Logic
(Métodos Formais em Engenharia de Software)

Maria João Frade

Departmento de Informática
Universidade do Minho

2011/2012

Roadmap

- **Inductive Definitions**
  - inductive types and its elimination mechanisms
  - proof by induction; case analysis; general recursion
  - relations as inductive types; logical connectives as inductive types
  - some datatypes of programming
- **Case Study: Programmig Language Semantics**
  - encoding of a very simple imperative programming language
  - natural semantic; axiomatic semantics
  - correctness of axiomatic semantics w.r.t. the natural semantics

Bibliography

- *Coq in a Hurry*. Yves Bertot. February 2010.
- [Bertot&Castéran 2004] *Interactive Theorem Proving and Program Development Coq'Art: The Calculus of Inductive Constructions.* Yves Bertot & Pierre Castéran. Springer, 2004.
- [Bertot 2009] *Theorem-proving support in programming language semantics.* From Semantics to Computer Science, essays in Honour of Gilles Kahn. Cambridge University Press, 2009.

Inductive Definitions

Induction

Induction is a basic notion in logic and set theory.

- When a set is defined inductively we understand it as being "built up from the bottom" by a set of basic constructors.
- Elements of such a set can be decomposed in "smaller elements" in a well-founded manner.
- This gives us principles of
  - *"proof by induction"* and
  - *"function definition by recursion"*.

Inductive types

We can define a new type I inductively by giving its *constructors* together with their types which must be of the form

$$\boxed{\tau_1 \to \ldots \to \tau_n \to I \quad \text{, with} \quad n \geq 0}$$

- Constructors (which are the *introduction rules* of the type I) give the canonical ways of constructing one element of the new type I.
- The type I defined is the smallest set (of objects) closed under its introduction rules.
- The inhabitants of type I are the objects that can be obtained by a finite number of applications of the type constructors.

Type I (under definition) can occur in any of the "domains" of its constructors. However, the occurrences of I in τ_i must be in *positive positions* in order to assure the well-foundedness of the datatype.

For instance, assuming that I does not occur in types A and B: $I \to B \to I$, $A \to (B \to I) \to I$ or $((I \to A) \to B) \to A \to I$ are valid types for a constructor of I, but $(I \to A) \to I$ or $((A \to I) \to B) \to A \to I$ are not.

Induction types - examples

- The inductive type $\mathbb{N}$: Set of natural numbers has two constructors

$$\begin{array}{l} 0 : \mathbb{N} \\ \mathsf{S} : \mathbb{N} \to \mathbb{N} \end{array}$$

- A well-known example of a higher-order datatype is the type $\mathbb{O}$: Set of ordinal notations which has three constructors

$$\begin{array}{lcl} \mathsf{Zero} & : & \mathbb{O} \\ \mathsf{Succ} & : & \mathbb{O} \to \mathbb{O} \\ \mathsf{Lim} & : & (\mathbb{N} \to \mathbb{O}) \to \mathbb{O} \end{array}$$

To program and reason about an inductive type we must have means to analyze its inhabitants.

The *elimination rules* for the inductive types express ways to use the objects of the inductive type in order to define objects of other types, and are associated to new computational rules.

Recursors

When an inductive type is defined in a type theory the theory should automatically generate a scheme for proof-by-induction and a scheme for primitive recursion.

- The inductive type comes equipped with a *recursor* that can be used to define functions and prove properties on that type.
- The recursor is a constant $\mathbf{R}_I$ that represents the structural induction principle for the elements of the inductive type I, and the computation rule associated to it defines a safe recursive scheme for programming.

For example, $\mathbf{R}_\mathbb{N}$, the recursor for $\mathbb{N}$, has the following typing rule:

$$\frac{\Gamma \vdash P : \mathbb{N} \to \mathsf{Type} \quad \Gamma \vdash a : P\,0 \quad \Gamma \vdash a' : \Pi x{:}\mathbb{N}.\, P\,x \to P\,(\mathsf{S}\,x)}{\Gamma \vdash \mathbf{R}_\mathbb{N}\,P\,a\,a' : \Pi n{:}\mathbb{N}.\, P\,n}$$

and its reduction rules are

$$\begin{array}{lcl} \mathbf{R}_\mathbb{N}\,P\,a\,a'\,0 & \to & a \\ \mathbf{R}_\mathbb{N}\,P\,a\,a'\,(\mathsf{S}\,x) & \to & a'\,x\,(\mathbf{R}_\mathbb{N}\,P\,a\,a'\,x) \end{array}$$

Proof-by-induction scheme

The proof-by-induction scheme can be recovered by setting P to be of type $\mathbb{N}\rightarrow\mathsf{Prop}$.

Let $\mathsf{ind}_{\mathbb{N}} \;:=\; \lambda\, P{:}\mathbb{N}\rightarrow\mathsf{Prop}.\, \mathbf{R}_{\mathbb{N}}\, P$ we obtain the following rule

$$\frac{\Gamma \vdash P:\mathbb{N}\rightarrow\mathsf{Prop} \quad \Gamma \vdash a : P\,0 \quad \Gamma \vdash a' : \Pi\, x{:}\mathbb{N}.\, P\, x \rightarrow P\,(\mathsf{S}\, x)}{\Gamma \vdash \mathsf{ind}_{\mathbb{N}}\, P\, a\, a' : \Pi\, n{:}\mathbb{N}.\, P\, n}$$

This is the well known structural induction principle over natural numbers. It allows to prove some universal property of natural numbers $(\forall n{:}\mathbb{N}.\, Pn)$ by induction on n.

Primitive recursion scheme

The primitive recursion scheme (allowing dependent types) can be recovered by setting $P:\mathbb{N}\rightarrow\mathsf{Set}$.

Let $\mathsf{rec}_{\mathbb{N}} \;:=\; \lambda\, P{:}\mathbb{N}\rightarrow\mathsf{Set}.\, \mathbf{R}_{\mathbb{N}}\, P$ we obtain the following rule

$$\frac{\Gamma \vdash T:\mathbb{N}\rightarrow\mathsf{Set} \quad \Gamma \vdash a : T\,0 \quad \Gamma \vdash a' : \Pi\, x{:}\mathbb{N}.\, T\, x \rightarrow T\,(\mathsf{S}\, x)}{\Gamma \vdash \mathsf{rec}_{\mathbb{N}}\, T\, a\, a' : \Pi\, n{:}\mathbb{N}.\, T\, n}$$

We can define functions using the recursors.

For instance, a function that doubles a natural number can be defined as follows:

$$\mathsf{double} \;:=\; \mathsf{rec}_{\mathbb{N}}\,(\lambda n{:}\mathbb{N}.\,\mathbb{N})\,0\,(\lambda x{:}\mathbb{N}.\,\lambda y{:}\mathbb{N}.\,\mathsf{S}\,(\mathsf{S}\,y))$$

This approach gives safe way to express recursion without introducing non-normalizable objects.

However, codifying recursive functions in terms of elimination constants is quite far from the way we are used to program. Instead we usually use general recursion and case analysis.

Case analysis

Case analyses gives an elimination rule for inductive types.

For instance, $n:\mathbb{N}$ means that n was introduced using either 0 or S, so we may define an object case n of $\{0 \Rightarrow b_1 \mid \mathsf{S} \Rightarrow b_2\}$ in another type σ depending on which constructor was used to introduce n.

A typing rule for this construction is

$$\frac{\Gamma \vdash n:\mathbb{N} \quad \Gamma \vdash b_1:\sigma \quad \Gamma \vdash b_2:\mathbb{N}\rightarrow\sigma}{\Gamma \vdash \mathsf{case}\; n\; \mathsf{of}\; \{0 \Rightarrow b_1 \mid \mathsf{S} \Rightarrow b_2\} : \sigma}$$

and the associated computation rules are

$$\begin{array}{lcl} \mathsf{case}\; 0\; \mathsf{of}\; \{0 \Rightarrow b_1 \mid \mathsf{S} \Rightarrow b_2\} & \rightarrow & b_1 \\ \mathsf{case}\; (\mathsf{S}\,x)\; \mathsf{of}\; \{0 \Rightarrow b_1 \mid \mathsf{S} \Rightarrow b_2\} & \rightarrow & b_2\, x \end{array}$$

The case analysis rule is very useful but it does not give a mechanism to define recursive functions.

General recursion

Functional programming languages feature *general recursion*, allowing recursive functions to be defined by means of pattern-matching and a general fixpoint operator to encode recursive calls.

The typing rule for $\mathbb{N}$ fixpoint expressions is

$$\frac{\Gamma \vdash \mathbb{N}\rightarrow\theta : s \quad \Gamma, f:\mathbb{N}\rightarrow\theta \vdash e:\mathbb{N}\rightarrow\theta}{\Gamma \vdash (\mathsf{fix}\; f = e):\mathbb{N}\rightarrow\theta}$$

and the associated computation rules are

$$\begin{array}{lcl} (\mathsf{fix}\; f = e)\, 0 & \rightarrow & e[(\mathsf{fix}\; f = e)/f]\, 0 \\ (\mathsf{fix}\; f = e)\, (\mathsf{S}\,x) & \rightarrow & e[(\mathsf{fix}\; f = e)/f]\, (\mathsf{S}\,x) \end{array}$$

Of course, this approach opens the door to the introduction of non-normalizable objects.

Using this, the function that doubles a natural number can be defined by

$$(\mathsf{fix}\; \mathsf{double} = \lambda n:\mathbb{N}.\, \mathsf{case}\; n\; \mathsf{of}\; \{0 \Rightarrow 0 \mid \mathsf{S} \Rightarrow (\lambda x:\mathbb{N}.\, \mathsf{S}\,(\mathsf{S}\,(\mathsf{double}\, x)))\})$$

About termination

- Checking convertibility between types may require computing with recursive functions. So, the combination of non-normalization with dependent types leads to undecidable type checking.
- To enforce decidability of type checking, proof assistants either require recursive functions to be encoded in terms of recursors or allow restricted forms of fixpoint expressions.
- A usual way to ensure termination of fixpoint expressions is to impose syntactical restrictions through a predicate $\mathcal{G}_f$ on untyped terms. This predicate enforces termination by constraining all recursive calls to be applied to terms structurally smaller than the formal argument of the function.

The restricted typing rule for fixpoint expressions hence becomes:

$$\frac{\Gamma \vdash \mathbb{N}\to\theta : s \qquad \Gamma, f:\mathbb{N}\to\theta \vdash e:\mathbb{N}\to\theta}{\Gamma \vdash (\text{fix } f = e):\mathbb{N}\to\theta} \quad \text{if } \mathcal{G}_f(e)$$

Computation

Recall that typing judgments in Coq are of the form $E\,|\,\Gamma \vdash M : A$, where E is the global environment and Γ is the local context.

Computations are performed as series of *reductions*.

β-reduction for compute the value of a function for an argument:

$$(\lambda x\!:\!A.\,M)\,N \quad \to_\beta \quad M[N/x]$$

δ-reduction for unfolding definitions:

$$M \quad \to_\delta \quad N \qquad \text{if } (M := N) \in E\,|\,\Gamma$$

ι-reduction for primitive recursion rules, general recursion and case analysis

ζ-reduction for local definitions: `let` $x := N$ `in` $M \quad \to_\zeta \quad M[N/x]$

Note that the conversion rule is

$$\frac{E\,|\,\Gamma \vdash M : A \qquad E\,|\,\Gamma \vdash B : s}{E\,|\,\Gamma \vdash M : B} \quad \text{if } A =_{\beta\iota\delta\zeta} B \text{ and } s \in \{\text{Prop}, \text{Set}, \text{Type}\}$$

Natural numbers

```
Inductive nat :Set :=   O : nat
                      | S : nat -> nat.
```

The declaration of this inductive type introduces in the global environment not only the constructors O and S but also the recursors: nat_rect, nat_ind and nat_rec

```
Check nat_rect.
nat_rect
     : forall P : nat -> Type,
       P 0 -> (forall n : nat, P n -> P (S n)) -> forall n : nat, P n
```

```
Print nat_ind.
nat_ind = fun P : nat -> Prop => nat_rect P
     : forall P : nat -> Prop,
       P 0 -> (forall n : nat, P n -> P (S n)) -> forall n : nat, P n
```

```
Print nat_rec.
nat_rec = fun P : nat -> Set => nat_rect P
     : forall P : nat -> Set,
       P 0 -> (forall n : nat, P n -> P (S n)) -> forall n : nat, P n
```

Vectors of length n over A.

```
Inductive vector (A : Type) : nat -> Type :=
  | Vnil : vector A 0
  | Vcons : A -> forall n : nat, vector A n -> vector A (S n).
```

Remark the difference between the two parameters A and n:
- A is a general parameter, global to all the introduction rules,
- n is an index, which is instantiated differently in the introduction rules.

The type of constructor Vcons is a dependent function.

```
Variables b1 b2 :  B.
Check (Vcons _ b1 _ (Vcons _ b2 _ (Vnil _))).
Vcons B b1 1 (Vcons B b2 0 (Vnil B))  : vector B 2
```

```
Check vector_rect.
vector_rect
     : forall (A : Type) (P : forall n : nat, vector A n -> Type),
       P 0 (Vnil A) ->
       (forall (a : A) (n : nat) (v : vector A n),
        P n v -> P (S n) (Vcons A a n v)) ->
       forall (n : nat) (v : vector A n), P n v
```

Equality

In Coq, the propositional equality between two inhabitants a and b of the same type A, noted $a = b$, is introduced as a family of recursive predicates "to be equal to a", parameterized by both a and its type A. This family of types has only one introduction rule, which corresponds to reflexivity.

```
Inductive eq (A : Type) (x : A) : A -> Prop :=
  | refl_equal : (eq A x x).
```

The induction principle of eq is very close to the Leibniz's equality but not exactly the same.

```
Check eq_ind.
```

```
eq_ind : forall (A : Type) (x : A) (P : A -> Prop),
          P x -> forall y : A, x = y -> P y
```

Notice that the syntax "`a = b`" is an abbreviation for "`eq a b`", and that the parameter A is implicit, as it can be inferred from a.

```
Inductive eq (A : Type) (x : A) : A -> Prop :=
  | refl_equal : x = x.
```

Relations as inductive types

Some relations can also be introduced as an inductive family of propositions. For instance, the order $n \leq m$ on natural numbers is defined as follows in the standard library:

```
Inductive le (n:nat) : nat -> Prop :=
  | le_n : (le n n)
  | le_S : forall m : nat, (le n m) -> (le n (S m)).
```

- Notice that in this definition `n` is a general parameter, while the second argument of `le` is an index. This definition introduces the binary relation $n \leq m$ as the family of unary predicates "to be greater or equal than a given n", parameterized by n.
- The Coq system provides a syntactic convention, so that "`le x y`" can be written "`x <= y`".
- The introduction rules of this type can be seen as rules for proving that a given integer n is less or equal than another one. In fact, an object of type $n \leq m$ is nothing but a proof built up using the constructors `le_n` and `le_S`.

Logical connectives in Coq

In the Coq system, most logical connectives are represented as inductive types, except for $\Rightarrow$ and $\forall$ which are directly represented by $\rightarrow$ and Π-types, negation which is defined as the implication of the absurd and equivalence which is defined as the conjunction of two implications.

```
Definition not := fun A : Prop => A -> False.

Notation "~ A" := (not A) (at level 75, right associativity).
```

```
Inductive True : Prop :=  I : True.
```

```
Inductive False : Prop := .
```

```
Inductive and (A : Prop) (B : Prop) : Prop :=
  | conj : A -> B -> (and A B).

Notation "A /\ B" := (and A B) (at level 80, right associativity).
```

Logical connectives in Coq

```
Inductive or (A : Prop) (B : Prop) : Prop :=
  | or_introl : A -> (or A B)
  | or_intror : B -> (or A B).

Notation "A \/ B" := (or A B) (at level 85, right associativity).
```

```
Inductive ex (A : Type) (P : A -> Prop) : Prop :=
  | ex_intro : forall x : A, P x -> ex P.

exists x:A, P  is an abbreviation of  ex A (fun x:A => P).
```

```
Definition iff (P Q:Prop) := (P -> Q) /\ (Q -> P).

Notation "P <-> Q" := (iff P Q) (at level 95, no associativity).
```

The constructors are the introduction rules.
The induction principle gives the elimination rules.
All the (constructive) logical rules are now derivable.

Some datatypes of programming

```
Inductive unit : Set :=  tt : unit.

Inductive bool : Set :=  true : bool | false : bool.

Inductive nat : Set :=  O : nat | S : nat -> nat.

Inductive option (A : Type) : Type :=  Some : A -> option A
                                     | None : option A.

Inductive identity (A : Type) (a : A) : A -> Type :=
    refl_identity : identity A a a.
```

Some operations on `bool` are also provided: `andb` (with infix notation &&), `orb` (with infix notation ||), `xorb`, `implb` and `negb`.

Some datatypes of programming

```
Inductive sum (A B : Type) : Type :=  inl : A -> A + B
                                    | inr : B -> A + B.

Inductive prod (A B : Type) : Type :=  pair : A -> B -> A * B.

Definition fst (A B : Type) (p : A * B) := let (x, _) := p in x.

Definition snd (A B : Type) (p : A * B) := let (_, y) := p in y.
```

The constructive sum `{A}+{B}` of two propositions `A` and `B`.

```
Inductive sumbool (A B : Prop) : Set :=
  | left : A -> {A} + {B}
  | right : B -> {A} + {B}.
```

If-then-else

- The `sumbool` type can be used to define an "if-then-else" construct in Coq.
- Coq accepts the syntax `if` *test* `then` ... `else` ... when *test* has either of type `bool` or `{A}+{B}`, with propositions `A` and `B`.
- Its meaning is the pattern-matching

```
match test with
 | left H => ...
 | right H => ...
end.
```

- We can identify `{P}+{~P}` as the type of decidable predicates:

The standard library defines many useful predicates, e.g.

```
le_lt_dec : forall n m : nat, {n <= m} + {m < n}
Z_eq_dec : forall x y : Z, {x = y} + {x <> y}
Z_lt_ge_dec : forall x y : Z, {x < y} + {x >= y}
```

If-then-else

A function that checks if an element is in a list.

```
Fixpoint elem (a:Z) (l:list Z) {struct l} : bool :=
    match l with
      | nil => false
      | cons x xs => if Z_eq_dec x a then true else (elem a xs)
    end.
```

Exercises

Demo

Load the file IndDefs.v in the Coq proof assistant. Analyse the examples and solve the exercises proposed.

Case Study: Programming Language Semantics

Introduction

- A typical application of Coq is the formalization of programming languages semantics. E.g.
  - The CompCert compiler certification project (INRIA);
  - The Concurrent C Minor Project (Princeton);
  - The Ynot project (Harvard);
  - Java Card EAL7 certification (in industrial context).
- The following paper illustrates the formalization, in Coq, of many aspects of programming language semantics.

Yves Bertot. Theorem-proving support in programming language semantics. *From Semantics to Computer Science, essays in Honour of Gilles Kahn*. Cambridge University Press, 2009.

We will see a fragment of this work.

Semantics of programming languages

The meaning of a grammatically correct program can be formalized in different ways:

- **Operational semantics** is focused on the computation the program induces on a machine (*small-step*, or *structural*, if the emphasis is on the individual steps of the execution; *big-step*, or *natural semantics*, if the emphasis is on the relationship between the initial and the final state of the execution).
- **Denotational semantics** is focused on representing the effect of executing a program by a mathematical object.
- **Axiomatic semantics** is focused on specific properties (expressed by assertions) of the effect of executing a program.

The paper

- Describes several views of the semantics of a simple programming language.
- Covered aspects are: operational semantics (big-step and small-step), denotational semantics, axiomatic semantics, and abstract interpretation.
- Descriptions as recursive functions are also provided whenever suitable, thus yielding a verification condition generator and a static analyser that can be run inside the theorem prover.
- Extraction of an interpreter from the denotational semantics is also described.
- All different aspects are formally proved sound with respect to the natural semantics specification.

We will just focus on the encoding of the programming language, its natural semantics and axiomatics semantics, and the correctness of axiomatic semantics with respect to the natural semantics.

Encoding

One can decribe programming languages relying mostly on the basic concepts of inductive types and inductive propositions.

- Programs can be represented as an inductive datatype, following the tradition of abstract syntax trees.
- Operational semantics: the execution of instructions can be described as inductive propositions.
- Axiomatic semantics: Hoare triples can be described as inductive propositions.

The programming language

- We consider a *while loop* programing language with simple arithmetic expressions.
- The language has been trimmed to a bare skeleton, but still retains the property of being Turing complete.
- Execution states are represented as *environments*, encoded by lists of pairs binding a variable name and a value.
- We will use:

  ρ as meta-variables for variable declarations (environment),
  e for expressions,
  b for boolean expressions,
  i for instructions,
  x, y, x_1 for variable names,
  n, n_1, n' to represent integers.

The programming language

The syntactic categories are defined as follows:

$$\rho ::= (x, n) \cdot \rho | \emptyset \qquad e ::= n \mid x \mid e{+}e \qquad b ::= e < e$$

$$i ::= \mathsf{skip} \mid x{:=}e \mid i;i \mid \mathsf{while}\ b\ \mathsf{do}\ i\ \mathsf{done}$$

The programming language

In the theorem prover, we use inductive types to describe these syntactic categories.

```
Inductive aexpr : Type :=
    avar (s : string) | anum (n : 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).
```

Evaluation of expressions

Evaluation of expressions is decribed using judgments of the form $\rho \vdash e \rightarrow v$ or $\rho \vdash b \rightarrow v$. The value v is an integer or a boolean value depending on the kind of expression being evaluated.
Evaluation is decribed by the following rules:

$$\frac{}{\rho \vdash n \rightarrow n} \qquad \frac{}{(x,n)\cdot\rho \vdash x \rightarrow n}$$

$$\frac{\rho \vdash x \rightarrow n \quad x \neq y}{(y,n')\cdot\rho \vdash x \rightarrow n} \qquad \frac{\rho \vdash e_1 \rightarrow n_1 \quad \rho \vdash e_2 \rightarrow n_2}{\rho \vdash e_1{+}e_2 \rightarrow n_1 + n_2}$$

$$\frac{\rho \vdash e_1 \rightarrow n_1 \quad \rho \vdash e_2 \rightarrow n_2 \quad n_1 < n_2}{\rho \vdash e_1{<}e_2 \rightarrow \mathsf{true}} \qquad \frac{\rho \vdash e_1 \rightarrow n_1 \quad \rho \vdash e_2 \rightarrow n_2 \quad n_2 \leq n_1}{\rho \vdash e_1{<}e_2 \rightarrow \mathsf{false}}$$

Evaluation of expressions

Judgments of the form $\rho \vdash e \rightarrow v$ are represented by the three-argument inductive predicate `aeval`.
The encoding of premises is quite straight forward using nested implications, and universal quantifications are added for every variable that occurs in the inference rules.

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

Inductive aeval : env -> aexpr -> Z -> Prop :=
  ae_int : 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).
```

Evaluation of expressions

Judgments of the form $\rho \vdash b \rightarrow v$ are represented by the three-argument inductive predicate `beval`.

```
Inductive beval : env -> 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.
```

Environment update

During the execution of instructions, it is necessary to describe the modification of an environment, so that the value associated to a variable is modified. The judgments $\rho \vdash x, n \mapsto \rho'$ state the environment update, which can be described by the following inference rules:

$$\frac{}{(x,v)\cdot\rho \vdash x, v' \mapsto (x,v')\cdot\rho}$$

$$\frac{\rho \vdash x, v' \mapsto \rho' \quad x \neq y}{(y,v)\cdot\rho \vdash x, v' \mapsto (y,v)\cdot\rho'}$$

These rules are also encoded as an inductive definition for a predicate named `s_update`.

```
Inductive s_update : env -> string -> Z -> env -> 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').
```

Natural semantics

Natural semantics (introduced by *Gilles Kahn*), also called big-step semantics, describes how the overall results of the executions are obtained. Judgments $\rho \vdash i \rightsquigarrow \rho'$ should be read as *"executing i from the initial environment ρ terminates and yields the new environment ρ'"*.

$$\frac{}{\rho \vdash \textsf{skip} \rightsquigarrow \rho} \qquad \frac{\rho \vdash e \to n \qquad \rho \vdash x, n \mapsto \rho'}{\rho \vdash x{:=}e \rightsquigarrow \rho'}$$

$$\frac{\rho \vdash i_1 \rightsquigarrow \rho' \qquad \rho' \vdash i_2 \rightsquigarrow \rho''}{\rho \vdash i_1;i_2 \rightsquigarrow \rho''} \qquad \frac{\rho \vdash b \to \textsf{false}}{\rho \vdash \textsf{while}\ b\ \textsf{do}\ i\ \textsf{done} \rightsquigarrow \rho}$$

$$\frac{\rho \vdash b \to \textsf{true} \quad \rho \vdash i \rightsquigarrow \rho' \quad \rho' \vdash \textsf{while}\ b\ \textsf{do}\ i\ \textsf{done} \rightsquigarrow \rho''}{\rho \vdash \textsf{while}\ b\ \textsf{do}\ i\ \textsf{done} \rightsquigarrow \rho''}$$

Natural semantics

The judgment $\rho \vdash i \rightsquigarrow \rho'$ can be described with an inductive predicate `exec`.

```
Inductive exec : env -> instr -> env -> Prop :=
| SN1 : forall r, exec r skip r
| SN2 : forall r r' x e v,
          aeval r e v -> s_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.
```

Axiomatic semantics

Axiomatic semantics (proposed by *Tony Hoare*) defines the meaning of a program by describing its effect on assertions about the program state (the environment).

Judgments $\{P\}\,i\,\{Q\}$ state that *"if P is satisfied before executing i and executing i terminates, then Q is guaranteed to be satisfied after executing i"*.

$$\frac{}{\{P\}\textsf{skip}\{P\}} \qquad \frac{\{P\}i_1\{Q\} \quad \{Q\}i_2\{R\}}{\{P\}i_1;i_2\{R\}}$$

$$\frac{}{\{P[x \leftarrow e]\}x{:=}e\{P\}} \qquad \frac{\{b \wedge P\}i\{P\}}{\{P\}\textsf{while}\ b\ \textsf{do}\ i\ \textsf{done}\{\neg b \wedge P\}}$$

$$\frac{P \Rightarrow P_1 \quad \{P_1\}i\{Q_1\} \quad Q_1 \Rightarrow Q}{\{P\}i\{Q\}}$$

Axiomatic semantics

- Relates properties instead of relating states.
- Relates logical reasoning with program constructs.
- Requires a syntax for the assertions.
- The behavior of assignment is explained using substitution.
- The formal system describing the semantics is usualy called *Hoare Logic*.

Assertions and conditions

The language of assertions contains conjunctions, negations, and tests from the language's boolean expressions. It is also included the possibility to have arbitrary predicates on arithmetic expressions, represented by a name given as a string.

```
Inductive assert : Type :=
  a_b (b : bexpr) | a_not (a : assert) | a_conj (a a' : assert)
| pred(s : string)(l : list aexpr).
```

The consequence rule, makes it possible to mix logical reasoning about the properties with the symbolic reasoning about the program constructs. To prove the two premises that are implications, it is necessary to master the actual meaning of the properties, conjunction, and negation.

```
Inductive condition : Type :=
| c_imp : assert -> assert -> condition.
```

Substitution

We define substitution for assertions. The function is called a_subst, and uses substitution for arithmetic expressions, boolean expressions, and so on.

```
Fixpoint a_subst (a:assert)(s:string)(v:aexpr)
 {struct a} : assert :=
  match a with
    a_b e => a_b (b_subst e s v)
  | a_not a => a_not (a_subst a s v)
  | pred p l => pred p (l_subst l s v)
  | a_conj a1 a2 => a_conj(a_subst a1 s v)(a_subst a2 s v)
  end.
```

Substitution

We also define substitution for arithmetic expressions subst, boolean expressions b_subst, and lists of arithmetic exepressions l_subst.

```
Fixpoint subst (e:aexpr)(s:string)(v:aexpr) {struct e} : aexpr :=
  match e with
    avar s' => if string_dec s s' then v else e
  | anum n => anum n
  | aplus e1 e2 => aplus (subst e1 s v) (subst e2 s v)
  end.

Definition b_subst (b:bexpr) (s:string) (v:aexpr) : bexpr :=
  match b with blt e1 e2 => blt (subst e1 s v)(subst e2 s v) end.

Fixpoint l_subst (l:list aexpr)(s:string)(v:aexpr)
 {struct l} : list aexpr :=
  match l with
    nil => nil
  | a::tl => subst a s v::l_subst tl s v
  end.
```

Interpretation functions

For variables that occur inside arithmetic expressions, we use valuation functions of type `string->Z` instead of environments and we define a function `af'` (resp. `bf'`, `lf'`) to compute the value of an arithmetic expression (resp. boolean expressions, lists of arithmetic expressions) for a given valuation.

```
Fixpoint af' (g : string -> Z)(a : aexpr) {struct a} : Z :=
  match a with
    avar s => g s
  | anum n => n
  | aplus e1 e2 => af' g e1 + af' g e2
  end.
```

Interpretation functions

To give a meaning to predicates, we use lists of pairs associating names and predicates on lists of integers as *predicate* environments and we have a function f_p to map an environment and a string to a predicate on integers.

```
Definition p_env := list (string * (list Z -> Prop)).

Fixpoint f_p (preds : p_env) (s:string) (args:list Z)
 {struct preds} : Prop :=
  match preds with
    (a,m)::tl => if string_dec a s then m args else f_p tl s args
  | nil => True
  end.
```

Interpretation functions

With all these functions, we can interpret assertions as propositional values using a function i_a and conditions using a function i_c.

```
Fixpoint i_a (preds : p_env) (g : string->Z) (a : assert)
 {struct a} : Prop :=
  match a with
    a_b e => bf' g e
  | a_not a => ~ i_a preds g a
  | pred p l => f_p preds p (lf' g l)
  | a_conj a1 a2 => i_a preds g a1 /\ i_a preds g a2
  end.

Definition i_c (m : p_env) (g : string->Z) (c : condition) :=
  match c with
    c_imp a1 a2 => i_a m g a1 -> i_a m g a2
  end.
```

Interpretation functions

The validity of conditions can be expressed for a given predicate environment by saying that their interpretation should hold for any valuation.

```
Definition valid (preds: p_env) (c:condition) :=
  forall g, i_c preds g c.
```

Encoding of Hoare Logic

We can then define the axiomatic semantics.

```
Inductive ax_sem (preds : p_env) :
      assert -> instr -> assert -> Prop :=
  ax1 : forall P, ax_sem preds P skip P
| ax2 : forall P x e, ax_sem preds (a_subst P x e) (assign x e) P
| ax3 : forall P Q R i1 i2,
          ax_sem preds P i1 Q -> ax_sem preds Q i2 R ->
          ax_sem preds P (sequence i1 i2) R
| ax4 : forall P b i,
          ax_sem preds (a_conj (a_b b) P) i P ->
          ax_sem preds P (while b i) (a_conj (a_not (a_b b)) P)
| ax5 : forall P P' Q' Q i,
          valid preds (c_imp P P') -> ax_sem preds P' i Q' ->
          valid preds (c_imp Q' Q) ->
          ax_sem preds P i Q.
```

Proving the correctness

We want to certify that the properties of programs that we can prove using axiomatic semantics hold for actual executions of programs, as described by the natural semantics.
We first define a mapping from the environments used in operational semantics to the valuations used in the axiomatic semantics. This mapping is called e_to_f, the expression (e_to_f e g x) is the value of x in the environment e, when it is defined, and g x otherwise. The formula (e_to_f e g) is also written e@g.

```
Fixpoint e_to_f(r:list(string*Z))(g:string->Z)(var:string):Z :=
  match r with
    nil => g var
  | (s,v)::tl =>
    if string_dec s var then v else e_to_f tl g var
  end.

Notation "r @ g" := (e_to_f r g) (at level 30, right associat...
```

Proving the correctness

The correctness of axiomatic semantics is expressed by stating that if (exec r i r') and (ax sem P i Q) hold, if P holds in the initial environment, Q should hold in the final environment Q.

```
Theorem ax_sem_sound :
  forall m r i r' g P Q, exec r i r' -> ax_sem m P i Q ->
    i_a m (r@g) P -> i_a m (r'@g) Q.
```

Proving the correctness

- When we attempt to prove this statement by induction on exec and case analyis on ax_sem, we encounter problem because uses of consequence rules may make axiomatic semantics derivations arbitrary large.
- To reduce this problem we introduce a notion of *normalized* derivations where exactly one consequence step is associated to every structural step.
- We introduce an extra inductive predicate call nax to model these normalized derivation.
- nax is an encoding of a *goal-directed* system for Hoare Logic.

Proving the correctness

```
Inductive nax (preds: p_env) :
       assert -> instr -> assert -> Prop :=
  nax1 : forall P Q,
     valid preds (c_imp P Q) -> nax preds P skip Q
| nax2 : forall P P' Q x e,
    valid preds (c_imp P (a_subst P' x e)) ->
    valid preds (c_imp P' Q) ->
    nax preds P (assign x e) Q
| nax3 : forall P P' Q' R' R i1 i2,
    valid preds (c_imp P P') ->
    valid preds (c_imp R' R) ->
    nax preds P' i1 Q' -> nax preds Q' i2 R' ->
    nax preds P (sequence i1 i2) R
| nax4 : forall P P' Q b i,
    valid preds (c_imp P P') ->
    valid preds (c_imp (a_conj (a_not (a_b b)) P') Q) ->
    nax preds (a_conj (a_b b) P') i P' ->
    nax preds P (while b i) Q.
```

Proving the correctness

ax_sem and nax are proved equivalent.

```
Lemma ax_sem_nax : forall m P i Q,
      ax_sem m P i Q -> nax m P i Q.

Lemma nax_ax_sem : forall m P i
      Q, nax m P i Q -> ax_sem m P i Q.
```

The correctness statement can now be proved by induction on exec and by cases on nax, while a proof by double induction would be required with ax_sem.

```
Theorem nax_sound :
  forall m r i r' g, exec r i r' ->
  forall P Q, nax m P i Q ->
  i_a m (r@g) P -> i_a m (r'@g) Q.
```

Further reading

Check the package Semantics in the Coq Users' Contributions.