *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[18] R. Bird *et al.*, *Introduction to functional programming using Haskell*, vol. 2. Prentice Hall Europe London, 1998.

[19] J. A. Goguen and G. Malcolm, *Algebraic Semantics of Imperative Programs*. MIT Press, 1996.