Semantics for programming languages with Coq encodings

Yves Bertot

Semantics for programming languages with Coq encodings
First part: natural semantics

Yves Bertot

Mars 2015

1 Why study programming language semantics

The objective of programming language semantics is to provide methods to reason about programs and make it possible to program without making mistakes. To avoid programming errors, we will rely on three techniques:

- Thanks to programming language semantics, it will be possible to describe precisely the behavior of a program and to compare it with an expected behavior, for instance in the form of logical relations between inputs and outputs.
- Semantics will also make it possible to define programming disciplines and to show once and for all how these disciplines make it possible to avoid certain classes of programming errors.
- Based on programming language semantics, we will be able to construct new computer-based tools, which observe programs and verify that they do not contain certain classes of errors.

This topic has a strong economic impact. Embedded software, used more and more often in modern appliances, vehicles, or hand-held devices, requires a much higher level of quality than past applications of computer technology. In particular, the transportation industry uses software in condition where failures could have catastrophic impact, leading to the loss of human lives. In telecommunication, in electronic banking, the consequences may not be as dire with respect to human life, but the financial consequences of software failure are humongous.

This course is devised as an introduction to different techniques used in studying programming language semantics. It is inspired from the first chapters of a book written in 1993 by Glynn Winskel, *The formal semantics of programming languages, an introduction* and published by MIT Press in the series *Foundations of Computing*. We will present the following aspects:

1. Natural semantics: presenting program execution as a logical system,
2. Proofs by induction: applications to programming languages,
3. Executing semantic specifications,
4. Proofs in semantics.

All our work will be illustrated on the study of a very small language, which represents a fragment present in `C`, `Java` or `C++`: the language of assignments and while-loops.

The various semantic concept will then be illustrated and implemented in the Coq system. For a good understanding of this system, it is advised to refer to an introductory text like *Coq in a hurry*, available at the following address:

`http://cel.archives-ouvertes.fr/inria-00001173`

2 Describing data structures

We voluntarily reduce the language we study. Still, the concepts that we will study are representative of the concepts we would encounter when studying a larger language.

2.1 Expressions et instructions

The first stage of treatment of a program is syntax analysis. The result of this analysis is an *abstract syntax term*. The description of the programming language starts with the description of the set of all these abstract syntax term.

- We assume the existence of V, a set whose elements are called variables and are noted v, v', v_i, w, x, ...
- We then have a set of arithmetic expressions, $aexp$, whose elements are noted e, e'. This set contains the variables v, the integers n, and additions of two arithmetic expressions, $e_1 + e_2$. In this context, $+$ does not represent an operation on numbers, but a composed arithmetic expression, which combines two existing arithmetic expressions. Sometimes, to make this distinction clear, we will also write $plus(e_1, e_2)$. In the literature about programming language, the description of abstract syntax terms for arithmetic expressions is sometimes given in the following condensed form:

  $$e ::= v \mid n \mid e_1 + e_2$$

  For a more general programming language, there could be more forms of composed arithmetic expressions. For instance, the language could also contain multiplication and subtraction, but for our study it will be enough to study a language where addition is the only available operation.

- We then have a set of boolean expressions *bexp*, whose elements are noted b, b', etc, and in our small language we will consider only one kind of boolean expression, noted $e_1 < e_2$ where e_1 and e_2 are two arithmetic expressions (defined in the previous item. Sometimes, to distinguish abstract syntax terms from actual comparisons between integers, we will write $blt(e_1, e_2)$
- We then have a set of instructions, whose elements will be noted I, I', etc, and there will be only three forms of expressions, assignments, sequence pairs, and while loops: le langage des instructions, *instr*, dont les éléments seront $I ::= skip \mid v := e \mid (I_1; I_2) \mid$ `while` b `do` I `done`. Again, when emphasing that we are working on abstract syntax term, we will write $assign(v, e)$, $seq(I_1, I_2$, $while(b, I)$ instead. The *skip* instruction is used to represent a program that does nothing.

In this language we can represent the following program fragment:

```
v1 := 0;
v2 := 0;
while v2 < v3 do
  v1 := v2 + v1;
  v2 := v2 + 1
done
```

When the initial value of `v3` is positive, this program fragment computes

$$(0 + 1 + \cdots),$$

with `v3` elements in the sum.

Exercises

1. *Write the abstract syntax term for a program that exchanges the values of two variables v_1 and v_2, while using a third intermediate variable v_3.*
2. *Write the abstract syntax term for a program that computes the product of two variables (assuming their initial inputs are positive integers).*
3. *Write the abstract syntax term for a program that computes in a variable v_2 the value of the factorial of v_1, assuming v_1 initially contains a positive integer.*

2.2 Coq encoding

We will use the Coq system to perform computer-assisted experiments about the semantics of the programming language. In Coq, the representation of datatypes like abstract syntax terms is usually given as *inductive datatypes*.

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

Inductive aexpr : Type :=
  avar (s:string) | anum (x:Z) | aplus (e1 e2 : aexpr).

Inductive bexpr : Type :=
  blt (e1 e2 : aexpr).

Inductive instr : Type:=
  skip | assign (s:string) (e:aexpr)
| sequence (i1 i2 : instr) | while (b : bexpr) (i:instr).
```

The expression `assign "x" (aplus (avar "x") (anum 1)` represents the instruction $x := x + 1$.

3 Machine state

The behavior of an imperative program can be expressed by the modification of a state. We must choose how to represent this state. For the language we are studying, it is enough to model this state by a correspondence between program variables and values. There is only a finite number of variables in the program and we represent the state as a list of pairs, where the first element is a variable and the second one is a value. This list of pairs will also be a term in a language:

- The language of pairs contains terms of the form $pair(v, n)$ where v is a variable and n is a number nombre
- the language of machine states *state*, whose elements will be noted s, s', etc, is the set of terms of two possible forms, *nil* or $cons(pair(v, n), s)$.

For more concision, we will often write

$$(v, n) \cdot s$$

instead of

$$cons(pair(v, n), s).$$

3.1 Representing states with Coq

Coq provides its own implementation of lists and pairs. The type of pairs is usually called a *cartesian product*. The type of states will therefore be represented as follows:

```
Definition state := list (string * Z).
```

4 Semantic descriptions

4.1 Judgments

In our work on the description of programming languages, we will reason on statements of the form *excuting instruction i from the state s returns the new state s'* or *evaluating variable v in the state s returns the numerical value n.* For each of this kind of sentence, we will use an abbreviated notation, whose style is borrowed from proof theory.

In this section we will enumerate the different kinds of statements that appear in our description of programming languages, trying to make the notations less frightening.

For each of these statements, we will use notations with various kinds of arrows, like $\rightarrow$, $\mapsto$, or $\leadsto$. The meaning attached to each of these arrows is a matter of convention.

4.1.1 Evaluating expressions

To determine the value of an expression in a given machine state, we use a statement of the form *in the state s, the expression e has value n.* We write this statement in the following way:

$$s \vdash e \rightarrow n$$

It is not useful to look for a precise meaning to the symbols $\vdash$ and $\rightarrow$. We only want to have a short notation for the phrase, which describes a relation between three terms. The $\vdash$ symbol is often used in proof theory to separate a context of known facts from a formula to be proved. In this case, the state represents known facts: the values of each variable.

We use the same notation to represent what happens when evaluating boolean expressions.

4.1.2 Updating the state

Assigning the value of an expression to a variable provokes the change of the value associated to this variable. In our logical representation, this is implemented as the creation of a new state whose only difference with the previous state lies in the value associated to just that variable. We will use a statement of the form *in the state s, updating variable v with the value n returns the new state s'.* We will write this statement as follows:

$$s \vdash v, n \mapsto s'$$

Here again, the use of symbols $\vdash$ and $\mapsto$ is purely a matter of convention.

4.1.3 Executing instructions

When executing an instruction, one modifies the state of the machine. We shall be using statements of the form *in the state s, executing instruction I returns the new state s'*. This statement will be written as follows:

$$s \vdash I \rightsquigarrow s'.$$

When this judgment is used, this also expresses that the execution of I terminates (it does not loop forever).

4.1.4 Encoding these judgments in Coq

In Coq, we shall use a three argument predicate for $s \vdash e \to n$, in other words a function `aeval` that takes three arguments and returns a proposition. Thus, the result has type `Prop`.

```
aeval : state -> aexpr -> Z -> Prop
```

For instance the judgment $s \vdash e \to n$ will be represented in Coq by `aeval` s e n.

We also have to use the notation $s \vdash e \to v$ when e is a boolean expression and v is boolean. In this case, we need to use a different predicate, which we will call `beval`

```
beval : state -> bexpr -> bool -> Prop
```

For updating the state, it is a four-place predicate that we need:

```
update : state -> string -> Z -> state -> Prop
```

Last for executing instructions, we again use a three-place predicate

```
exec : state -> instr -> state -> Prop
```

We shall see later how these predicates are defined.

4.2 Auxiliary logical statements

W will also need simple statements about natural numbers, of the form $n_1 < n_2$, $n_1 \leq n_2$, or $n_1 + n_2 = n_3$. The meaning of these statements will always be simple to understand, since n_1, n_2, and n_3 are integer values, which are well known at the time the question is posed.

In Coq, these logical formula will be directly encoded as Coq logical formulas.

4.3 How to reason: Inference rules

In the approach of this chapter, describing the semantics of a programming language is done by describing the principles that make it possible to assert that some statement holds. These principles have the shape of *reasoning steps* which combine several statements.

The notation is again inspired from proof theory. Reasoning steps are described as inference rules, which all have the form *if the propositions* P_1, ..., P_k *sont already proved, then we can also proven the proposition* Q. The tradition from proof theory is to draw these inference rules with an horizontal bar, as follows:

$$\frac{P_1 \dots P_k}{Q}$$

Propositions P_i are usually called *premises* while Q is called the *conclusion*.

Rules present reasoning steps in a schematic way, in the sense that they can be applied in a variety of sitations. More precisely, the propositions P_1, ..., P_k, Q may contain variables, which possibly occur in several propositions, and each inference rule describes the cases that may happen when the variables are replaced by arbitrary terms. In this way, a finite number of rules can represent the infinite set of all possible executions.

For Coq, the horizontal bars used in inference rules simply represent implications and we will directly use Coq's arrows (which also have a meaning of implication) to represent these bars.

We shall now enumerate the whole set of inference rules that make it possible to describe the semantics of our example language.

4.4 Evaluating variables

To know the value of a variable v in a given state s, it is necessary that this value is given in the state. Because the state is represented by a list of pairs, two cases may happen, depending on whether the first pair mentions the variable v or not.

If the variable appears in the first pair, then s has the following shape:

$$(v, n) \cdot s'$$

where s' is some other state. In this case the value of the variable must be n. We express this with the following inference rule:

$$\frac{}{(v, n) \cdot s' \vdash v \to n} \tag{1}$$

Note that this inference rule has no premises.

Here is a second reasoning rule: if a variable v has a value in a state s, then it has the same value in a state where an extra pair for a different variable is added in front. This is expressed by the following inference rule:

$$\frac{s \vdash v \to n \quad v \neq v'}{(v', n') \cdot s \vdash v \to n} \tag{2}$$

These two inference rules are enough to describe the value of a variable in any state, as long as the state contains a pair concerning that variable.

4.5 Evaluating expressions

Aside from variables, we also have to evaluate numbers and addition expressions.

4.5.1 Immediate numerical expressions

For numerical values, the situation is simple: their value does not depend from the state and is the same numerical value. This is expressed by the following inference rule:

$$\frac{}{s \vdash n \to n} \tag{3}$$

4.5.2 Addition

If the evaluation of two expressions e_1 and e_2 in a same state s returns two numerical values n_1 and n_2, then evaluating the expression $plus(e_1, e_2)$ in the same state s returns the sum of n_1 and n_2:

$$\frac{s \vdash e_1 \to n_1 \quad s \vdash e_2 \to n_2}{s \vdash plus(e_1, e_2) \to n_1 + n_2} \tag{4}$$

4.5.3 Coq encoding

We give all the evaluation rules for expressions in one definition. At the same time, this defines the predicate `aeval` and four theorems that make it possible to prove instances of this predicate.

```
Inductive aeval : state -> aexpr -> Z -> Prop :=
  ae_num : forall r n, aeval r (anum n) n
| ae_var1 : forall r x v, aeval ((x,v)::r) (avar x) v
| ae_var2 : forall r x y v v' , x <> y -> aeval r (avar x) v' ->
              aeval ((y,v)::r) (avar x) v'
| ae_plus : forall r e1 e2 v1 v2,
              aeval r e1 v1 -> aeval r e2 v2 ->
              aeval r (aplus e1 e2) (v1 + v2).
```

This technique to describe `aeval` also expresses that the four theorems `ae_num`, `ae_var1`, `ae_var2`, and `ae_plus` are the only way to prove instances of the `aeval` predicate, moreover these rules must be used a finite number of times. All this is implied by the notion of *inductive predicate*.

4.6 Evaluating boolean expressions

4.6.1 Comparisons

We can use several rules for the same operator. For instance, the comparison can be treated almost like addition, by first stating that the two expressions e_1 and e_2 are evaluated in the same state and return two numerical values. One inference rule can then be used to express when the comparison of these two expressions returns a boolean value *true* and another rule can be used to express when the comparison returns *false*.

$$\frac{s \vdash e_1 \to n_1 \quad s \vdash e_2 \to n_2 \quad n_1 < n_2}{s \vdash blt(e_1, e_2) \to true} \tag{5}$$

$$\frac{s \vdash e_1 \to n_1 \quad s \vdash e_2 \to n_2 \quad n_2 \leq n_1}{s \vdash blt(e_1, e_2) \to false} \tag{6}$$

4.6.2 Coq encoding

```
Inductive beval : state -> bexpr -> bool -> Prop :=
| be_lt1 : forall r e1 e2 v1 v2,
            aeval r e1 v1 -> aeval r e2 v2 ->
            v1 < v2 -> beval r (blt e1 e2) true
| be_lt2 : forall r e1 e2 v1 v2,
            aeval r e1 v1 -> aeval r e2 v2 ->
            v2 <= v1 -> beval r (blt e1 e2) false.
```

4.7 Updating the value for a variable

4.7.1 Inference rules

If we want to associate the value n to the variable v, we will have to produce a new machine state. This modification is easy to express if the variable v appears in the first pair of the state: the numerical value must simply be changed to n, while the rest of the state is unchanged. This is expressed by the following inference rule:

$$\frac{}{(v, n') \cdot s \vdash v, n \mapsto (v, n) \cdot s} \tag{7}$$

On the other hand, if we know how to update the state s for the variable v and the value n, thus obtaining a new state s', then updating a state where the first pair concerns a variable v' that is different from v and the rest is s, then the value of the first variable must be kept unchanged, but the rest of the resulting state can still be s'. This is described quite clearly in the following inference rule (which is actually clearer than the plain text explanation).

$$\frac{s \vdash v, n \mapsto s' \qquad v \neq v'}{(v', n') \cdot s \vdash v, n \mapsto (v', n') \cdot s'} \tag{8}$$

4.7.2 Coq encoding

```
Inductive update : state -> string -> Z-> state -> Prop :=
| s_up1 : forall r x v v', update ((x,v)::r) x v' ((x,v')::r)
| s_up2 : forall r r' x y v v', update r x v' r' ->
            x <> y -> update ((y,v)::r) x v' ((y,v)::r').
```

Exercises

4. *If one adds exact boolean values true and false, what should be the inference rules that describe how to evaluate these expressions (see what is done for immediate numerical values).*

5. *If one adds an expression constructor minus for subtraction, how would the evaluation for this kind of arithmetic expression be described in inference rules.*

6. *Same question for an equality comparison operator.*

4.8 Executing instructions

4.8.1 Assignment

An assignment is composed of a variable v and an expression e. If evaluating the expression e in a state s returns a value n and updating this state s for the variable v and the value n produces a new state s', then the execution of the assignment v `:=` e (also written $assign(v, e)$) in the state s terminates and returns the new state s'.

$$\frac{s \vdash e \to n \quad s \vdash v, n \mapsto s'}{s \vdash assign(v, e) \rightsquigarrow s'} \tag{9}$$

4.8.2 sequences

If executing an instruction I_1 in a state s terminates and returns a new state s' and executing the instruction I_2 in the state s' also terminates and returns the state s'', then executing the instruction I_1 `;` I_2 (also written $sequence(I_1, I_2)$) in the state s terminates and returns the state s''.

$$\frac{s \vdash I_1 \rightsquigarrow s' \quad s' \vdash I_2 \rightsquigarrow s''}{s \vdash sequence(I_1, I_2) \rightsquigarrow s''} \tag{10}$$

4.8.3 Loop

if the expression e evaluates to *true* in the state s, if the instruction I executes normally in states s and terminates, outputting a new state s', and if the loop $while(e, i)$ can be executed in state s', terminates, and returns the state s'',

then executing the same loop instruction $while(e, i)$ in state s terminates and returns the state s''.

$$\frac{s \vdash e \to true \quad s \vdash i \rightsquigarrow s' \quad s' \vdash while(e,i) \rightsquigarrow s''}{s \vdash while(e,i) \rightsquigarrow s''} \tag{11}$$

Another inference rule describes the behavior of a loop instruction when the expression evaluates to $false$. In this case, the execution terminates immediately and returns the same state:

$$\frac{s \vdash e \to false}{s \vdash while(e,I) \rightsquigarrow s} \tag{12}$$

4.8.4 skip

The *skip* instruction does not perform any action. This is expressed by the fact that executing this instruction in any state always terminates and returns the same state.

$$\frac{}{s \vdash skip \rightsquigarrow s} \tag{13}$$

4.8.5 Coq encoding

```
Inductive exec : env->instr->env->Prop :=
| SN1 : forall r, exec r skip r
| SN2 : forall r r' x e v,
   aeval r e v -> update r x v r' -> exec r (assign x e) r'
| SN3 : forall r r' r'' i1 i2,
    exec r i1 r' -> exec r' i2 r'' ->
    exec r (sequence i1 i2) r''
| SN4 : forall r r' r'' b i,
    beval r b true -> exec r i r' ->
    exec r' (while b i) r'' ->
    exec r (while b i) r''
| SN5 : forall r b i,
    beval r b false -> exec r (while b i) r.
```

Exercises

7. *Add in the language an instruction $if(e, I_1, I_2)$ and the necessary inference rules.*

8. *Supposing that the if instruction is already described, write a new inference rule for loops that re-uses the behavior of if.*

5 Using semantic specifications

The inference rules are designed to be composed, using some rules to prove the premises of instances of another rule. We shall illustrate this on an example. The important point is that derivations made of composed inference rules can be used to represent traces of execution.

5.1 Building derivations as graphical figures

Let us consider the following sequence of instructions

```
x := x + y;
y := x + (- 4);
x := x + (- 3)
```

This is a sequence of instructions, where the first component is an assignment (`x := x + y`) and the second component is itself a sequence of instructions.

We want to describe the execution of this sequence from the state where `y` is initially 4 and `x` is initially 3.

We will use the following abbreviations:

$$\begin{array}{lcllcl} P_2 & = & \text{y} := \text{x} + (\text{-}\ 4);\ \text{x} := \text{x} + (\text{-}\ 3) & P_1 & = & \text{x} := \text{x} + \text{y};\ \ P_2 \\ s_0 & = & (\text{“x”}, 3) \cdot (\text{“y”}, 4) \cdot \emptyset & s_1 & = & (\text{“x”}, 7) \cdot (\text{“y”}, 4) \cdot \emptyset \\ s_2 & = & (\text{“x”}, 7) \cdot (\text{“y”}, 3) \cdot \emptyset & s_3 & = & (\text{“x”}, 4) \cdot (\text{“y”}, 3) \cdot \emptyset \\ s_4 & = & (\text{“y”}, 4) \cdot \emptyset & & & \end{array}$$

We shall now build a derivation for the following statement:

$$s_0 \vdash P_1 \rightsquigarrow s_3$$

The main structure of the derivation has the following shape:

$$\dfrac{\dfrac{D_1}{s_0 \vdash \texttt{x:=x+y} \rightsquigarrow s_1} \qquad \dfrac{\dfrac{D_2}{s_1 \vdash \texttt{y:=x + (-4)} \rightsquigarrow s_2} \qquad \dfrac{D_3}{s_2 \vdash \texttt{x:=x + (-3)} \rightsquigarrow s_3}}{s_1 \vdash P_2 \rightsquigarrow s_3}}{s_0 \vdash P_1 \rightsquigarrow s_3}$$

In this picture, D_1, D_2 et D_3 represent sub-derivation, each one of them concerned with the execution of an assignment. Here is derivation D_1:

$$\dfrac{D_4 \qquad \overline{(\text{“x”}, 3) \cdot s_4 \vdash \text{“x”}, 7 \mapsto (\text{“x”}, 7) \cdot s_4}}{s_0 \vdash \texttt{x:=x+y} \rightsquigarrow s_1}$$

and here is derivation D_4:

$$\dfrac{\dfrac{}{(\text{“x”}, 3) \cdot (\text{“y”}, 4) \cdot s \vdash \text{“x”} \to 3} \qquad \dfrac{\dfrac{}{(\text{“y”}, 4) \cdot s \vdash \text{“y”} \to 4} \quad \text{“x”} \neq \text{“y”}}{(\text{“x”}, 3) \cdot (\text{“y”}, 4) \cdot s \vdash \text{“y”} \to 4}}{s_0 \vdash \texttt{x+y} \to 7}$$

The proof that $x \neq$ “y” holds does not rely on any of the inference rules that we described for this language, so we will not build a derivation, but this proof must necessarily be checked.

Exercises

9. *Build derivations D_2 and D_3.*

10. *Build a derivation for the following statement*

$$(\text{“x”}, 1) \cdot s \vdash \texttt{while(2>x)\{x:= x+x\}} \rightsquigarrow (\text{“x”}, 2) \cdot s$$

11. *We define the size of an arithmetic expression to be the number of operators used to build this expression (counting variables and integers as 1 each). For instance the expression $plus(x, 1)$ has size 3. For derivations, we consider that the size of derivation is the number of rules. What is the minimal size for a derivation concerning the evaluation of an arithmetic expression of size n. If the state in which evaluation occurs has size p, what is the maximal size for a derivation concerning an arithmetic expression of size n.*

12. *What is the size of the derivation for executing $while(x > y, assign(y, y + 1))$ in the state $(x, n) \cdot (y, 0) \cdot s$?*

13. *What is the size of the derivation for executing*

$$while(x > y, sequence(while(x > z, assign(z, z + 1)), assign(y, y + 1)))$$

in the state$(x, n) \cdot (y, 0) \cdot (z, 0) \cdot s$?

5.2 Representing derivations in Coq

In Coq, derivations are proofs of logical statements. These proofs are obtained by composing the constructors of the inductive definitions. Each time an inference rule is used in a derivation, the corresponding constructor can be used in the encoding of this derivation inside Coq.

This can be illustrated by looking again at the derivations that we built in the previous section. For instance, derivation D_4 can be represented in the following way:

```
Lemma D4 :
  aeval (("x", 3)::("y", 4)::nil)(aplus (avar "x")(avar "y")) 7.
Proof.
apply ae_plus with (v1 := 3) (v2 := 4).
  apply ae_var1.
apply ae_var2.
  discriminate.
apply ae_var1.
Qed.
```

The use of the three constructors from the definition `aeval`: `ae_plus` for addition, `ae_var1` and `ae_var2` for the variables, appears clearly in this proof. On

the other hand, the proof that `"x" <> "y"` is done by calling another Coq proof tactic (`discriminate`).

The derivation D_1 can be represented by the following Coq proof, where we re-use the proof D_4.

```
Lemma D1 :
  exec (("x", 3)::("y", 4)::nil)
       (assign "x" (aplus (avar "x")(avar "y")))
       (("x", 7)::("y", 4)::nil).
apply SN2 with (v := 7).
  exact D4.
apply s_up1.
Qed.
```

5.3 Reasoning on derivations

5.3.1 Simple case-by-case reasoning

When we want to prove a statement of the form $s \vdash e \to n \Rightarrow P(s, e, n)$, we can use the knowledge that the statement $s \vdash e \to n$ was necessarily proved using the inference rules. We can then reason by cases, drawing information from the form of the rules that could have been used.

For instance, we can prove that the value computed when evaluating the variable y in the state $(\text{“x”}, 1) \cdot (\text{“y”}, 2) \cdot \emptyset$ is necessarily 2. The proof works as follows:

1. To prove the statement $(\text{“x”}, 1) \cdot (\text{“y”}, 2) \cdot \emptyset \vdash \text{“y”} \to n$, it was not possible to use the rule (3) or the rule for sums (4). Moreover, among the rules that concern the evaluation of variable, we were not able to use the rule (1), because this rule requires that the first variable in the state is the same as the variable one wants to evaluate. Here the variable at the head is “x” and the variable we want to evaluate “y”. The only rule that could have been used is the rule (2) that we introduced around page 7. Since this rule is applied we know that the premise $(\text{“y”}, 2) \cdot \emptyset \vdash \text{“y”} \to n$ was also proved.

2. To prove $(\text{“y”}, 2) \cdot \emptyset \vdash \text{“y”} \to n$, we know that the rules (2), (3), and (4) could not have been used. In particular, the rule (2) could not have been used because this rule requires that the first variable in the state and the variable being evaluated should be different. Thus, only the rule (1) could have been applied

   $$(x, n) \cdot s \vdash x \to n$$

   This rule imposes that the number value that will be the result of evaluation has to be the same number as the one in the first pair of the state. Thus, it is necessarily 2.

5.3.2 Reasoning by induction on derivations

Another criterion that we can use to reason on derivations is their size. In particular, the size of derivations can be used to reason by induction.

In the case of a derivation for evaluation, we suppose that there exists a derivation d that proves a statement of the form $s \vdash e \to n$ we can then attempt to prove a statement of proposition $P(s, e, n)$. When we perform a proof by induction on the derivation d, this means that we can use an induction hypothesis, which states that a similar property is satisfied for all derivations that have a smaller size than d. It is not absolutely necessary to define what is the size of a derivation, because most of the time we only use the induction hypothesis associated to sub-derivations of d.

For illustration, we will prove that the evaluation of an arithmetic expression always returns the same result.

Theorem *if* $s \vdash e \to n$ *and* $s \vdash e \to n'$ *then* $n = n'$.

Démonstration. Let us consider a derivation D and let's prove by induction on the size of this derivation that for any state s, any arithmetic expression e and any n, if D proves $s \vdash e \to n$ then for every n' such that $s \vdash e \to n'$ is also provable, we have $n = n'$.

The fact that we are performing a proof by induction gives us the following fact: for every derivation δ that is smaller than D, if δ proves $s_\delta \vdash e_\delta \to n_\delta$ and if we know that $s_\delta \vdash e_\delta \to n'_\delta$ is also provable, then we can deduce $n_\delta = n'_\delta$.

The rule that starts the derivation D is necessarily one of the four rules in the semantic definition. We can use this to reason by cases:

1. If D is built on rule (1), then the terms s and e are necessarily of the following shape:

$$\begin{aligned} s &= (v, n) \cdot s' \\ e &= v \end{aligned}$$

   In this case, the second derivation proving $s \vdash e \to n'$ can only use the rule (1). Indeed, all other cases are impossible: the rule (2) can't be used because the variable in the first pair of the state would have to be different from the variable being evaluated; the rule (4) cannot be used because the expression being evaluated would have to be an addition (but here it is a variable); the rule (3) cannot be used because the expression being evaluated would have to be an integer.

   Since the rule used to prove $(v, n) \vdash v \to n'$ is necessarily the rule (1), we necessarily have $n = n'$.

2. if D was built with the rule (2), then the terms s and e necessarily have the following shape

$$\begin{aligned} s &= (v', n_1) \cdot s' \\ e &= v \end{aligned}$$

for two arbitrary variables v' and v. Moreover, there must be a proof of the premises for this rule. So there must be a derivation D' to prove $s' \vdash v \to n$ and we know also that $v \neq v'$ holds. Because of this, the statement $s \vdash e \to n'$ is necessarily of the form $(v', n_1) \cdot s' \vdash v \to n'$. This statement cannot have been proved using the rule (1) because this would require $v = v'$. Thus, only the rule (2) can have been used and there necessarily also exists a proof of $s' \vdash v \to n'$.

Let us notice that the derivation D' is a sub-derivation of the derivation D, thus it is a smaller derivation, it proves $s' \vdash v \to n$ and we have another proof for a statement $s' \vdash v \to n'$. Using the induction hypothesis, we can deduce that $n = n'$.

3. If the derivation D is obtained by using the rule (3) then the expression e is the integer value n. The proof of $s \vdash e \to n'$ must also have the shape $s \vdash n \to n'$ and can only have been proved with the rule (3), and therefore $n = n'$.

4. If the derivation D starts with the rule (4), then there exist two expressions e_1 and e_2 so that $e = plus(e_1, e_2)$ and two numbers n_1 and n_2 so that $n = n_1 + n_2$ and the two premises of the rule with statements $s \vdash e_1 \to n_1$ and $s \vdash e_2 \to n_2$ must have been proved by two derivations D_1 and D_2. Moreover, since e is $plus(e_1, e_2)$, the statement $s \vdash e \to n'$ can only have been proved by the rule 4 and there must exist two numbers n'_1 and n'_2 so that $n' = n'_1 + n'_2$ and so that the statements $s \vdash e_1 \to n'_1$ and $s \vdash e_2 \to n'_2$ are also provable. We can now use the induction hypothesis twice, once for D_1 and the proof of $s \vdash e_1 \to n'_1$ to obtain $n_1 = n'_1$ and once for D_2 and the proof of $s \vdash e_2 \to n'_2$ to obtain $n_2 = n'_2$. We can then deduce that

$$n = n_1 + n_2 = n'_1 + n'_2 = n'$$

This finishes our proof.

5.4 Reasoning about derivations in Coq

As we did before, we will now illustrate how the previous proofs can be performed in the Coq system. There will be one key difference, however, because Coq naturally provides a framework where proofs by induction on derivations are covered, but induction hypotheses are given only for the sub-proofs (not for all proofs that have a smaller size). The advantage of this approach is that there is no need to define what is the size of a derivation, a question that is more tricky than it it seems at first sight.

5.4.1 Proof by cases

In the first example, we want to show that in a specific state, the evaluation of a specific variable is bound to return exactly the value prescribed by the state. Here is the example:

```
Lemma eval_example :
 forall n, aeval (("x", 1)::("y", 2)::nil) (avar "y") n -> n = 2.
```

There are many different approaches for this proof. Experienced Coq users will use a tactic called `inversion`. This tactic makes it possible to use the fact that only the four constructors of the inductive definition could have been used to prove the statement

```
aeval (("x", 1)::("y", 2)::nil) (avar "y") n
```

and even more precisely, it analyses which of the rules could really apply. For instance, it sees that the constructor named `ae_var1` could not have been used, because `"x"` and `"y''` would have to be equal, which they are obviously not. Similarly, it recognizes that `ae_num` and `ae_plus` do not apply, because they expect a datatype constructor different from `avar` for the second argument to the predicate `aeval`.

Here is how to use `inversion`:

```
intros n hyp.
inversion hyp as [ | | s y x N' N dif aev | ].
```

The text between square brackets `[ | | s y x N' N dif aev | ]` gives us the possibility to choose the names that are generated by the inversion tactic for all the arguments of the third constructor in the `aeval` definition. We do not bother giving names to the arguments of the other constructors, because we know that these constructors cannot be used, and the `inversion` tactic will solve and discard these goals without our intervention. The new goal has the following shape.

```
  n : nat
  hyp : aeval (("x", 1)::("y", 2)::nil) (avar "y") n
  s : state
  y : string
  x : string
  N' : nat
  N : nat
  dif : "y" <> "x"
  aev : aeval (("y", 2)::nil) (avar "y") n
  ...
  ======================
   n = 2
```

We can then use the tactic inversion again. However, this time, it is not clever enough to discover on its own that the constructor `ae_var2` is not applicable. The tactic generates two goals, the first one is trivial, because it requires that we prove `2 = 2`, the second is more complex, but it contains a silly hypothesis, which shows that this case is actually impossible.

We call `inversion` again in the following manner:

```
inversion aev as [ | | s' y' x' N'' N2 dif2 | ].
```

The first generated goal has the following shape:

```
  ...
  ====================
   2 = 2
```

This is solved by the following tactic:

```
reflexivity.
```

The second generated goal has the following shape:

```
  dif2 : "y" <> "y"
  ...
  ====================
   n = 2
```

This goal can easily be solved by exploiting the absurd hypothesis named `dif2`, for instance by using the following tactic.

```
case dif2; reflexivity.
```

This finishes our proof. We can save it by typing the usual proof-saving command.

```
Qed.
```

5.4.2 Proof by induction

We want to prove that when $s \vdash e \to n$ and $s \vdash e \to n'$ are both provable, then $n = n'$. In coq, this statement can be specified as the following lemma.

```
Lemma eval_deterministic :
  forall s e n n', aeval s e n -> aeval s e n' -> n = n'.
```

To begin with, let's note that the whole proof fits in a dozen lines, given here:

```
intros s e n n' aev; revert n'.
induction aev as
  [s n|s x v|s x y v v' dif A Ia|s e1 e2 v1 v2 A1 I1 A2 I2].
   intros n' aev'; inversion aev'; reflexivity.
  intros n' aev'; inversion aev' as [ | | ? ? ? ? ? dif2 | ];
     [reflexivity | case dif2; reflexivity].
 intros n' aev'; inversion aev'; [ | apply Ia; assumption ].
 case dif; solve[auto].
intros n' aev'; inversion aev' as [ | | | ? ? ? v1' v2'].
assert (v1 = v1' /\ v2 = v2') as [vv1 vv2] by auto.
rewrite vv1, vv2; reflexivity.
```

We will now explain this dozen lines in more detail.

The first line introduces all the universally quantified variables in the context, together with the first assumption. It then reverts the variable `n'` to the goal's conclusion, so that it remains universally quantified. The goal takes the following shape.

```
  aev : aeval s e n
  ======================
   forall n':nat, aeval s e n' -> n = n'
```

It is important that `n'` is universally quantified in the goal, as it will change the shape of induction hypotheses during our proof by induction.

The next step is to instruct Coq to perform a proof by induction. This is done with the following command:

```
induction aev as
  [s n|s x v|s x y v v' dif A Ia|s e1 e2 v1 v2 A1 I1 A2 I2].
```

The text between square brackets is used to choose the names of the various variables and hypotheses created by the `induction` tactic. In particular, the fragment `s x y v v' dif A Ia` takes care of the third construct, `ae_var2`, whose statement we recall here:

```
forall r x y v v' , x <> y -> aeval r (avar x) v' ->
              aeval ((y,v)::r) (avar x) v'
```

When the constructor `ae_var2` is applied, it means that this statement is instantiated, so there must exists some term for the variable that is universally quantified with the name `r`. We instruct Coq to place an element in the context, named `s` to represent this term, and so on. So the universally quantified `x`, `y`, `v`, and `v'` are represented in the goal context by variables named `x`, `y`, `v`, and `v'`. Now, the hypotheses `x <> y` and `aeval r (avar x) v'` are also introduced in the context and the collection of names we provided require that the name for the former should be `dif` and the name for the latter should be `A`. The hypothesis `A` deserves a specific treatment, its statement is an instance of the predicate `aeval`, the same predicate that appeared in the initial `aev` hypothesis. For this reason, an extra hypothesis is generated, which contains the induction hypothesis, whose statement is the instance of the initial goal that corresponds to `A`, in the same way that the initial goal corresponded to `aev`. Let us recall what are these correspondences.

The goal just before the tactic `induction aev as ...` had the following shape:

```
  aev : aeval s e n
  ======================
   forall n':nat, aeval s e n' -> n = n'
```

Now the hypothesis `A` has the following statement:

```
A :  aeval s (avar x) v'
```

Then the statement of the induction hypothesis must be:

```
forall n': nat, aeval s (avar x) n' -> v' = n'
```

To obtain this statement, we took the initial goal's conclusion, and replaced `s` with `s`, `e` with `(avar x)`, and `n` with `v'`.

This is the statement of the induction hypothesis, and we instructed Coq to give the name `Ia` to this hypothesis. As a consequence, the third goal generated by the `induction` tactic will have the following form:

```
...
dif : x <> y
A : aeval s (avar x) v'
Ia : forall n' : Z, aeval s (avar x) n' -> v' = n'
============================
 forall n' : Z, aeval ((y, v) :: s) (avar x) n' -> v' = n'
```

We also gave names to the various hypotheses appearing in the fourth case, and we will see that in this case there are two induction hypotheses that are generated, and these induction hypotheses receive the names `I1` and `I2`.

Let us now come back to the four goals generated by the induction tactic. Each of these goals corresponds to one of the inference rules, so treating these four goals is the Coq way of performing the proof by cases that was described in Section 5.3.2.

First case. The first case corresponds to Rule `ae_num`, which we recall here:

```
ae_num : forall r n, aeval r (anum n) n
```

This rule is only quantified over two variables, and the variable `n` appears at two different places.

The goal has the following shape.

```
s : state
n : Z
============================
 forall n' : Z, aeval s (anum n) n' -> n = n'
```

The two universally quantified variables were renamed according to the first range of names provided between brackets to the `induction` tactic. The initial goal was reproduced, but with the replacements corresponding to the modification from `aeval s e n` to `aeval s (anum n) n`: `e` was replaced with `(anum n)` and `n` was replaced with `n`.

We can now perform the usual operations to reason on universal quantifications and implications.

```
intros n' aev'.
```

At this point, `aev'` has the statement `aeval (anum n) n'`. In Section 5.3.2, we perform a second step of reasoning by cases, to show that this statement can only be proved using the rule `ae_num`. This step of reasoning cases, which also prunes the absurd cases is done by `inversion`.

```
inversion aev'.
```

This tactic returns only one goal, with the following shape.

```
  s : state
  n : Z
  n' : Z
  aev' : aeval s (anum n) n'
  r : state
  n0 : Z
  H : r = s
  H1 : n0 = n
  H0 : n = n'
  ============================
   n' = n'
```

So, the reasoning step performed by this tactic makes it clear that the values `n` and `n'` must be the same. We can easily conclude using the reflexivity of equality.

```
reflexivity.
```

Second case. Let us now observe the second goal generated by the induction step. This goal corresponds to the constructor `ae_var1` of the inductive definition, which we recall here:

```
 ae_var1 : forall r x v, aeval ((x,v)::r) (avar x) v
```

The goal has the following shape:

```
  s : list (string * Z)
  x : string
  v : Z
  ============================
   forall n' : Z, aeval ((x, v) :: s) (avar x) n' -> v = n'
```

This time the state `s` (from the hypothesis `aev`, just before the call to the `induction aev as ...` tactic) has been replaced with `((x, v) :: s)` and the expression `e` has been replaced by `(avar x)`. We wish again to introduce two elements in the context and to reason by cases on the premise that relies on the `aeval` predicate. We do this in the following manner.

```
  intros n' aev'; inversion aev' as [ | | ? ? ? ? ? dif2 | ].
```

Note that we use square brackets for the inversion tactic. This is because we want to control the name given to the assumption corresponding to the verification that two variables must be distinct when using the third constructor of the `aeval` predicate, we use question-mark characters `?` for all the other names, which do not matter.

This use of the `inversion` tactic generates two goals. The first goal corresponds to the case where `aeval ((x, v)::s) (avar x) n'` was proved using the constructor `ae_var1` and in this case, `v = n'`. This goal has the following shape:

```
...
aev' : aeval ((x, v) :: s) (avar x) n'
r : list (string * Z)
x0 : string
v0 : Z
H0 : x0 = x
H1 : v0 = v
H2 : r = s
H3 : v = n'
============================
 n' = n'
```

This goal is again easily solved by `reflexivity`.

```
aev' : aeval ((x, v) :: s) (avar x) n'
r : state
x0 : string
y : string
v0 : Z
v' : Z
dif2 : x <> x
H4 : aeval s (avar x) n'
H : y = x
H1 : v0 = v
H2 : r = s
H3 : x0 = x
H0 : v' = n'
============================
 v = n'
```

In this goal, which corresponds to the fact that the statement

```
aeval ((x, v)::s) (avar x) n'
```

could have been proved by the second constructor, there is an absurd hypothesis. This hypothesis is named `dif2`, thanks to the efforts we made to specify this name when calling the inversion tactic. Discarding this goal is done easily by using the fact that `x <> x` is an abbreviation for `not (x = x)` and we can indeed prove `x = x`. This is done in the following way:

```
case dif2; reflexivity.
```

Third case. The third goal generated by the induction tactic corresponds to the case where `aeval s e n` was proved using the third constructor, `ae_var2`. We already discussed how the naming scheme for the induction tactic was directed to choose the name of the various hypotheses. Let us now observe the goal we obtain.

```
...
dif : x <> y
A : aeval s (avar x) v'
Ia : forall n' : Z, aeval s (avar x) n' -> v' = n'
============================
 forall n' : Z, aeval ((y, v) :: s) (avar x) n' -> v' = n'
```

We can again introduce the variable and the hypothesis of statement `aeval ...` and call the tactic `inversion` on this hypothesis.

```
intros n' aev'; inversion aev'.
```

This call to `inversion` generates two goals. The first one corresponds to the case where `aeval ((y, v)::s) (avar x) n'` would be proved by the first constructor. But in this case, we would need `x = y` to hold, and this is incompatible with the hypothesis `dif`. The goal has the following shape:

```
dif : x <> y
A : aeval s (avar x) v'
Ia : forall n' : Z, aeval s (avar x) n' -> v' = n'
n' : Z
aev' : aeval ((y, v) :: s) (avar x) n'
...
H : y = x
...
============================
 v' = n'
```

We solve this goal with the following tactic.

```
case dif; solve[auto].
```

The second goal generated by the inversion step has the following shape:

```
...
Ia : forall n' : Z, aeval s (avar x) n' -> v' = n'
...
H5 : aeval s (avar x) n'
...
============================
 v' = n'
```

This goal corresponds to the case where `aeval ((y, v)::s) (avar x) n'` could be proved by the third constructor. In this case, the premise `aeval s (avar x) n'` must have been proved, and all the conditions to use the induction hypothesis are present in the context, so that we can conclude with the following tactic.

```
apply Ia; assumption.
```

Fourth case. The last goal generated by the `induction` tactic corresponds to the fact that `aeval s e n` could have been proved using the fourth constructor, `ae_plus`. In this case, `e` must have the shape `(plus e1 e2)` and two hypotheses `aeval s e1 n1` and `aeval s e2 n2` must hold. The names of various components and hypotheses are fixed by the names given between square brackets when the `induction` tactic was called. Moreover, there are two induction hypotheses corresponding to the two hypotheses that concern the `aeval` predicate. Let us observe the goal we obtain:

```
  s : state
  e1 : aexpr
  e2 : aexpr
  v1 : Z
  v2 : Z
  A1 : aeval s e1 v1
  A2 : aeval s e2 v2
  I1 : forall n' : Z, aeval s e1 n' -> v1 = n'
  I2 : forall n' : Z, aeval s e2 n' -> v2 = n'
  ============================
   forall n' : Z, aeval s (aplus e1 e2) n' -> v1 + v2 = n'
```

In this case, we want to use the inversion tactic to show that `aeval s (aplus e1 e2)` can only be proved using the fourth constructor, and therefore there exist other sub-derivations returning values `v'1` and `v'2` such that `v'1 + v'2 = n'`. This is done in the following way:

```
intros n' aev'; inversion aev' as [ | | | ? ? ? v1' v2'].
```

This generates the following goal:

```
  ...
  A1 : aeval s e1 v1
  A2 : aeval s e2 v2
  I1 : forall n' : Z, aeval s e1 n' -> v1 = n'
  I2 : forall n' : Z, aeval s e2 n' -> v2 = n'
  n' : Z
  aev' : aeval s (aplus e1 e2) n'
  ...
  H2 : aeval s e1 v1'
  H4 : aeval s e2 v2'
```

```
  H1 : s' = s
  H : e1' = e1
  H0 : e2' = e2
  H3 : v1' + v2' = n'
  ============================
   v1 + v2 = v1' + v2'
```

At this point, we wish to establish that `v1` and `v1'` are equal (and so are `v2` and `v2'`) using induction hypotheses. We can perform these steps using the following tactics, because the `auto` tactic knows how to use the inductions hypotheses, which are in the context.

```
assert (v1 = v1' /\ v2 = v2') as [vv1 vv2] by auto.
rewrite vv1, vv2; reflexivity.
```

This finishes the proof and we can now save it for later use.

```
Qed.
```

After a few steps of proof engineering, the same proof can be made even shorter, as illustrated here:

```
Lemma eval_deterministic2 :
  forall s e n n', aeval s e n -> aeval s e n' -> n = n'.
Proof.
intros s e n n' aev; revert n';
 induction aev as [| | |? ? ? v1 v2];
 intros n' A; inversion A as [| | | ? ? ? v1' v2']; auto;
 try match goal with i : _ <> _ |- _ => case i; now auto end.
(assert (v1 = v1' /\ v2 = v2') as [U V] by auto); subst; auto.
Qed.
```

In the same spirit, we can prove that `update`, `beval`, and `exec` are deterministic.