15-819K: Logic Programming

Lecture 4

Operational Semantics

Frank Pfenning

September 7, 2006

In this lecture we begin in the quest to formally capture the operational semantics in order to prove properties of logic programs that depend on the way proof search proceeds. This will also allow us to relate the logical and operational meaning of programs to understand deeper properties of logic programs. It will also form the basis to prove the correctness of various form of program analysis, such as type checking or termination checking, to be introduced later in the class.

4.1 Explicating Choices

To span the distance between the logical and the operational semantics we have to explicate a series of choices that are fixed when proof search proceeds. We will proceed in this order:

1. Left-to-right subgoal selection. In terms of inference rules, this means that we first search for a proof of the first premiss, then the second, etc.

2. First-to-last clause selection and backtracking. In terms of inference rules this means when more than one rule is applicable, we begin by trying the one listed first, then the one listed second, etc.

3. Unification. In terms of inference rules this means when we decide how to instantiate the schematic variables in a rule and the unknowns in a goal, we use a particular algorithm to find the most general unifier between the conclusion of the rule and the goal.

4. Cut. This has no reflection at the level of inference rules. We have to specify how we commit to particular choices made so far when we encounter a cut or another control constructs such as a conditional.

5. Other built-in predicates. Prolog has other built-in predicates for arithmetic, input and output, changing the program at run-time, foreign function calls, and more which we will not treat formally.

It is useful not to jump directly to the most accurate and low-level semantics, because we often would like to reason about properties of programs that are independent of such detail. One set of examples we have already seen: we can reason about the logical semantics to establish properties such as that the sum of two even numbers is even. In that case we are only interested in successful computations, and how we searched for them is not important. Another example is represented by cut: if a program does not contain any cuts, the complexity of the semantics that captures it is unwarranted.

4.2 Definitional Intpreters

The general methodology we follow goes back to Reynolds [3], adapted here to logic programming. We can write an interpreter for a language in the language itself (or a very similar language), a so-called *definitional interpreter*, *meta-interpreter*, or *meta-circular interpreter*. This may fail to completely determine the behavior of the language we are studying (the *object language*), because it may depend on the behavior of the language in which we write the definition (the *meta-language*), and the two are the same! We then transform the definitional interpreter, removing some of the advanced features of the language we are defining, so that now the more advanced constructs are explained in terms of simpler ones, removing circular aspects. We can interate the process until we arrive at the desired level of specification.

For Prolog (although not pure first-order logic programming), the simplest meta-interpreter, hardly deserving the name, is

```
solve(A) :- A.
```

To interpret the argument to `solve` as a goal, we simply execute it using the meta-call facility of Prolog.

This does not provide a lot of insight, but it brings up the first issue: how do we represent logic programs and goals in order to execute them in our definitional interpreter? In Prolog, the answer is easy: we think of the comma which separates the subgoal of a clause as a binary function symbol denoting conjunction, and we think of the constant `true` which always succeeds as just a constant. One can think of this as replicating the language of predicates in the language of function symbols, or not distinguishing between the two. The code above, if it were formally justified using higher-order logic, would take the latter approach: logical connectives *are* data and can be treated as such. In the next interpreter we take the first approach: we overload comma to separate subgoals in the meta-language, but we also use it as a function symbol to describe conjunction in the object language. Unlike in the code above, we will not mix the two. The logical constant `true` is similarly overloaded as a predicate constant of the same name.

```
solve(true).
solve((A , B)) :- solve(A), solve(B).
solve(P) :- clause(P, B), solve(B).
```

In the second clause, the head `solve((A , B))` employs infix notation, and could be written equivalently as `solve(',' (A, B))`.[1] The additional pair of parentheses is necessary since `solve(A , B)` would be incorrectly seen as a predicate `solve` with two arguments.

The predicate `clause/2` is a built-in predicate of Prolog.[2] The subgoal `clause(P, B)` will unify `P` with the head of each program clause and `B` with the corresponding body. In other words, if `clause(P, B)` succeeds, then `P :- B.` is an instance of a clause in the current program. Prolog will try to unify `P` and `B` with the clauses of the current program first-to-last, so that the above meta-interpreter will work correctly with respect to the intuitive semantics explained earlier.

There is a small amount of standardization in that a clause P`.` in the program with an empty body is treated as if it were P `:- true.`

This first interpreter does not really explicate anything: the order in which subgoals are solved in the object language is the same as the order in the meta-language, according to the second clause. The order in which clauses are tried is the order in which `clause/2` delivers them. And unification between the goal and the clause head is also inherited by the object

[1]In Prolog, we can quote an infix operator to use it as an ordinary function or predicate symbol.

[2]In Prolog, it is customary to write p/n when refering to a predicate p of arity n, since many predicates have different meanings at different arities.

language from the meta-language through the unification carried out by `clause(P, B)` between its first argument and the clause heads in the program.

4.3 Subgoal Order

According to our outline, the first task is to modify our interpreter so that the order in which subgoals are solved is made explicit. When encountering a goal `(`A `,` B`)` we push B onto a stack and solve A first. When A as has been solved we then pop B off the stack and solve it. We could represent the stack as a list, but we find it somewhat more elegant to represent the goal stack itself as a conjunction of goals because all the elements of goal stack have to be solved for the overall goal to succeed.

The `solve` predicate now takes two arguments, `solve(`A`, `S`)` where A is the goal, and S is a stack of yet unsolved goals. We start with the empty stack, represented by `true`.

```
solve(true, true).
solve(true, (A , S)) :- solve(A, S).
solve((A , B), S) :- solve(A, (B , S)).
solve(P, S) :- clause(P, B), solve(B, S).
```

We explain each clause in turn.

If the goal is solved and the goal stack is empty, we just succeed.

```
solve(true, true).
```

If the goal is solved and the goal stack is non-empty, we pop the most recent subgoal A of the stack and solve it.

```
solve(true, (A , S)) :- solve(A, S).
```

If the goal is a conjunction, we solve the first conjunct, pushing the second one on the goal stack.

```
solve((A , B), S) :- solve(A, (B , S)).
```

When the goal is atomic, we match it against the heads of all clauses in turn, solving the body of the clause as a subgoal.

```
solve(P, S) :- clause(P, B), solve(B, S).
```

We do not explicitly check that P is atomic, because `clause(`P`, `B`)` will fail if it is not.

4.4 Subgoal Order More Abstractly

The interpreter from above works for pure Prolog as intended. Now we would like to prove properties of it, such as its soundness: if it proves `solve(`A`, true)` then it is indeed the case that A is true. In order to do that it is advisable to reconstruct the interpreter above in *logical* form, so we can use induction on the structure of deductions in a rigorous manner.

The first step is to define our first logical connective: conjunction! We also need a propositional constant denoting truth. It is remarkable that for all the development of logic programming so far, not a single logical connective was needed, just atomic propositions, the truth judgment, and deductions as evidence for truth.

When defining logical connectives we follow exactly the same ideas as for defining atomic propositions: we define them via inference rules, specifying what counts as evidence for their truth.

$$\frac{A\ \mathit{true} \quad B\ \mathit{true}}{A \wedge B\ \mathit{true}}\ \wedge I \qquad\qquad \frac{}{\top\ \mathit{true}}\ \top I$$

These rules are called *introduction rules* because they introduce a logical connective in the conclusion.

Next we define a new judgment on propositions. Unlike $A\ \mathit{true}$ this is a binary judgment on two propositions. We write it $A\ /\ S$ and read it as A *under* S. We would like it to capture the logical form of `solve(`A`, `S`)`. The first three rules are straightforward, and revisit the corresponding rules for `solve/2`.

$$\frac{}{\top\ /\ \top} \qquad\qquad \frac{A\ /\ S}{\top\ /\ A \wedge S}$$

$$\frac{A\ /\ B \wedge S}{A \wedge B\ /\ S}$$

The last "rule" is actually a whole family of rules, one for each rule about truth of atoms P.

$$\frac{B_1 \wedge \cdots \wedge B_m\ /\ S}{P\ /\ S} \quad \text{for each rule} \quad \frac{B_1\ \mathit{true} \ldots B_m\ \mathit{true}}{P\ \mathit{true}}$$

We write the premiss as $\top\ /\ S$ if $m = 0$, thinking of $\top$ as the empty conjunction.

It is important that the definitions of truth ($A\ \ jtrue$) and provability under a stack-based search strategy ($A\ /\ S$) do not mutually depend on each other so we can relate them cleanly.

Note that each rule of the new judgment $A\ /\ S$ has either one or zero premisses. In other words, if we do proof search via goal-directed search, the question of subgoal order does not arise. It has explicitly resolved by the introduction of a subgoal stack. We can now think of these rules as just defining a transition relation, reading each rule from the conclusion to the premiss. This transition relation is still non-deterministic, because more than one rule could match an atomic predicate, but we will resolve this as well as we make other aspects of the semantics more explicit.

4.5 Soundness

To show the soundness of the new judgment with respect to truth, we would like to show that if $A\ /\ \top$ then $A\ true$. This, however, is not general enough to prove by induction, since if A is a conjunction, the premiss will have a non-empty stack and the induction hypothesis will not be applicable. Instead we generalize the induction hypothesis. This is usually a very difficult step; in this case, however, it is not very difficult to see what the generalization should be.

Theorem 4.1 *If $A\ /\ S$ then $A\ true$ and $S\ true$.*

Proof: By induction on the structure of the deduction $\mathcal{D}$ of $A\ /\ S$.

Case: $\mathcal{D} = \dfrac{}{\top\ /\ \top}$ where $A = S = \top$.

$A\ true$	By $A = \top$ and rule $\top I$
$S\ true$	By $S = \top$ and rule $\top I$

Case: $\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_2\\ A_1\ /\ S_2\end{array}}{\top\ /\ A_1 \wedge S_2}$ where $A = \top$ and $S = A_1 \wedge S_2$.

$A\ true$	By $A = \top$ and rule $\top I$
$A_1\ true$ and $S_2\ true$	By ind.hyp. on $\mathcal{D}_2$
$A_1 \wedge S_2\ true$	By rule $\wedge I$
$S\ true$	Since $S = A_1 \wedge S_2$

Case: $\mathcal{D} = \dfrac{\overset{\mathcal{D}_1}{A_1 \;/\; A_2 \wedge S}}{A_1 \wedge A_2 \;/\; S}$ where $A = A_1 \wedge A_2$.

A_1 *true* and $A_2 \wedge S$ *true*	By ind.hyp. on $\mathcal{D}_1$
A_2 *true* and S *true*	By inversion (rule $\wedge I$)
$A_1 \wedge A_2$ *true*	By rule $\wedge I$

Case: $\mathcal{D} = \dfrac{\overset{\mathcal{D}'}{B_1 \wedge \cdots \wedge B_m \;/\; S}}{P \;/\; S}$ where $A = P$ and $\dfrac{B_1\ \mathit{true} \ldots B_m\ \mathit{true}}{P\ \mathit{true}}$.

$B_1 \wedge \cdots \wedge B_m$ *true* and S *true*	By ind.hyp. on $\mathcal{D}'$
$B_1\ \mathit{true}, \ldots, B_m\ \mathit{true}$	By $m - 1$ inversion steps if $m > 0$ ($\wedge I$)
P *true*	By given inference rule

If $m = 0$ then the rule for P has no premisses and we can conclude P *true* without any inversion steps.

□

4.6 Completeness

The completeness theorem for the system with a subgoal stack states that if A *true* then $A \;/\; \top$. It is more difficult to see how to generalize this. The following seems to work well.

Theorem 4.2 *If A true and $\top \;/\; S$ then $A \;/\; S$.*

Proof: By induction on the structure of the deduction of A *true*.

Case: $\mathcal{D} = \dfrac{}{\top\ \mathit{true}}\ \top I$ where $A = \top$.

$\top \;/\; S$	Assumption
$A \;/\; S$	Since $A = \top$

Case: $\mathcal{D} = \dfrac{\overset{\mathcal{D}_1}{A_1\ \mathit{true}} \quad \overset{\mathcal{D}_2}{A_2\ \mathit{true}}}{A_1 \wedge A_2\ \mathit{true}}\ \wedge I$ where $A = A_1 \wedge A_2$.

$\top \;/\; S$	Assumption
$A_2 \;/\; S$	By ind.hyp. on $\mathcal{D}_2$
$\top \;/\; A_2 \wedge S$	By rule
$A_1 \;/\; A_2 \wedge S$	By ind.hyp. on $\mathcal{D}_1$
$A_1 \wedge A_2 \;/\; S$	By rule

Case: $\mathcal{D} = \dfrac{\begin{array}{ccc}\mathcal{D}_1 & & \mathcal{D}_m \\ B_1\ \mathit{true} & \ldots & B_m\ \mathit{true}\end{array}}{P\ \mathit{true}}$ where $A = P$.

This is similar to the previous case, except we have to repeat the pattern $m-1$ times. One could formalize this as an auxiliary induction, but we will not bother.

$\top \;/\; S$	Assumption
$B_m \;/\; S$	By ind.hyp. on $\mathcal{D}_m$
$\top \;/\; B_m \wedge S$	By rule
$B_{m-1} \;/\; B_m \wedge S$	By ind.hyp. on $\mathcal{D}_{m-1}$
$B_{m-1} \wedge B_m \;/\; S$	By rule
$\top \;/\; (B_{m-1} \wedge B_m) \wedge S$	By rule
$B_1 \wedge \ldots \wedge B_{m-1} \wedge B_m \;/\; S$	Repeating previous 3 steps

$\square$

This form of completeness theorem is a non-deterministic completeness theorem, since the choice which rule to apply in the case of an atomic goal remains non-deterministic. Furthermore, the instantiation of the schematic variables in the rules is "by magic": we just assume for the purpose of the semantics at this level, that all goals are ground and that the semantics will pick the right instances. We will specify how these choices are to be resolved in the next lecture.

4.7 Historical Notes

The idea of defining one language in another, similar one for the purpose of definition goes back to the early days of functional programming. The idea to transform such definitional interpreters so that advanced features are not needed in the meta-language was formulated by Reynolds [3]. After successive transformation we can arrive at an abstract machine. A very systematic account for the derivations of abstract machines in the functional setting has been given by Danvy and several of his students (see, for example, [1]). Meta-interpreters are also common in logic programming, mostly with the goal to extend capabilities of the language. One of the earliest published account is by Bowen and Kowalski [2].

4.8 Exercises

Exercise 4.1 *Extend the meta-interpreter without goal stacks to a bounded interpreter which fails if no proof of a given depth can be found. In terms of proof trees, the depth is length of the longest path from the final conclusion to an axiom.*

Exercise 4.2 *Extend the meta-interpreter without goal stacks with loop detection, so that if while solving an atomic goal P the identical goal P arises again, that branch of the search will be terminated with failure instead of diverging. You may assume that all goals are ground.*

Exercise 4.3 *Extend the meta-interpreter with goal stacks so that if an atomic goal succeeds once, we do not search for a proof of it again but just succeed. You may assume that all goals are ground and you do not need to preserve the memo table upon backtracking.*

Exercise 4.4 *Extend the meta-interpreter with the goal stack to trace the execution of a program, printing information about the state of search.*

4.9 References

[1] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. A functional correspondence between evluators and abstract machines. In *Proceedings of the 5th International Conference on Principles and Practice of Declarative Programming (PPDP'03)*, pages 8–19, Uppsala, Sweden, August 2003. ACM Press.

[2] Kenneth A. Bowen and Robert A. Kowalski. Amalgamating language and metalanguage in logic programming. In K.L. Clark and S.-A. Tärnlund, editors, *Logic Programming*, pages 153–172. Academic Press, London, 1982.

[3] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM Annual Conference*, pages 717–740, Boston, Massachusetts, August 1972. ACM Press. Reprinted in *Higher-Order and Symbolic Computation*, 11(4), pp.363–397, 1998.