Executing Semantic specifications

Yves Bertot

October 25, 2004

1 Prolog execution of Natural Semantics

Semantics specifications can be executed with the help of a prolog interpreter by translating expressions and instructions into Prolog terms, judgements into Prolog predicates, and inference rules into Prolog clauses.

1.1 Representing terms

Prolog provides notations for lists, which we are going to use directly for environments. However, we need to view pairs as terms constructed with a function symbols. We choose `p` as the function symbol.

The rules we have described in our previous course assume that we can distinguish integers from other values. To make it easier, we will simply assume that integers in expressions are always given as term of the form `int(N)`. In the same spirit, we shall assume that variables are always given as terms of the form `var(X)`.

For the addition of two arithmetical expressions, we will use the binary function symbol `plus(E1,E2)`, for `true` and `false` we will use the same function symbols, and for the comparison `lt(E1,E2)`.

For instructions, we use the binary function symbols `assign`, `sequence`, `while`, and the constant `skip`. Thus,

```
while(lt(var(x), int(10)),
  sequence(assign(x, plus(var(x), int(1))),
          assign(y, plus(var(x),var(y)))))
```

represents the program:

```
while x < 10 do
{ x := x+1; y:=x+y }
```

1.2 evaluating expressions

We construct one Prolog clause for each inference rule. The clause head is a direct translation of the rule conclusion, while the clause body is the conjunction of the translations of the rule premises. Variables that appear only once in an inference rule are replaced with the anonymous variable "_". We use Prolog predefined predicates to represent the test checking that two symbols are different, arithmetic operations, and the comparison of two integer values.

We choose the name `s_eval` to represent the judgements $\rho \vdash e \rightarrow v$ and $\rho \vdash b \rightarrow v$.

```
s_eval([p(X,V)|_], var(X), V).
s_eval([p(Y,_)|R], var(X), V) :-
  X \== Y, s_eval(R,var(X),V).
s_eval(_, int(V), V).
s_eval(R, plus(E1, E2), V) :-
  s_eval(R,E1,V1), s_eval(R,E2,V2), V is V1+V2.
s_eval(R, lt(E1,E2),true) :- s_eval(R,E1,V1),
  s_eval(R,E2,V2), V1 < V2.
s_eval(R, lt(E1,E2),false) :-
  s_eval(R,E1,V1), s_eval(R,E2,V2), V2 =< V1.
```

In Prolog, all variables are represented by a symbol starting with a capital letter. This tranlation does not yield a very efficient execution of expressions, because the sub-expressions of a comparison are computed twice if this expression evaluates to `false`.

We can test our encoding with a query like the following one:

```
s_eval([p(x,1),p(y,2)],
       plus(var(x),plus(var(y),int(1))),
       V).
```

1.3 Execution of instructions

We can now add the `s_update` and `s_exec` predicates for the other two judgements used in the natural semantics for IMP.

```
s_update([p(X,_)|T],X,V,[p(X,V)|T]).
s_update([p(Y,Vy)|T],X,V,[p(Y,Vy)|T1])
  :- X \== Y, s_update(T,X,V,T1).

s_exec(R,skip,R).
s_exec(R,assign(X,E),R1) :-
  s_eval(R,E,V),s_update(R,X,V,R1).
s_exec(R,sequence(I1,I2),R2) :-
  s_exec(R,I1,R1), s_exec(R1, I2,R2).
s_exec(R, while(B,_),R) :-
  s_eval(R,B,false).
s_exec(R, while(B,I),R2) :-
  s_eval(R,B,true), s_exec(R,I,R1),
  s_exec(R1, while(B,I), R2).
```

We can now test our specification by making it execute our sample program in the initial context where `x` and `y` have the value 0. Here is the query.

```
s_exec([p(x,0),p(y,0)],
  while(lt(var(x), int(10)),
    sequence(assign(x, plus(var(x), int(1))),
             assign(y, plus(var(x), var(y))))), R).
```

When executing this query, the Prolog interpreter replies that there is only one answer, for R being the value

```
R = [p(x,10),p(y,55)]
```

All these examples can be run on a linux machine using gnu-prolog. I used Gnu-prolog version 1.1.2.

1.4 Constructing derivations

The prolog engine actually search for the existence of a proof using the inference rules, but it does not construct a derivation that can be inspected by the user. It is possible to make it produce both the final result and the derivation structure, by adding an extra argument to predicates. In the clause head, the extra argument is a term indicating the name of the rule being used and its as arguments the various terms representing the sub-derivations. Here is an example, where we instrumented only the semantics rules for instruction execution.

```
s_exec(R,skip,R,sn1).
s_exec(R,assign(X,E),R1,sn2) :-
  s_eval(R,E,V),s_update(R,X,V,R1).
s_exec(R,sequence(I1,I2),R2,sn3(D1,D2)) :-
  s_exec(R,I1,R1,D1), s_exec(R1, I2,R2,D2).
s_exec(R, while(B,_),R,sn4) :- s_eval(R,B,false).
s_exec(R, while(B,I),R2,sn5(D1,D2)) :-
  s_eval(R,B,true), s_exec(R,I,R1,D1),
  s_exec(R1, while(B,I), R2,D2).
```

Queries must now be adapted to take into account the extra parameter of the execution predicate.

```
s_exec([p(x,0),p(y,0)],
  while(lt(var(x), int(10)),
    sequence(assign(x, plus(var(x), int(1))),
             assign(y, plus(var(x), var(y))))), R,P).
```

The answer to this query is the following one:

```
P = sn5(sn3(sn2,sn2),sn5(sn3(sn2,sn2),sn5(sn3(sn2,sn2),
     sn5(sn3(sn2,sn2),sn5(sn3(sn2,sn2),sn5(sn3(sn2,sn2),
      sn5(sn3(sn2,sn2),sn5(sn3(sn2,sn2),sn5(sn3(sn2,sn2),
        sn5(sn3(sn2,sn2),sn4))))))))))
R = [p(x,10),p(y,55)] ?
```

The variable P contains the squeleton of the derivation that proves the judgement corresponding to the query. However, information about the exact judgement appearing as conclusion of each sub-derivation is missing. Extra arguments to the derivations can be added to make sure this information is also recorded in the derivation structure. However, such a instrumentation is even more costly and we will not spend more time on this.

This approach to executing semantic specifications can also be adapted to structural operational semantics (little step). This would be a good exercise.

1.5 How to use Prolog

One should put all the clauses in a file. For instance, the file `sn.pl` should contain the following text.

```
s_eval([p(X,V)|_], var(X), V).
s_eval([p(Y,_)|R], var(X), V) :-
   X \== Y, s_eval(R,var(X),V).
s_eval(_, int(V), V).
s_eval(R, plus(E1, E2), V) :-
   s_eval(R,E1,V1), s_eval(R,E2,V2), V is V1+V2.
s_eval(R, lt(E1,E2),true) :-
   s_eval(R,E1,V1), s_eval(R,E2,V2), V1 < V2.
s_eval(R, lt(E1,E2),false) :-
   s_eval(R,E1,V1), s_eval(R,E2,V2), V2 =< V1.

s_update([p(X,_)|T],X,V,[p(X,V)|T]).
s_update([p(Y,Vy)|T],X,V,[p(Y,Vy)|T1])
  :- X \== Y, s_update(T,X,V,T1).

s_exec(R,skip,R).
s_exec(R,assign(X,E),R1) :-
   s_eval(R,E,V),s_update(R,X,V,R1).
s_exec(R,sequence(I1,I2),R2) :-
   s_exec(R,I1,R1), s_exec(R1, I2,R2).
s_exec(R, while(B,_),R) :-
   s_eval(R,B,false).
s_exec(R, while(B,I),R2) :-
  s_eval(R,B,true), s_exec(R,I,R1),
  s_exec(R1, while(B,I), R2).
```

One can then run prolog and require that it loads the file in the following manner (the text in italics corresponds to answers from the computer).

```
[bertot@localhost gprolog-1.1.2]$ bin/gprolog
GNU Prolog 1.1.2
By Daniel Diaz
Copyright (C) 1999 Daniel Diaz
```

```
| ?- consult('sn.pl').
compiling sn.pl for byte code...
/home/bertot/gprolog-1.1.2/sn.pl compiled,
   20 lines read - 5401 bytes written, 23 ms

yes
| ?- s_exec([p(x,0),p(y,0)],
  while(lt(var(x), int(10)),
    sequence(assign(x, plus(var(x), int(1))),
             assign(y, plus(var(x), var(y))))), R).
R = [p(x,10),p(y,55)] ?
```

When the system give an answer, there may be several, one can request the next one by typing a semi-column "`;`".

Executing specifications in a Prolog interpreter gives the possibility to construct derivations, but it does not make it possible to have computer support when reasoning about specifications.

2 Executing inside a proof system

To reason about derivations we will use a proof system. The system I propose is the `Coq` system, but any proof system containing inductive types is well suited for this purpose: `Isabelle`, `HOL`, `PVS` are well-suited, but obviously with a different syntax each time.

2.1 Representing expressions

The `Coq` system provides many pre-defined data-structures, but you must state explicitely that you are going to use them. In our case, we will rely on the built-in notions of integers and lists. For this reason, our session needs to start with the following two commands:

```
Require Export ZArith.
Require Export List.
Open Scope Z_scope.
```

In the Coq system, everything expression must have a type and every type must be defined. Parametric types are allowed, so that there exist only one type definition for all kinds of pairs: a pair of an integer and an integer, a pair of a list of integers and an boolean value, for instance. To know the type of a well-formed expression, and to know whether an expression is well-formed we can use a command called `Check`.

```
Check (1,2).
(1,2) : (Z*Z)%type

Check ((1,2)::nil).
(1,2)::nil : list(Z*Z)
```

We asked two questions and received two answers. The first answer, is that `(1,2)` has type `(Z*Z)`, but this answer is would be ambiguous, because `*` can be used to denote both the multiplication of two integers and the cartesian product of two types.

In the second answer there is no ambiguity, because `list` expects a type and (Z*Z), in this context can only be a type, actually a cartesian product type. The `list` type is a parametric type, in fact it is a function that takes a type and returns a new type. We can use the `Check` function to verify this:

```
Check list.
list : Set -> Set
```

We will now define a new type for the arithmetic expression of our language. There are three kind of expressions. An expression can be a variable, an integer, or an addition, but an addition can have an expression as a sub-term. The type of expressions is the type of terms obtained in this manner. This kind of definition is known as an inductive type, because it is defined as the smallest set that is stable for a collection of operations. The way to define an inductive type in `Coq` is to use the following kind of command:

```
Inductive aexpr : Set :=
  avar : Z -> aexpr
| aint : Z -> aexpr
| aplus : aexpr -> aexpr -> aexpr.
```

This definition actually defines three functions that return values in `aexpr` at the same time as it defines the type `aexpr`. We use `avar` to represent the fact that variables are taken from a denumerable set. The function `aint` is used to express that an integer value of type `Z` can also be viewed as an expression. The function `aplus` is a binary function that takes two arithmetic expressions and returns an arithmetic expression. The type expression `aexpr -> aexpr -> aexpr` actually is the type `aexpr -> (aexpr -> aexpr)`, in other words the type of a function that takes an arithmetic expression as input and returns and a function taking an arithmetic expression as input and returning an arithmetic expression.

We define a second type to describe boolean expressions.

```
Inductive bexpr : Set :=
  blt : aexpr -> aexpr -> bexpr.
```

We define a third type to describe instructions.

```
Inductive instr : Set :=
  skip : instr
| assign : Z -> aexpr -> instr
| sequence : instr -> instr -> instr
| while : bexpr -> instr -> instr.
```

All these definitions introduce a collection of functions, `avar`, `aint`, `aplus`, `blt`, `skip`, `assign`, `sequence`, and `while`, with which we can construct terms of type `instr` to represent programs. The convention in `Coq` is to represent the application of functions by simple juxtaposition, adding parentheses around the function and its arguments when necessary to disambiguate. For instance, if we take the convention that the variable `x` corresponds to the index 1 and the variable `y` corresponds to the index `y`, we can represent the program

```
while x < 10 do { x := x+1; y := x+y }
```

using the following expression:

```
while (blt (avar 1)(aint 10))
 (sequence (assign 1 (aplus (avar 1)(aint 1)))
     (assign 2 (aplus (avar 1)(avar 2)))).
```

2.2 Semantic specification for evaluation

We are going to write logical formulas in `Coq`. We need to learn a bit more about writing logical formulas. here is a simple introduction.

- The expression `A -> B` can be read as "`A` implies `B`",
- The expression `forall a, B` is the logical formula that states that the expression `B` is always true, forall possible value of `a`, which may occur in `B`.
- The expression `A \/ B` can be read as "`A` or `B`".
- The expression `A /\ B` can be read as "`A` and `B`".
- The expression `V1 = V2` is the proposition stating that `V1` and `V2` are equal.
- The expression `V1 <> V2` is the proposition stating that `V1` and `V2` are not equal.
- The expression `V1 <= V2` can be read as "the integer value `V1` is smaller than or equal to `V2`" and a similar notation exists for strict comparison.

It is important to understand that the same notation with an arrow is used both to represent the type of functions and the implication between two propositions. In fact, every logical proposition is understood as a type and the proofs of propositions actually are elements of a type. This confusion is possible because well-typed expressions do represent reasoning steps correctly. For instance, if we have an element of the type `A` and a function of type `A -> B`, then we can construct an element of type `B` by applying this function to that element. Now, if we rephrase it in logical terms, this gives : if we have a proof of `A` and a proof that `A` implies `B`, then we can deduce a proof of `B`. This makes sense. The `Coq` system is based on this analogy, known in the proof theory community as the *Curry-Howard isomorphism*.

Because we have implication and universal quantification, we can represent the inference rules of semantic specifications as theorems: the conclusion of an inference rule is the conclusion of an implication, and the premises are are premisses of an implication. One extra point is that the variables occurring in an inference rule are always quantified universally.

When we described the types of functions, we showed that a type with two arrows corresponded to the type of a function with two arguments. Rephrased in logical terms, this means that two nested implications correspond to an implication with two premises (when the nesting occurs on the right). Rephrased again, the two following formulas are always equivalent:

```
(A /\ B) -> C

A -> B -> C
```

Because of this equivalence, we will almost never use conjunctions but describe inference rules with several premises using nested implications.

We use an inductive definition to describe the judgement of evaluation of arithmetic expressions. We define simultaneously the predicate `a_eval` and the four theorems that state when this predicate is true. These theorems are called *constructors*.

```
Inductive a_eval : list(Z*Z) -> aexpr -> Z -> Prop :=
  ae_int : forall r n, a_eval r (aint n) n
| ae_var1 : forall r x v, a_eval ((x,v)::r) (avar x) v
| ae_var2 : forall r x y v v' ,
              x <> y -> a_eval r (avar x) v' ->
              a_eval ((y,v)::r) (avar x) v'
| ae_plus : forall r e1 e2 v1 v2,
              a_eval r e1 v1 -> a_eval r e2 v2 ->
              a_eval r (aplus e1 e2) (v1 + v2).
```

This definition states that `a_eval` is a three place predicate, and it imposes the type of the three arguments. It also introduces three theorems that can be used to prove instances of this predicate. Theorem can be used by applying to arguments of the right type and to proofs of the right proposition. For instance, the following commands show that a few well-formed expressions are proofs of logical statements that use `a_eval`:

```
Check (ae_int ((1,2)::nil) 4).
ae_int ((1,2)::nil) 4 : a_eval ((1,2)::nil) (aint 4) 4

Check (ae_var1 nil 1 2).
ae_var1 nil 1 2 : a_eval ((1,2)::nil) (avar 1) 2

Check (ae_plus ((1,2)::nil) (avar 1) (aint 4) 2 4
        (ae_var1 nil  1 2) (ae_int ((1,2)::nil) 4)).
ae_plus ((1,2)::nil) (aplus (avar 1) (aint 4)) 2 4
        (ae_var1 nil  1 2) (ae_int ((1,2)::nil) 4)
  : a_eval ((1,2)::nil) (aplus (avar 1)(aint 4)) (2+4)
```

2.3 Performing and recording proofs.

The Coq system actually makes it possible to define new values, by attaching a name to the terms we are able to construct. For instance, the experiments above can be re-used to establish an example theorem:

```
Theorem ex1 :
   a_eval ((1,2)::nil)(aplus (avar 1)(aint 4)) 6.
Proof (ae_plus ((1,2)::nil) (avar 1) (aint 4) 2 4
        (ae_var1 nil  1 2) (ae_int ((1,2)::nil) 4)).
```

Please note that the theorem statement is not exactly the same as the type of the value, because the theorems contains a 6 where the type of the value contains an expression (2+4). However, The Coq system recognizes these two expressions as representing the same thing.

Constructing theorems by full application of theorems is a long and tedious task. The Coq system actually provides another execution mode, where people just give the statement they want to prove and then apply special commands, called *tactics* to decompose these facts into simpler facts, until they are eventually trivial to prove. Here is an example:

```
Reset ex1.
Theorem ex1 :
 a_eval ((1,2)::nil)(aplus (avar 1)(aint 4)) 6.
Proof.
 replace 6 with (2+4).
2 subgoals

  ============================
   a_eval ((1, 2) :: nil) (aplus (avar 1) (aint 4)) (2+4)

subgoal 2 is:
 2 + 4 = 6

ex1 <  apply ae_plus.
3 subgoals

  ============================
   a_eval ((1, 2) :: nil) (avar 1) 2

subgoal 2 is:
 a_eval ((1, 2) :: nil) (aint 4) 4
subgoal 3 is:
 2 + 4 = 6

ex1 <  apply ae_var1.
2 subgoals
```

```
  ============================
   a_eval ((1, 2) :: nil) (aint 4) 4

subgoal 2 is:
 2 + 4 = 6

ex1 <  apply ae_int.
1 subgoal

  ============================
   2 + 4 = 6

ex1 < trivial.
Proof completed.

ex1 < Qed.
```

In this proof, we only had to replace `6` with `(2+4)` by hand, and then indicate which theorems should be applied. We could have been even faster by indicating that the theorems from the definition should be used in automatic proofs and then relying on automatic proofs. Here is an example:

```
Reset ex1.
Hint Resolve ae_int ae_var1 ae_var2 ae_plus.
Theorem ex1 :
  a_eval ((1,2)::nil)(aplus(avar 1)(aint 4)) 6.
Proof.
 replace 6 with (2+4).
 auto.
 trivial.
Qed.
```

2.4 Completing the semantic specification

We can now give the definitions for the predicate `b_eval` (to evaluate boolean expressions), `s_update` (to update the environment), and `s_exec` (to execute an instruction). The encoding is straight forward and we add all the new theorems in the database.

```
Inductive b_eval : list(Z*Z) -> bexpr -> bool -> Prop :=
| be_lt1 : forall r e1 e2 v1 v2,
            a_eval r e1 v1 -> a_eval r e2 v2 ->
            v1 < v2 -> b_eval r (blt e1 e2) true
| be_lt2 : forall r e1 e2 v1 v2,
            a_eval r e1 v1 -> a_eval r e2 v2 ->
            v2 <= v1 -> b_eval r (blt e1 e2) false.
```

```
Inductive s_update : list(Z*Z)->Z->Z->list(Z*Z)->Prop :=
| s_up1 :
   forall r x v v’, s_update ((x,v)::r) x v’ ((x,v’)::r)
| s_up2 :
   forall r r’ x y v v’, s_update r x v’ r’ -> x <> y ->
    s_update ((y,v)::r) x v’ ((y,v)::r’).

Inductive s_exec : list(Z*Z)->instr->list(Z*Z)->Prop :=
| SN1 : forall r, s_exec r skip r
| SN2 : forall r r’ x e v,
   a_eval r e v -> s_update r x v r’ ->
   s_exec r (assign x e) r’
| SN3 : forall r r’ r’’ i1 i2,
    s_exec r i1 r’ -> s_exec r’ i2 r’’ ->
    s_exec r (sequence i1 i2) r’’
| SN4 : forall r r’ r’’ b i,
    b_eval r b true -> s_exec r i r’ ->
    s_exec r’ (while b i) r’’ ->
    s_exec r (while b i) r’’
| SN5 : forall r b i,
    b_eval r b false -> s_exec r (while b i) r

Hint Resolve
  be_lt1 be_lt2 s_up1 s_up2 SN1 SN2 SN3 SN4 SN5.
```

We can also describe the structural operational semantics for the same language. Here are the definitions:

```
Inductive sos_step :
   list(Z*Z)->instr->instr->list(Z*Z)->Prop :=
 SOS1 : forall r r’ x e v,
   a_eval r e v -> s_update r x v r’ ->
   sos_step r (assign x e) skip r’
| SOS2 : forall r i2, sos_step r (sequence skip i2) i2 r
| SOS3 : forall r r’ i1 i1’ i2,
          sos_step r i1 i1’ r’ ->
          sos_step r (sequence i1 i2)(sequence i1’ i2) r’
| SOS4 :
     forall r b i, b_eval r b true ->
       sos_step r (while b i) (sequence i (while b i)) r.
| SOS5 :
     forall r b i, b_eval r b false ->
       sos_step r (while b i) skip r.

Inductive sos_star :
   list(Z*Z)->instr->instr->list(Z*Z)->Prop :=
  SOS6 : forall r i, sos_star r i i r
| SOS7 : forall r r' r'' i i' i'',
         sos_step r i i' r' -> sos_star r' i' i'' r'' ->
         sos_star r i i'' r''.
```

3 Proofs on inductive properties

In our study of programming language semantics, we performs proofs by induction on the size of derivations. When using this technique, we actually assume that some judgement, for instance $\rho \vdash i \leadsto \rho'$ holds and is proved by some derivation D and that we want to prove a property $P(\rho, i, \rho')$, and we assume an induction hypothesis stating that the property $P(\rho_0, i_0, \rho_0')$ already holds when the judgement $\rho_0 \vdash i_0 \leadsto \rho_0'$ already holds and is proved by a derivation D_0 whose size is smaller that the size of the derivation D. In fact, we only use this induction hypothesis for the direct sub-derivations of D. When we define a predicate as an inductive predicate, as we did for `a_eval`, `b_eval`, `s_exec`, `sos_step`, and `sos_star`, The `coq` system actually makes it possible to do the same reasoning, by using a tactic called `elim`. This tactic actually observes the possible derivations of a proof for an inductive predicate and expresses the assumptions that can be made in each case. Among the assumptions that can be made, it provides induction hypotheses for all the premises of the rule being used that have the right shape.

3.1 First lemma

Let us start with the proof of our first lemma concerning structural semantics execution.

```
Theorem lemma1 :
   forall r r' r'' i i' i'',
    sos_star r i i' r' -> sos_star r' i' i'' r'' ->
    sos_star r i i'' r''.
Proof.
```

When we perform a proof in the `Coq` system, it displays goals that contain a formula to prove and a local context of assumptions that can be used. When proving formulas that are implications or universal quantifications, it is customary to introduce assumptions or constant in the context. Here for example, we will prove our goal by assuming that `r`, `r'`, etcetera, are fixed constants and we will assume that the first premise holds, so that we only have an implication with one premise. We can direct the system to perform this step with the `intro` command.

```
lemma1 < intros r r' r'' i i' i'' H1.
1 subgoal

  r : list (Z * Z)
  r' : list (Z * Z)
  r'' : list (Z * Z)
  i : instr
  i' : instr
  i'' : instr
  H1 : sos_star r i i' r'
  ============================
   sos_star r' i' i'' r'' -> sos_star r i i'' r''
```

We are now going to say that we want to do a proof by induction on the size of the derivation for `H1`, using the `elim` tactic.

```
lemma1 < elim H1.
2 subgoals

  r : list (Z * Z)
  r' : list (Z * Z)
  r'' : list (Z * Z)
  i : instr
  i' : instr
  i'' : instr
  H1 : sos_star r i i' r'
  ============================
   forall (r0 : list (Z * Z)) (i0 : instr),
   sos_star r0 i0 i'' r'' -> sos_star r0 i0 i'' r''

subgoal 2 is:
 forall (r0 r'0 r''0 : list(Z*Z))(i0 i'0 i''0 : instr),
 sos_step r0 i0 i'0 r'0 ->
 sos_star r'0 i'0 i''0 r''0 ->
 (sos_star r''0 i''0 i'' r'' ->
  sos_star r'0 i'0 i'' r'') ->
 sos_star r''0 i''0 i'' r'' -> sos_star r0 i0 i'' r''
```

The command generated two goals, corresponding to the fact that the judgement can only be proved. In the first goal, both `r` and `r'` have been replaced by a new variable `r0` that is universally quantified, and the same happens for `i` and `i'` with a new variable `i0`. This corresponds to the fact that if $\rho \vdash i \rightsquigarrow^* i', \rho'$ was proved by the rule SOS6, then $\rho = \rho'$ and $i = i'$. This goal is trivial to prove, because it contains an implication where the premise and conclusion are the same proposition. This goal is solved easily and the `Coq` system proposes the second goal.

```
lemma1 < trivial.
1 subgoal

  r : list (Z * Z)
  r' : list (Z * Z)
  r'' : list (Z * Z)
  i : instr
  i' : instr
  i'' : instr
  H1 : sos_star r i i' r'
  ============================
   forall (r0 r'0 r''0 : list(Z*Z))(i0 i'0 i''0 : instr),
   sos_step r0 i0 i'0 r'0 ->
   sos_star r'0 i'0 i''0 r''0 ->
   (sos_star r''0 i''0 i'' r'' ->
    sos_star r'0 i'0 i'' r'') ->
   sos_star r''0 i''0 i'' r'' -> sos_star r0 i0 i'' r''
```

This goal consider the case where the derivation is constructed using the rule SOS7. In this case, the judgement mut be of the form $r_0 \vdash i_0 \rightsquigarrow^* i''_0, r''_0$, there must be a derivation proving $r_0 \vdash i_0 \rightsquigarrow i'_0, r'_0$, a derivation $r'_0 \vdash i'_0 \rightsquigarrow^* i''_0, r''_0$, and for this derivation, there is an induction hypothesis stating that $r''_0 \vdash i''_0 \rightsquigarrow^* i'', r''$ implies $r'_0 \vdash i'_0 \rightsquigarrow^* i'', r''$. Under these assumptions, we must prove that $r''_0 \vdash i''_0 \rightsquigarrow^* i'', r''$ implies $r_0 \vdash i_0 \rightsquigarrow^* i'', r''$. The first step we perform is to assume all the premises and place them in the context:

```
lemma1 < intros r0 r'0 r''0 i0 i'0 i''0 Hp1 Hp2 Hr H.
 subgoal

  r : list (Z * Z)
  r' : list (Z * Z)
  r'' : list (Z * Z)
  i : instr
  i' : instr
  i'' : instr
  H1 : sos_star r i i' r'
  r0 : list (Z * Z)
  r'0 : list (Z * Z)
  r''0 : list (Z * Z)
  i0 : instr
  i'0 : instr
  i''0 : instr
  Hp1 : sos_step r0 i0 i'0 r'0
  Hp2 : sos_star r'0 i'0 i''0 r''0
  Hr : sos_star r''0 i''0 i'' r'' ->
       sos_star r'0 i'0 i'' r''
  H : sos_star r''0 i''0 i'' r''
  ============================
   sos_star r0 i0 i'' r''
```

We obtain `sos_star r'0 i'0 i'' r''` by combining `Hr` and `H`.

```
lemma 1 < assert (Hp3 : sos_star r'0 i'0 i'' r'').
...
```

```
lemma 1 < apply Hr; exact H.
1 subgoal

  r : list (Z * Z)
  r' : list (Z * Z)
  r'' : list (Z * Z)
  i : instr
  i' : instr
  i'' : instr
  H1 : sos_star r i i' r'
  r0 : list (Z * Z)
  r'0 : list (Z * Z)
  r''0 : list (Z * Z)
  i0 : instr
  i'0 : instr
  i''0 : instr
  Hp1 : sos_step r0 i0 i'0 r'0
  Hp2 : sos_star r'0 i'0 i''0 r''0
  Hr : sos_star r''0 i''0 i'' r'' ->
       sos_star r'0 i'0 i'' r''
  H : sos_star r''0 i''0 i'' r''
  Hp3 : sos_star r'0 i'0 i'' r''
  ============================
   sos_star r0 i0 i'' r''
```

We can now apply the rule SOS7, using `Hp1` and `Hp3` as hypotheses. When applying SOS7, we need to specify the two intermediate values, which do not appear in the conclusion:

```
lemma1 < apply SOS7 with r'0 i'0.
...
lemma1 < exact Hp1.
...
lemma1 < exact Hp3.
Proof completed
Qed.
```

Let's discuss again on the behavior of the `elim` tactic. We had a statement

`sos_star r i i' r'`

and the tactic looked at the statement to be proved and made it appear as a statement `P r i i' r'` where `P` is the function defined by:

```
P = fun a b c d => sos_star a b i'' r'' -> sos_star c d
                   i'' r''
```

Then it constructed two goals corresponding to the two theorems of the inductive definition with fresh variables, where we had to prove the property `P` instantiated for the variables appearing in the conclusion of these theorems. Also, we could use an hypothesis `P x y z t` for each premise of the rule that was constructed with `sos_star`. All this behavior is described by a theorem called `sos_star_ind`, which is actually used by the `elim` tactic.

```
Coq < Check sos_star_ind.
sos_star_ind
 : forall P : list(Z*Z)->instr->instr->list(Z*Z)->Prop,
   (forall (r:list(Z*Z)) (i:instr), P r i i r) ->
   (forall (r r' r'':list(Z*Z))(i i' i'':instr),
    sos_step r i i' r' ->
    sos_star r' i' i'' r'' -> P r' i' i'' r'' ->
      P r i i'' r'')->
    forall (l : list (Z*Z)) (i i0:instr) (l0:list(Z*Z)),
    sos_star l i i0 l0 -> P l i i0 l0
```

When performing proofs by induction like here, we always have to be careful about the statement that needs to be proved. For instance, our proof would have been impossible to finish if we had started with `intros r r' r'' i i' i'' H1 H2`. When performing this kind of proof directly on the computer, we often have to perform several tries and correct our errors in proof planning.

3.2 Second lemma

When addressing the second lemma, we need to be careful, because we have an hypothesis of the form

```
 sos_star r (sequence i1 (sequence i2 i3)) skip r'
```

and we want to prove an hypothesis of the form

```
 sos_star r (sequence (sequence i1 i2) i3) skip r'
```

The expression `(sequence i1 (sequence i2 i3))` does not appear simply in the conclusion. To work around this difficulty, we actually prove an auxiliary lemma with a more complex form, but where the expressions that appear in the hypothesis are plain variables.

```
Theorem lemma2_aux : forall r r' i i',
  sos_star r i i' r' ->
  i' = skip ->
  forall i1 i2 i3, i = (sequence i1 (sequence i2 i3))
  sos_star r (sequence (sequence i1 i2) i3) skip r'.
Proof.
intros r r' i i' H; elim H.
2 subgoals

  r : list (Z * Z)
  r' : list (Z * Z)
  i : instr
  i' : instr
  H : sos_star r i i' r'
  ============================
   forall (r0 : list (Z * Z)) (i0 : instr),
   i0 = skip ->
   forall i1 i2 i3 : instr,
   i0 = sequence i1 (sequence i2 i3) ->
   sos_star r0 (sequence (sequence i1 i2) i3) skip r0

...
```

We recognize that if the rule SOS6 was used, then the instructions `i` and `i'` must be the same: in the goal they are replaced by the same variable `i0`. We obtain two equalities that cannot be true at the same time. The way to get rid of these equalities is to write with one equality in the other and to use a tactic called `discriminate`, which can be used everytime there is an equality between two different constructors of an inductive set.

```
lemma2_aux < intros r0 i0 Heq1 i1 i2 i3 Heq2.
2 subgoals

  r : list (Z * Z)
  r' : list (Z * Z)
  i : instr
  i' : instr
  H : sos_star r i i' r'
  r0 : list (Z * Z)
  i0 : instr
  Heq1 : i0 = skip
  i1 : instr
  i2 : instr
  i3 : instr
  Heq2 : i0 = sequence i1 (sequence i2 i3)
  ============================
   sos_star r0 (sequence (sequence i1 i2) i3) skip r0

...
lemma2_aux < rewrite Heq1 in Heq2.
...
Heq2 : skip = sequence i1 (sequence i2 i3)
...

lemma2_aux < discriminate Heq2.
1 subgoal
```

```
  r : list (Z * Z)
  r’ : list (Z * Z)
  i : instr
  i’ : instr
  H : sos_star r i i’ r’
  ===========================
   forall (r0 r’0 r’’ : list(Z * Z))(i0 i’0 i’’ : instr),
   sos_step r0 i0 i’0 r’0 ->
   sos_star r’0 i’0 i’’ r’’ ->
   (i’’ = skip ->
    forall i1 i2 i3 : instr,
    i’0 = sequence i1 (sequence i2 i3) ->
    sos_star r’0 (sequence (sequence i1 i2) i3)
        skip r’’) ->
   i’’ = skip ->
   forall i1 i2 i3 : instr,
   i0 = sequence i1 (sequence i2 i3) ->
   sos_star r0 (sequence (sequence i1 i2) i3) skip r’’
```

After this step, we get to the case corresponding to the rules SOS7. To make it clearer, we put all the implication premises in the context.

```
lemma2_aux <intros r0 r’0 r’’ i0 i’0 i’’ Hstep Hstar Hrec
      Heq1 i1 i2 i3 Heq2.
...
  Hstep : sos_step r0 i0 i’0 r’0
  Hstar : sos_star r’0 i’0 i’’ r’’
  Hrec : i’’ = skip ->
         forall i1 i2 i3 : instr,
         i’0 = sequence i1 (sequence i2 i3) ->
         sos_star r’0 (sequence (sequence i1 i2) i3)
             skip r’’
  Heq1 : i’’ = skip
  i1 : instr
  i2 : instr
  i3 : instr
  Heq2 : i0 = sequence i1 (sequence i2 i3)
  ===========================
   sos_star r0 (sequence (sequence i1 i2) i3) skip r’’
```

Because `i0` is a sequence, the hypothesis can only be proved with the rule SOS2 or SOS3. We can Force the `Coq` system to consider only these two cases by rewriting first with `Heq2` in `Hstep` and then applying a specific tactic for this need.

```
lemma2_aux < rewrite Heq2 in Hstep; inversion Hstep.
 subgoals
```

```
  ...
  Hstar : sos_star r'0 i'0 i'' r''
  Hrec : i'' = skip ->
         forall i1 i2 i3 : instr,
         i'0 = sequence i1 (sequence i2 i3) ->
         sos_star r'0 (sequence (sequence i1 i2) i3)
             skip r''
  Heq1 : i'' = skip
  i1 : instr
  i2 : instr
  i3 : instr
  Heq2 : i0 = sequence i1 (sequence i2 i3)
  Hstep : sos_step r0 (sequence i1 (sequence i2 i3))
              i'0 r'0
  r1 : list (Z * Z)
  i4 : instr
  H0 : r1 = r0
  H2 : skip = i1
  H3 : i4 = sequence i2 i3
  H1 : sequence i2 i3 = i'0
  H4 : r0 = r'0
  ============================
   sos_star r'0 (sequence (sequence skip i2) i3) skip r''

subgoal 2 is:
 sos_star r0 (sequence (sequence i1 i2) i3) skip r''
```

The first goal expresses that the rule SOS6 was used. In this case, `i1` is `skip` (assumption `H2`) and `i'0` is `sequence i2 i3`. We can use the rules `SOS7`, `SOS3` and `SOS2` to conclude.

```
lemma2_aux < apply SOS7 with r'0 (sequence i2 i3).
lemma2_aux < apply SOS3.
lemma2_aux < apply SOS2.
lemma2_aux < rewrite <- H1 in Hstar;
lemma2_aux <   rewrite Heq1 in Hstar;
lemma2_aux <   exact Hstar.
```

We are left with the other case. In this case, `i1` reduce in one step to another instruction `i1'` (hypothesis `H5`. We can now use SOS7 to start a derivation.

```
lemma2_aux < apply SOS7 with r'0
lemma2_aux <      (sequence (sequence i1' i2) i3)..
...
  H5 : sos_step r0 i1 i1' r'0
  H1 : r1 = r0
  H0 : i4 = i1
  H3 : i5 = sequence i2 i3
  H2 : sequence i1' (sequence i2 i3) = i'0
  H4 : r'1 = r'0
  ============================
   sos_step r0 (sequence (sequence i1 i2) i3)
     (sequence (sequence i1' i2) i3)
     r'0

subgoal 2 is:
 sos_star r'0 (sequence (sequence i1' i2) i3) skip r''
```

For the first premise to the SOS7 rule, it is quite easy, we can use the rule SOS3 twice and conclude. Actually, this can also be done automatically.

```
lemma2_aux < auto.
1 subgoal

...
  Hstar : sos_star r'0 i'0 i'' r''
  Hrec : i'' = skip ->
         forall i1 i2 i3 : instr,
         i'0 = sequence i1 (sequence i2 i3) ->
         sos_star r'0 (sequence (sequence i1 i2) i3)
              skip r''
  Heq1 : i'' = skip
...
  Heq2 : i0 = sequence i1 (sequence i2 i3)
  Hstep : sos_step r0
            (sequence i1 (sequence i2 i3)) i'0 r'0
...
  H5 : sos_step r0 i1 i1' r'0
  H1 : r1 = r0
  H0 : i4 = i1
  H3 : i5 = sequence i2 i3
  H2 : sequence i1' (sequence i2 i3) = i'0
  H4 : r'1 = r'0
  ============================
   sos_star r'0 (sequence (sequence i1' i2) i3) skip r''
```

To prove this case, we need to use the induction hypothesis `Hrec`.

```
lemma2_aux < apply Hrec; auto.
Proof completed.
lemma2_aux < Qed.
```

The statement of this theorem is more complex than the statement of `lemma2` in the course notes. We can actually obtain a simpler lemma with the following encapsulating theorem, with the following script.

```
Theorem lemma2_1 : forall r r' i1 i2 i3,
  sos_star r (sequence i1 (sequence i2 i3)) skip r' ->
  sos_star r (sequence (sequence i1 i2) i3) skip r'.
Proof.
 intros r r' i1 i2 i3 Hstar.
 apply lemma2_aux with
    (sequence i1 (sequence i2 i3)) skip; trivial.
Qed.
```

We prove the other direction in a similar fashion. We give only the scripts, but we invite the reader to simulate the behavior of each tactic by hand or to run these tactics on the computer.

```
Theorem lemma2_aux2 : forall r r' i i',
  sos_star r i i' r' ->
  i' = skip ->
  forall i1 i2 i3, i = (sequence (sequence i1 i2) i3) ->
  sos_star r (sequence i1 (sequence i2 i3)) skip r'.
Proof.
 intros r r' i i' H; elim H.
 intros  r0 i0 Heq1 i1 i2 i3 Heq2.
 rewrite Heq1 in Heq2; discriminate Heq2.
 intros r0 r'0 r'' i0 i'0 i'' Hstep Hstar Hrec
   Heq1 i1 i2 i3 Heq2.
 rewrite Heq2 in Hstep.
 inversion Hstep.
*1 subgoal*
...
 *H5 : sos_step r0 (sequence i1 i2) i1' r'0*
...
*lemma2_aux2* < inversion H5.
 apply SOS7 with r'0 (sequence i1' i3).
 apply SOS2.
 rewrite H2; rewrite <- Heq1; trivial.
 apply SOS7 with r'0 (sequence i1'0 (sequence i2 i3));
 auto.
 apply Hrec.
 trivial.
 rewrite H8; rewrite H2;trivial.
Qed.
```

3.3 Summary of proof tactics

In the previous proofs, we only used the tactics `intro`, `apply`, `elim`, `exact`, `trivial`, `auto`, `rewrite`, `replace`, `inversion`, and `discriminate`. We summarize the behavior of each tactic here.

3.3.1 The tactic `intro`

The tactic `intro` works on goals that are universal quantifications or implications.

On implications, the tactic `intro H` replaces a goal of the form

```
...
=================
A -> B
```

into a goal with an extra assumption `H` whose statement is `A`, and leaves a goal where only `B` needs to be proved.

```
....
H : A
=================
B
```

On universally quantified formulas `intro x` replaces a goal of the form

```
...
=================
forall y:T, B
```

with a goal where a new local constant `x` is declared and where the statement to prove is `B'`, the same as `B`, but where all occurrences of `y` have been replaced with `x`

```
...
x : T
=================
B'
```

3.3.2 The tactic `apply`

The tactic `apply` take as argument a theorem or an hypothesis from the local context of the form

`forall` x_1 `...` x_n, P_1 `->` `...` P_m `->` `F` x_1 `...` x_n

When the goal is an instance of this theorem, `F` a_1 `...`a_n, it creates m goals corresponding to the theorem premises P_1 P_m where the variables $x_1 \dots x_n$ have been replaced with the variables $a_1 \dots a_m$.

If the some of the variables x_1 x_n do not occur in the theorem's conclusion, they have to be provided by the user using the `with` directive.

3.3.3 The tactic `elim`

The tactic `Elim H` can be used only if `H` is an hypothesis whose statement has an inductive predicate as conclusion. Its behavior is too complex to describe completely in these course notes, but we can give the following indications:

1. The number of generated goals is the number of constructors given for the inductive predicate,
2. Each goal has a shape similar to the shape of one of the constructors. In particular, all the premises of the constructor are present. In practice, we can read the goal as *try to prove the initial goal under the assumption that this constructor was used to prove the assumption* `H`.
3. For those premises that use the same inductive predicate, induction hypotheses are given.

3.4 The tactics `exact`, `trivial`, `auto`

The tactic `exact H` can be used if the statement to prove is exactly the statement of the hypothesis `H`.

The tactics `trivial` and `auto` are automatic tactics that use the assumptions from the local context and the theorems that have been recorded using the `Hint Resolve` command. The tactic `trivial` performs only very simple proof search.

3.4.1 The tactics `rewrite` and `replace`

The tactic `rewrite H` works if `H` is an equality and replace all occurrences of the left-hand-side of this equality by the right-hand-side. If the tactic is `rewrite <- H`, then the rewriting works the other way around. If `rewrite H in H1` is used, then the rewriting does not occur in the goal conclusion but in the presecribed assumption.

The tactic `replace` is like `rewrite` but the equality that is used to rewrite is left as an extra goal.

3.4.2 The tactic `inversion`

Like the tactic `elim H`, the tactic `inversion H` only works if the assumption `H` has an inductive predicate as statement. It then analyzes which constructor could have been applied to prove this assumption and generates as many goals as constructors that could apply, with all the information about this constructor as added assumptions. However, no induction hypothesis is created. This tactic is used every time we want to *observe a derivation* and we do not need to produce a new induction hypothesis.

3.4.3 The tactic `discriminate`

This tactic can be used every time previously used tactics generated an assumption where two different constructors from an inductive set appear to be equal. In this case the assumption is contradictory and the goal is directly solved.

4 Proving the equivalence

Now, that we have described the main tools, we will only list the various proof scripts for the lemmas. The proof explanation have already been given in the previous lemma.

```
Theorem lemma3 :
 forall r r’ i,  s_exec r i r’ ->
   forall r’’ i’, sos_star r’ i’ skip r’’
         -> sos_star r (sequence i i’) skip r’’.
Proof.
intros r r’ i H; elim H.
(* First case: rule SN1 *)
intros r0 r’’ i’ H1.
apply SOS7 with r0 i’.
apply SOS2.
exact H1.

(* rule SN2 *)
intros r0 r’0 x e v Hev Hup r’’ i’ Hsos.
apply SOS7 with r’0 (sequence skip i’).
apply SOS3.
apply SOS1 with v;trivial.
apply SOS7 with r’0 i’;auto.

(* rule SN3 *)
intros
  r0 r’0 r’’0 i1 i2 Hex1 Hrec1 Hex2 Hrec2 r’’ i’ Hsos.
apply lemma2_1.
apply Hrec1.
apply Hrec2.
trivial.

(* rule SN4 *)
intros
 r0 r’0 r’’0 b i1 Hev Hex1 Hrec1 Hex2 Hrec2 r’’ i’ Hsos.
apply SOS7 with
   r0 (sequence (sequence i1 (while b i1)) i’).
apply SOS3.
apply SOS5.
trivial.
apply lemma2_1.
apply Hrec1.
apply Hrec2.
trivial.

(* rule SN5 *)
intros r0 b i1 Hev r’’ i’ Hsos.
apply SOS7 with r0 (sequence skip i’).
apply SOS3.
apply SOS4.
trivial.
```

```
apply SOS7 with r0 i'.
apply SOS2.
trivial.
Qed.

Theorem lemma4_aux : forall r r' i0 i1,
   sos_star r i0 i1 r' ->
   forall i', i0 = (sequence i' skip) -> i1 = skip ->
   sos_star r i' skip r'.
Proof.
 intros r r' i0 i1 H; elim H.
(* rule SOS6 *)
 intros r2 i2 i' He1 He2; rewrite He1 in He2;
 discriminate.
(* rule SOS7 *)
 intros r2 r3 r4 i2 i3 i4 Hstep Hsos Hrec i' He1 He2.
 rewrite He1 in Hstep.

 inversion Hstep.
(* rule SOS2 *)
 rewrite <- H1 in Hsos; rewrite He2 in Hsos;trivial.

(* rule SOS3 *)
 apply SOS7 with r3 i1';auto.
Qed.

Theorem lemma4_1 : forall r r' i,
  sos_star r (sequence i skip) skip r' ->
  sos_star r i skip r'.
Proof.
 intros r r' i H.
 apply lemma4_aux with (sequence i skip) skip;auto.
Qed.

Theorem lemma4_aux2 : forall r r' i is,
  sos_star r i is r' -> is = skip ->
  sos_star r (sequence i skip) skip r'.
Proof.
 intros r r' i is H; elim H.

(* rule SOS6, auto uses SOS2 and SOS6 *)
 intros r1 i1 Heq.
 apply SOS7 with r1 skip.
 rewrite Heq; auto.
 auto.
```

```
(* rule SOS7 *)
 intros r1 r2 r3 i1 i2 i3 Hstep Hstar Hrec Heq.
 apply SOS7 with r2 (sequence i2 skip).
 auto.
 auto.
Qed.

Theorem lemma4_2 : forall r r' i,
  sos_star r i skip r' ->
  sos_star r (sequence i skip) skip r'.
Proof.
 intros r r' i H; apply lemma4_aux2 with skip;auto.
Qed.

Theorem sn_imp_sos :
  forall r r' i, s_exec r i r' -> sos_star r i skip r'.
Proof.
 intros r r' i Hsn.
 apply lemma4_1.
 apply lemma3 with r'; auto.
Qed.

Theorem lemma5 : forall r r' i i',
  sos_step r i i' r' ->
  forall r'', s_exec r' i' r'' -> s_exec r i r''.
Proof.
 intros r r' i i' H; elim H.

(* rule SOS1 *)
 intros r0 r'0 x e v Hev Hup r'' Hexec.
 inversion Hexec.
 rewrite <- H2; apply SN2 with v; auto.

(* rule SOS2 *)
 intros r0 i2 r'' Hexec.
 apply SN3 with r0;auto.

(* rule SOS3 *)
 intros r0 r'0 i1 i1' i2 Hstep1 Hrec1 r'' Hexec.
 inversion Hexec.

(* auto uses Hrec1 and assumptions from the inversion *)
 apply SN3 with r'1;auto.

(* rule SOS4 *)
 intros r0 b i0 Hev r'' Hexec.
 inversion Hexec.
 apply SN4 with r’0;auto.

(* rule SOS5 *)
 intros r0 b i0 Hev r’’ Hexec; inversion Hexec.
(* the inversion gives an equality r0 = r’’ *)
 rewrite <- H2; apply SN5;auto.

Qed.

Theorem sos_imp_sn_aux : forall r i is r’,
  sos_star r i is r’ -> is = skip -> s_exec r i r’.
Proof.
 intros r i is r’ H; elim H.

(* rule SOS6 *)
 intros r0 i0 Heq; rewrite Heq; apply SN1.

(* rule SOS7 *)
 intros r0 r’0 r’’0 i0 i’0 i’’0 Hstep Hstar Hrec Heq.
 apply lemma5 with r’0 i’0;auto.
Qed.
```