Semantics of Programming Languages

Pietro Di Gianantonio

Università di Udine

18th June, 2009
A short introduction to semantics of programming:
- objectives,
- some approaches,
- some examples.

Quite basic:
- almost no previous knowledge is required
- try to use only simple notions avoiding complex mathematics.

An introduction the notions presented in the second part of the course (with some repetitions)
Semantics of Programming Languages

Aims to formally describe the behaviour of programs, programs constructors.

Useful:

- to describe and specify a programming languages without ambiguities
  - as standard for syntax,
  - fundamental for building compilers,
- to reasons on programs: to proof that a program satisfies some given requirements; that is correct.

Formal methods used in some approaches to software engineering: formal system development, to produce reliable software.
For example inside UML (unified modeling language)
Approaches to semantics

Different styles to describe the program behaviour:

- **Operational semantics.** A formal, simple machine used to describes the behaviour of the programs.
- **Structural operational semantics (SOS).** A set of rules describing the behaviour of programs.
- **Denotational semantics.** To represent the behaviour of programs through a mathematical object.
- **Axiomatics semantics:** the meaning of program is expressed in terms of preconditions and postconditions

They describe different aspects of program behaviour, have different purposes.
Several aspects in programming languages

- non termination,
- store,
- environment,
- non determinism,
- concurrency,
- higher order functions,
- exceptions,
- continuations.

If a programming language is enriched with new features, it is necessary to enrich or modify the semantics to deal with the new aspects.
A simple imperative language: IMP

\[ a ::= n \mid X \mid a_0 + a_1 \mid \ldots \]

\[ b ::= \text{true} \mid \text{false} \mid a_0 = a_1 \mid b_0 \text{ or } b_1 \mid \ldots \]

\[ c ::= \text{skip} \mid X ::= a \mid c_0; c_1 \mid \text{if } b \text{ then } c_0 \text{ else } c_1 \text{ fi } \mid \text{while } b \text{ do } c \text{ od} \]

A minimal Turing-complete language:

- no environment and aliasing,
- no procedure definitions,
- no recursive definition.
Structured Operational Semantics

A set of rules describe program computation. They derive judgements in the form:

\[ \langle a, \sigma \rangle \Rightarrow n \]

meaning: the evaluation of the expression \( a \) with the store (memory) \( \sigma \) returns the value (number) \( n \).

The evaluation of an expression depends only on store. The effect is to generate a number (the store is left unchanged.)

And in the form:

\[ \langle c, \sigma \rangle \Rightarrow \sigma' \]

meaning: the computation of the command \( c \) in the store \( \sigma \) terminates, and the result of the computation is a store \( \sigma' \).

The computation of a command depends only on store. The effect is modify the store (no value is returned.)
SOS rules

Rules have a *natural deduction* style.

\[
\begin{align*}
\langle n, \sigma \rangle \Rightarrow n \\
\langle a_0, \sigma \rangle \Rightarrow n_0 & \quad \langle a_1, \sigma \rangle \Rightarrow n_1 \\
\langle a_0 + a_1, \sigma \rangle \Rightarrow (n_0 + n_1) \\
\langle b, \sigma \rangle \Rightarrow \text{true} & \quad \langle c_0, \sigma \rangle \Rightarrow \sigma' \\
\langle \text{if } b \text{ then } c_0 \text{ else } c_1 \text{ fi}, \sigma \rangle \Rightarrow \sigma'
\end{align*}
\]

- There is one, or more, rules for each program constructor.
- Most of the rule are intuitive.
- Formalize the intended meaning of program constructors in a simple and synthetic way.
On the hypothesis that $\sigma(X) = \sigma(Y) = 3$, we can derive:

$$\langle X, \sigma \Rightarrow 3 \rangle \quad \langle Y, \sigma \Rightarrow 3 \rangle$$

$$\frac{\langle X = Y, \sigma \Rightarrow \text{true} \rangle}{\langle X + 1, \sigma \Rightarrow 4 \rangle}$$

$$\langle \text{if } X = Y \text{ then } X := X + 1 \text{ else } Y = 0 \text{ fi}, \sigma \rangle \Rightarrow \sigma[4/X]$$
SOS at work, meta reasoning, examples:

- If $\langle a, \sigma \rangle = n$ and $\langle a, \sigma \rangle = m$ then $m = n$.
  
  **Proof** By induction on the structure of $a$, showing that, for any $a, \sigma$, there is only one possible derivation;

- **while** $b$ **do** $c$ **od**

  and

  **if** $b$ **then** $c$; **while** $b$ **do** $c$ **od** else **skip** fi

are equivalent. That is, for any $b$, $c$ and $\sigma$, the two commands returns the same store.

**Proof** By case analysis. Exercise.
Software tools

Formal reasoning on programs is quite lengthy: a formal proof of an intuitively obvious fact can take several pages.

Formal semantics is useful because:

- Behavior of programs can be hide subtilities, especially when concurrency and aliasing are involved.
  What is the value of $Y$ at the end of the command $X = 1; [Y = X - X \parallel X = 2]$?
- Complex code almost always contains mistakes.
- Once the reasoning on program is formalize, it can be mechanized.

Software tools can greatly facilitate formal analysis of programs.
- Guiding and checking the correctness of the formal analysis.
- Automatizing the simple steps in the formal analysis.
Simple, synthetic, intuitive.

Quite flexible:

- Can easily accommodate various program feature: environments, higher-order types, concurrency.
- The basic structure of rules remains unchanged when SOS is applied to different programming languages.

We will consider concurrency, (and higher order types).
The semantics is syntax dependent.

The semantics is not compositional: the behaviour of program cannot be described starting from the behaviour of its subterms.

The semantics does not induce a suitable equivalence relation on programs.
Denotational semantics

Aims:
- a language independent semantics, to compare programs written in different programming languages
- a semantics that is compositional, the behaviour of a program is obtained from the behaviour of its components
- more abstract, inspire new methods for reasoning on programs.

Main idea, to describe the behaviour of a program through a suitable mathematical object.
A command is described as a **partial** function from States to States ($\Sigma \rightarrow \Sigma$).

$$C[\ ] : \Sigma \rightarrow \Sigma$$

In turn,

$$\Sigma = \text{Loc} \rightarrow \mathbb{Z}$$

a state (store) assigns to each **location** an **integer**.
The interpretation function, assigning to each expression and command its semantics, is described by a set of equations, in the form

\[ \mathcal{A}[[n]](\sigma) = n \]

\[ \mathcal{A}[[a_0 + a_1]](\sigma) = (\mathcal{A}[[a_0]](\sigma)) + (\mathcal{A}[[a_1]](\sigma)) \]

\[
\mathcal{C}[[\text{if } b \text{ then } c_0 \text{ else } c_1 \text{ fi}}](\sigma) = \begin{cases} 
\mathcal{C}[[c_0]](\sigma) & \text{if } \mathcal{B}[[b]](\sigma) = \text{true} \\
\mathcal{C}[[c_1]](\sigma) & \text{if } \mathcal{B}[[b]](\sigma) = \text{false}
\end{cases}
\]
The difficult case: the \textbf{while} constructor

\[
C[\text{while } b \text{ then } c \text{ od}](\sigma) = \begin{cases} 
\sigma \\
C[\text{while } b \text{ then } c \text{ od}](C[c](\sigma)) 
\end{cases} 
= \begin{cases} 
\text{if } B[b](\sigma) = \text{false} \\
\text{if } B[b](\sigma) = \text{true} 
\end{cases}
\]

This is a recursive definition (normally formulated in different form). One needs to prove that the recursive definition has a solution and to characterize it.

In this case, we use:

- the partial order of partial function on $\Sigma$

\[
C[\ ] : \Sigma \rightarrow \Sigma
\]

- and a variation of Knaster-Tarski theorem: any monotone and continuous function on chains complete orders has a minimal fixed-point.

(Knaster-Tarski theorem: monotone functions on complete partial has a minimal fixed-point)
In more detail

\[ C[\text{while } b \text{ do } \text{cod}] \]

turns out to be the limit of the following list of partial functions:

\[
\begin{align*}
C[\bot] \\
C[\text{if } b \text{ then } c; \bot \text{ fi}] \\
C[\text{if } b \text{ then } c; \text{if } b \text{ then } c; \bot \text{ fi } \text{ fi}] \\
C[\text{if } b \text{ then } c; \text{if } b \text{ then } c; \text{if } b \text{ then } c; \bot \text{ fi } \text{ fi } \text{ fi}] \\
\ldots
\end{align*}
\]

where \( \bot \) is always non-terminating program

and \textbf{if } b \textbf{ then } c \textbf{ fi} is syntactic sugar for \textbf{if } b \textbf{ then } c; \textbf{ else } \text{skip} \textbf{ fi}

(use syntactic sugar is a way to enrich the language without adding new semantics definition)
Domain theory

Standard denotational semantics uses domain theory, a class of partial orders with some additional properties.

1. monotone increasing chains have limits,
2. any elements is the limits of a chain of finite elements,
3. finite elements are elements that can be described by a finite amount of informations (for example: a partial function defined on a finite set of points).

The order on object is the information order: $a \sqsubseteq b$ if $b$ contains more information of $a$.

In the particular case of partial functions: $a \sqsubseteq b$ if $b$ is define on more points, gives more outputs that $a$.

General domain theory is necessary to accommodate real languages, with environment, higher-order function, non concurrency.
Alternatives versions of domain theory

There are a plethora of different kind of domains.

- Scott-Domains – consistently complete dcpo
- SFP-Domains
- Continuous-Domains
- Coherent spaces

Different property for the information order and for the finite elements. Different construction can be performed.
Alternatives to domain theory

Different mathematical structure to interpret languages

- **Game semantics**. A program described by the interaction of the program with its environment, (not as a function from input to output). A different paradigm.

- **Metric spaces and not-well-founded sets**. Semantics of concurrent languages.

- **$C^*$ algebras**.

**Category theory** is used to derived general results, to give prescription on the newly defined semantics structures.

Quite often, the underlying mathematics is quite (too) sophisticated.
A simple concurrent program: IMP + program parallel composition,

\[ c ::= \ldots | c_0 \parallel c_1 \]

informally the execution of the two commands \(c_0\) and \(c_1\) proceeds in parallel, 
or alternatively 
the execution of the two commands can be interleaved.
Concurrency

SOS for parallel language need a different set of judgement.

Judgments in the form

\[ \langle c, \sigma \rangle \Rightarrow \sigma' \]

Specify the input-output behaviour of \( c \).

To determine the evaluation of \( c_0 \parallel c_1 \) is not sufficient to know the input-output behaviour of \( c_0 \) and \( c_1 \) given by th

We need to know also how the computation of \( c_0 \) an \( c_1 \) proceeds.

This is can be specified by a different set of judgments.
Small-step operational semantics

An alternative formulation of SOS uses judgements are in the form:

\[ \langle c, \sigma \rangle \rightarrow \langle c', \sigma' \rangle \]

meaning, in one step of computation, the command \( c \) in the store \( \sigma \) evolves in command \( c' \) and store \( \sigma' \)

and in the form:

\[ \langle c, \sigma \rangle \rightarrow \sigma' \]

meaning, in one step of computation, the command \( c \) in the store \( \sigma \) terminates and returns the store \( \sigma' \)

The rule for all constructor needs to be reformulated.
Example: the new rules for composition

Big-step

\[
\langle c_0, \sigma \rangle \Rightarrow \sigma' \quad \langle c_1, \sigma' \rangle \Rightarrow \sigma''
\]

\[
\langle c_0; c_1, \sigma \rangle \Rightarrow \sigma''
\]

Small-step:

\[
\langle c_0, \sigma \rangle \rightarrow \langle c_0', \sigma' \rangle
\]

\[
\langle c_0; c_1, \sigma \rangle \rightarrow \langle c_0'; c_1, \sigma' \rangle
\]

\[
\langle c_0', \sigma \rangle \rightarrow \sigma'
\]

\[
\langle c_0; c_1, \sigma \rangle \rightarrow \langle c_1, \sigma' \rangle
\]
Parallel composition

Just small step operational semantics:

\[
\begin{align*}
\langle c_0, \sigma \rangle & \rightarrow \langle c'_0, \sigma' \rangle \\
\langle c_0 \parallel c_1, \sigma \rangle & \rightarrow \langle c'_0 \parallel c_1, \sigma' \rangle \\
\langle c_1, \sigma \rangle & \rightarrow \langle c'_1, \sigma' \rangle \\
\langle c_0 \parallel c_1, \sigma \rangle & \rightarrow \langle c_0 \parallel c'_1, \sigma' \rangle
\end{align*}
\]

The reduction relation is non deterministic.

Non determinism is accommodated smoothly. There is no need to modify the SOS approach.
Communicating processes

The above parallel composition \( \parallel \) assumes a common store. Parallel process communicate through the store. Multiprocessors architecture: several processors sharing a common memory.

Multicomputer architecture: several computers communicating through a network. They need a different programming style. No common store, parallel processes communicate through message exchange.
Communicating processes

IMP + communication primitives became

c ::= skip | X := a | c₀; c₁ | if b then c₀ else c₁ fi | while b do c od | α!a | α?X | c₀ || c₁

Where:

- α range on a set of communication channels,
- α!a is a command (process) that sends the value a along the channel α
- α?X is the process that receives a value along the channel α and stores the received value on the location X.
- In the parallel composition of two processes c₀ || c₁, processes c₀ and c₁ cannot have a shared location.

The above calculus is similar to CSP: Communicating Sequential Processes
To analyze concurrency, it is convenient to study a language without imperative features, containing only basic concurrent constructors.

<table>
<thead>
<tr>
<th>( p \ ::= )</th>
<th>( nil )</th>
<th>( \alpha.p )</th>
<th>( \overline{\alpha}.p )</th>
<th>( \tau.p )</th>
<th>( p_0 \parallel p_1 )</th>
<th>( p_0 + p_1 )</th>
<th>( X )</th>
<th>( recX = p )</th>
<th>( p \setminus [\alpha_1, \ldots, \alpha_n] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>empty action,</td>
<td>communication along a channel,</td>
<td>silent action,</td>
<td>parallel and non-deterministic composition,</td>
<td>recursive definition, infinitary processes</td>
<td>action hiding, creates private channels.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Two main reductions rules:

\[
((\alpha.p_0) + p_1) \parallel ((\overline{\alpha}.q_0) + q_1) \rightarrow p_0 \parallel q_0
\]

\[
((\tau.p_0) + p_1) \rightarrow p_0
\]

Together with a set of equations defining structurally congruent processes:

\[
p_0 + p_1 \equiv p_1 + p_2 \quad p_0 + (p_1 + p_2) \equiv (p_0 + p_1) + p_2
\]

\[
\ldots
\]

\[
\text{rec } X = p \equiv p[\text{rec } X = p/X]
\]

\[
\ldots
\]

and the rule

\[
p \rightarrow q \quad p \equiv p' \quad q \equiv q' \quad \overrightarrow{p'} \rightarrow \overrightarrow{q'}
\]
An alternative description: to derive the communications that a process can perform with the environment.

Judgement in the form:

$$ p \xrightarrow{\alpha} q $$

\( p \) communicates along the channel \( \alpha \) and becomes the process \( q \).
Basic rules describing communicating features:

\[
\begin{align*}
\alpha \cdot p & \xrightarrow{\alpha} p \\
\tau & \rightarrow q \\
p_0 + p_1 & \xrightarrow{\alpha} q \\
p_0 & \xrightarrow{\alpha} q_0 \\
p_1 & \xrightarrow{\alpha} q_1 \\
p_0 \parallel p_1 & \xrightarrow{\tau} q_0 \parallel q_1
\end{align*}
\]

\(\tau\) represents the silent action, a step of computation with no communication involved.

Also here it is necessary to add a rule stating that a process can be substituted by a congruent one.
Denotational semantics for processes: a process is described by the (unordered, infinitary) tree of communications it can perform. For example, the process:

\[ \text{rec}X = \alpha.\text{nil} + \beta.X \equiv \alpha.\text{nil} + \beta.\text{rec}X = \alpha.\text{nil} + \beta.X \]

is represented by the (unordered) tree:
Two processes are equivalent if they generate equivalent infinitary trees. Two trees are equivalent if one can be transformed into the other by (an infinite number of) permutations, duplications, contractions of some subtrees. Formally, tree equivalence is captured by the notion of **bisimulation**.
A symmetric relation $R$ on processes is bisimulation if
For any $p, q, p'$
\[ pRq \land p \xrightarrow{\tau} p' \]
implies that there exits $q'$ such that
\[ q \xrightarrow{\tau} q' \land p'Rq' \]

Two processes are bisimilar ($p \sim q$) if they are related by bisimulation. Informally: two bisimilar processes generates equivalent (bisimilar) tree.
A mathematical structures in which to interpret processes. More directed that the set of tree quotient by bisimuation. Several proposals: not-well founded set, metric spaces, final coalgebra.

Bisimilarity is a congruence relation:

\[ p \sim q \Rightarrow C[p] \sim C[q] \]

Relate reductions semantics, and LTS semantics.

Move from a reductions system to a (good, congruent) LTS systems. To obtain a denotational semantics starting from a operational semantics.
Categories theory are quite often used in denotational semantics. A possible reason: too many different mathematical structures are used in semantics.

Category theory is able

- to give a global framework in which to present general results.
- to characterize the key properties that a (new) mathematical structures need to have.

Prescriptive role.
Category theory can be seen as further generalization form the generalization provided by the notion of group, algebra, topological space.

**Definition.** A category theory is formed by:

- a set of objects, \( \{A, B, C, \ldots \} \)
- for any pair of objects, a set of morphism, \( \{f, g, h, \ldots : A \to B\} \)
- on morphism there is an operation of composition \( \circ \),
  \[ f : A \to B, \ g : B \to C, \ g \circ f : A \to C \]
  - that is **associative**: \( h \circ (g \circ f) = (h \circ g) \circ f \) and
  - has **identity elements**: \( \text{id}_A : A \to B, \ f \circ \text{id}_A = f, \ \text{id}_B \circ f = f. \)
Examples

- sets and functions;
- groups and groups morphisms (functions respecting group operations);
- topological spaces and continuous functions;
- elements of a partial order and arrows representing the order relations.
Main ideas

To express properties, defining concepts in term of morphisms, without looking at the internal structure of objects. Examples:

- **Final object**: an object $C$ is final if for any object $A$ there is a single morphism from $A$ to $C$
  - in **Sets**, a final object is

- Initial object: an object $C$ is initial if for any object $A$ there is a single morphism from $C$ to $A$.
  - in **Sets**, the initial object is
  - in **Groups**, an initial object is
  - in the category induced by a partial order, the initial object, if it exists, is the minimum element.
Main ideas

To express properties, defining concepts in term of morphisms, without looking at the internal structure of objects. Examples:

- **Final object**: an object $C$ is final if for any object $A$ there is a single morphism from $A$ to $C$
  - in **Sets**, a final object is any singleton set;
  - in **Groups** a final object is
Main ideas

To express properties, defining concepts in term of morphisms, without looking at the internal structure of objects. Examples:

- **Final object**: an object \( C \) is final if for any object \( A \) there is a single morphism from \( A \) to \( C \)
  - in **Sets**, a final object is any singleton set;
  - in **Groups** a final object is a group formed a single elements;
  - in the category induced by a partial order, the final object,
Main ideas

To express properties, defining concepts in term of morphisms, without looking at the internal structure of objects. Examples:

- **Final object**: an object $C$ is final if for any object $A$ there is a single morphism from $A$ to $C$
  - in **Sets**, a final object is any singleton set;
  - in **Groups** a final object is a group formed a single elements;
  - in the category induced by a partial order, the final object, if it exists, is the maximum element.

- **Initial object**: an object $C$ is initial if for any object $A$ there is a single morphism from $C$ to $A$.
  - in **Sets**, the initial object is
Main ideas

To express properties, defining concepts in term of morphisms, without looking at the internal structure of objects. Examples:

- **Final object**: an object $C$ is final if for any object $A$ there is a single morphism from $A$ to $C$
  - in $\text{Sets}$, a final object is any singleton set;
  - in $\text{Groups}$ a final object is a group formed a single elements;
  - in the category induced by a partial order, the final object, if it exists, is the maximum element.

- **Initial object**: an object $C$ is initial if for any object $A$ there is a single morphism from $C$ to $A$.
  - in $\text{Sets}$, the initial object is the empty set;
  - in $\text{Groups}$ an initial object is
Main ideas

To express properties, defining concepts in term of morphisms, without looking at the internal structure of objects. Examples:

- **Final object**: an object $C$ is final if for any object $A$ there is a single morphism from $A$ to $C$.
  - in **Sets**, a final object is any singleton set;
  - in **Groups** a final object is a group formed a single elements;
  - in the category induced by a partial order, the final object, if it exists, is the maximum element.

- **Initial object**: an object $C$ is initial if for any object $A$ there is a single morphism from $C$ to $A$.
  - in **Sets**, the initial object is the empty set;
  - in **Groups** an initial object is a group formed a single elements (initial and final object coincide);
  - in the category induced by a partial order, the initial object, if it exists, is the minimum element.
Product

Cartesian product $\times$ can be defined categorically. The product of two objects $A, B$ is an object, denoted by $A \times B$, such that:

- there are two morphisms $\pi_1 : (A \times B) \to A$ and $\pi_2 : (A \times B) \to B$, called projections,

- for any other object $C$ and for any pair of morphisms $f : C \to A$ and $g : C \to B$, there exists a morphism $\langle f, g \rangle : C \to (A \times B)$ such that;
  
  \[ f = \pi_1 \circ \langle f, g \rangle \]  
  \[ g = \pi_2 \circ \langle f, g \rangle, \]

i.e. the following diagram commutes.

![Diagram](https://via.placeholder.com/150)
Examples

In the following categories categorical product coincides with:

- **Sets**: the cartesian product of two sets,
- **Groups**: the product of two groups, with the correct operations and unit element,
- **Tops**: the topological product, with the correct induced topology,
- in the category induced by a partial order: the greatest lower of two points.
Dually, the disjoint union $+\,$ of sets is defined categorically as:
The coproduct of two objects $A, B$ is an objects, denoted by $A + B$, such that:

- there are two morphism $\nu_1 : A \to (A + B)$ and $\nu_2 : B \to (A + B)$, called injections,
- for any other object $C$ and for any pair of morphism $f : A \to C$ and $g : B \to C$,
  there exist a morphism $[f, g] : (A + B) \to C$ such that;
  $f = [f, g] \circ \nu_1$ and $g = [f, g] \circ \nu_2$,
  i.e. the following diagram commutes.

\[
\begin{array}{ccc}
A & \xrightarrow{\nu_1} & A + B \\
\downarrow{f} & & \downarrow{[f, g]} \\
C & \xleftarrow{\pi_2} & B
\end{array}
\]
An algebra, that is a set together $A$ with a list of operation on $A$. An algebra, on a set $A$ can be defined as a morphism from a coproduct of products of $A$ into $A$. For example, to define a group structure on $A$, one needs to define

- the zero element $0$ that is a function $\{\ast\} \to A$
- the inverse function $^{-1}$, that is function $A \to A$
- the addition function $\cdot$, that is function $A \to (A + A)$

All together they form a morphism $[0, ^{-1}, \cdot] : (1 + A + (A \times A)) \to A$

Here we do not consider the equality laws.
The transformation from \( A \) to \( (1 + A + (A \times A)) \),
Defined a functor, from \textbf{Set} to \textbf{Set}.

A functor, \( F \), from a category \( A \) to a category of \( B \) is a functions from objects of \( A \) to objects \( B \) and a function from maps of \( A \) to maps of \( B \), preserving the categorical structure (identity and composition)

\[
F(f \circ g) = F(f) \circ F(g)
\]
Categorical generalization of the notion of algebra

Given an category $\mathbf{A}$, and an endofuctor $F : \mathbf{A} \to \mathbf{A}$,

- an $F$-algebra is given by an object $A$ and a morphism $h : F(A) \to A$;
- a morphism between an $F$-algebra $\langle A, h \rangle$ and an $F$-algebra $\langle A', h' \rangle$, is a map $g : A \to A'$ such that the following diagram commutes:

\[
\begin{array}{ccc}
F(A) & \xrightarrow{F(g)} & F(A') \\
\downarrow h & & \downarrow h' \\
A & \xrightarrow{g} & A'
\end{array}
\]

- $F$-algebras and $F$-algebra morphisms form a category.

When instantiated on the special case of $F(A) = (1 + A + (A \times A))$, in the $\mathbf{Set}$ category, we obtained the standard notion of algebra and algebra morphism.
Initial $F$-algebra

An $F$-algebra $\langle A, h \rangle$ is initial, if for any other algebra $\langle B, j \rangle$ there is exactly one $F$-algebra morphism from $\langle A, h \rangle$ to $\langle B, j \rangle$.

The initial algebra for the functor $F(A) = (1 + A + (A \times A))$, in the Set category, is the free-algebra, i.e. the set of terms that can be construct starting from three constants

- $0$ a ground constant
- $-1$ a unary function constant.
- $\cdot$ a binary function constant.

The syntax on a given signature can be characterize as the initial algebra.
Duality

In category theory, any notion has its dual obtained by inverting the arrows. In this case

- $F$-coalgebra
- Final $F$-coalgebras.

When instantiated on $\textbf{Set}$ (and on particular functors), final coalgebras are the set on infinitary terms (trees), build on a given signatures.

Final coalgebra can represent the infinitary behaviour of process. i.e. as semantics models.
Deepen the study of one of the subjects presented in the course, solve one the following exercises.

- Propose an small-step SOS for IMP. Sketch a proof that small-step and big-step SOS agree with each other.
- Propose a denotational semantics for a repeat command

\[
\text{repeat } c \text{ until } b.
\]

- Determine whether the following pair of processes are bisimilar:
  - \((\text{rec } X = \alpha.X) \parallel (\text{rec } Y = \overline{\alpha}.Y)\) and
  \[
  \text{rec } X = ((\alpha.X) + (\overline{\alpha}.X) + (\tau.X)).
  \]
  - \(\tau.((\alpha.nil) + (\beta.nil))\) and \((\tau.\alpha.nil) + (\tau.\beta.nil)\)

prove your claims.

Reference. The formal semantics of programming languages. G. Winskel, MIT Press.