Sémantique des langages de programmation

Master 1

Delia Kesner
UFR d’Informatique
Université de Paris

Email : kesner@irif.fr
URL : www.irif.fr/~kesner
Informations générales
Informations générales

- Enseignantes: D. Kesner (cours), C. Faggian (TD)
- Créneaux horaires:
  - Cours: Lundi 13h45-15h45
  - TD: Mercredi 16h00-18h00
- Note module: (CC + Examen)/2
- 2ème session: (CC + Examen)/2
Software runs our lives today. (Critical) software is everywhere.
Space
Transportation
Medical
Telecommunications
Energy
Household Appliances
Military
Encryption
Bugs

Bugs are also everywhere, and failure is expensive.

- **Bugs**: found in PC, Smartphones, enterprise software. They cause uncomfortable work, more actions than expected, inconsistency, frustration, loss of time, frequent upgrades, etc.

- **Bad bugs**: found in data security, network security, internet privacy. They cause data corruption, invalid data representation, unavailability to perform action, loss of integrity, loss of money, begin sued.

- **Critical bugs**: found in railways, medical, nuclear, airplanes. They damage health, nature, buildings, technics.... and all above.
Famous Bugs
Famous Bugs

- Ariane 5 Flight 501 was destroyed 40 seconds after takeoff (1996).
- Virus ILOVEYOU (2000).
- Google’s search engine (2009).
- Pac-Man (256th level).
Prevention

Safety vs correctness, automatic vs interactive.

- Testing (empirical, finite, dominant approach)
- Programming techniques (high-level languages, types, etc)
- Formal specifications (metro systems using Z)
- Static analysis (specificity, reachability properties)
  - Abstract Interpretation (approximation)
  - Model checkers (finite state)
- Semantics of programming languages
- Deduction program provers (first-order logic, automatic)
- Proof assistants (higher-order logics)

Rigorous prevention tools involve FORMAL METHODS using mathematical and logical theories.
## Mathematics in Computer Science

<table>
<thead>
<tr>
<th>Computer Science</th>
<th>Mathematics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Semantics of programming languages</td>
<td>Logic, Category Theory</td>
</tr>
<tr>
<td>Machine learning</td>
<td>Statistics</td>
</tr>
<tr>
<td>Cryptography</td>
<td>Number Theory</td>
</tr>
<tr>
<td>Numerical analysis</td>
<td>Differential Equations</td>
</tr>
<tr>
<td>Analysis of algorithms</td>
<td>Complex Analysis</td>
</tr>
<tr>
<td>Network performance</td>
<td>Probabilities</td>
</tr>
</tbody>
</table>

- Numerical analysis
- Analysis of algorithms
- Network performance
Logics in Computer Science

- Gottlob Frege (1848–1925). Logical basis for mathematics.
- David Hilbert (1862–1943). One of the founders of proof theory.
- Bertrand Russell (1872–1970). One of the founders of proof theory.
- Alain Turing (1912–1954). Father of theoretical computer science.
- William Alvin Howard (1926). Lambda calculus.
- Per Martin-Löf (1942). Type Theory.
Testing - Dominant Approach

"In several cases, engineers have reported finding flaws in systems once they reviewed their designs formally." (R. Kling, 1995)

"Program testing can be used to show the presence of bugs, but never to show their absence! " (E. W. Dijkstra, 1972)

Formal methods are used to verify and infer properties that hold for **ALL** executions of a program. Used in industry to

- Obtain some guarantees (besides testing).
- Prove absence of some kind of errors.
- Save time and money (rigorous testing is expensive).
Certification - Huge progress

- Kepler conjecture (algebra), by T. Hales, ongoing work.
- Feit Thompson Theorem (finite groups), by G. Gonthier.
- Full certification of java compiler, by X. Leroy.
Programming techniques, programming languages, proof assistants.
Programming Languages

Syntax: what do programs are supposed to express?
Dynamics: how programs execute? what do they compute?
Types: what are well-formed programs?
Typing at primary school

A technique to automatically detect a class of (programming) errors.

\[
\begin{align*}
2 \quad \text{🍎} & \quad + \quad 3 \quad \text{🍎} & \quad = \quad 5 \\
3 \quad \text{🏠} & \quad + \quad 6 \quad \text{🏠} & \quad = \quad 9 \\
2 \quad \text{🍎} & \quad + \quad 6 \quad \text{🏠} & \quad = \quad ?
\end{align*}
\]
Correct and well-typed program:

```ml
let rec fact n =  
if n < 2 then 1 else fact(n - 1) * n
```

Ill-typed program:

```ml
let rec fact n =  
if n < 2 then 1 else fact(n - "toto") * n
```

Uncorrect but well-typed program:

```ml
let rec fact n =  
if n < 2 then 1 else fact(n + 1) * n
```
Typed algebras

Type = a set of data having the same form.

Base Types:
- int: integer numbers 0, 1, 12, ...
- bool: truth values true and false
- string: list of char

Composed Types:
- \( A \times B \): pairs with first (resp. second) element of type \( A \) (resp \( B \))
- \( A + B \): elements of type \( A \) or \( B \)
- \( A \rightarrow B \): functions taking arguments of type \( A \) and yielding results of type \( B \)
- list(\( A \)): lists of elements of type \( A \)

Exemple: list(int \( \times \) string) \( \rightarrow \) list(int) \( \times \) list(string)
Towards the notion of well-typed terms

- Constants 0, 1, 2, ... have type int.
- The expression $x < y$ is well-typed iff $x$ and $y$ have type int.
- The expression if $c$ then $x$ else $y$ is well-typed iff $c$ has type bool and $x$ and $y$ have the same type.

Exemples:
if 1 < 2 then 3 else 4 correct
true < false wrong
if 1 < 2 then 3 else false wrong
Types, formally

Typing judgments:

\[ x_1 : A_1, \ldots, x_n : A_n \vdash \text{expr} : B \]

Intuitive meaning:
"If variables \( x_1, \ldots, x_n \) have type \( A_1, \ldots, A_n \) resp. then \( \text{expr} \) has type \( B \)."

Typing Rules:

\[
\frac{H_1 \ldots H_n}{\text{C}}
\]

where \( H_1, \ldots, H_n, C \) are typing judgments.

Intuitive meaning:
"If premisses \( H_1, \ldots, H_n \) are true, then the conclusion \( C \) is true."
Example

\[ \Gamma \vdash 0 : \text{int} \]

\[ \Gamma(x) = A \]

\[ \Gamma \vdash x : A \]

\[ \Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int} \]

\[ \Gamma \vdash (e_1 + e_2) : \text{int} \]

\[ \Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : A \quad \Gamma \vdash e_3 : A \]

\[ \Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : A \]

\[ \Gamma \vdash f : A \to B \quad \Gamma \vdash a : A \]

\[ \Gamma \vdash fa : B \]

\[ x : A, \Gamma \vdash b : B \]

\[ \Gamma \vdash \text{function } x \mapsto b : A \to B \]
Curry-Howard Isomorphism

Logical system ⇔ Language
  Propositions ⇔ Types
  Logical Rules ⇔ Typing Rules
  Proofs ⇔ Programs
Curry-Howard

Logical system

Propositions

- \( A \) and \( B \)
- \( A \) or \( B \)
- \( A \) implies \( B \)

Logical Rules

\[
\Gamma \vdash A \text{ implies } B \quad \Gamma \vdash A
\]

\[
\Gamma \vdash B
\]

Language

Types

- \( A \times B \)
- \( A + B \)
- \( A \rightarrow B \)

Typing Rules

\[
\Gamma \vdash f : A \rightarrow B \quad \Gamma \vdash a : A
\]

\[
\Gamma \vdash f a : B
\]
Some CH examples

**Logical system**
- Intuitionistic propositional Logic
- Intuitionistic second order Logic
- Intuitionistic higher-order Logic
- Classical Logic

**Language**
- Typed Lambda Calculus
- Functional Languages (CAML, Haskell)
- Type Theory (Coq, Isabell)
- Control Operators (Scheme)
Types and expressivity

- Subtyping: natural numbers can be considered a subtype of real numbers, so that a function accepting real numbers, accepts also natural numbers.

- Path Polymorphism: a function can be applied to any argument having a particular data structure.

- Linear types: unique use of data.

- Intersection types: overloading, characterization of mathematical properties.

- Union types: values belonging to different types.

- Existential types: use to model abstract data types.
Types and Languages/Proof Assistants

- **Typed Lambda Calculus**: a function of type \( \text{int} \rightarrow \text{int} \) can only be applied to an integer. But not Turing complete: only total functions.

- **Polymorphic Lambda Calculus**: a function of type \( \alpha \rightarrow \alpha \) can be applied to any type, in particular to an argument of type \( \text{int}, \text{int} \rightarrow \text{int}, (\text{int} \rightarrow \text{int}) \rightarrow \text{int} \), etc. Example: Caml, Haskell.

- **Dependent Types**: distinction between arrays of different lengths (i.e. \([] : \text{list}(0)\) and \([1; 3] : \text{array}(2)\). Example: AGDA, Epigram.

- **Calculus of Constructions**: polymorphism + dependent types. Powerful enough to model a proof assistant. Examples: Coq proof assistant.
The Lambda Cube

- **Terms** depending on **terms**.
- **Terms** depending on **types**, or polymorphism (2).
- **Types** depending on **types**, or type operators ($\omega$).
- **Types** depending on **terms**, or dependent types ($P$).
Programming Languages

- Functional Programming
- Logic Programming
- Data Base languages
- Concurrent Languages (Mobility, Security)
- Imperative Programming
- Rewriting Programming
- Object-Oriented Programming
- Proof Assistants
- ...
Syntax vs Semantics

Syntax:

$$\forall x. s(x) \neq 0$$

Semantics:

"Every positive natural number is different from zero"
Semantics of programming languages

The meaning of programming languages.

Different techniques:

- Operational semantics
- Denotational semantics
- Axiomatic semantics
Operational semantics

The meaning of a program is the sequence of states of the machine executing the program.

```
program = transition system
```

Example

P1: a=1; b=0;  
    P2: a=1;    
    P3: b=0; a=1;  
    P4: a=1; b=1;  
    b=0;

P1 is equivalent to P2, but P1 is not equivalent to P3/P4, and P3 is not equivalent to P4.

⚠️ non equivalent programs may have the same result.
This notion can be refined in order to observe just a piece of memory of the machine.
Denotational semantics

program = mathematical function

Argument of this function: the state of the memory before execution
Result of this function: the state of the memory after execution

Example
P1: a=1; b=0;  P2: a=1;  P3: b=0; a=1;  P4: a=1; b=1;
   b=0;

P1, P2 and P3 are equivalent, but they are not equivalent to P4.
Axiomatic semantics

program = transformation of logical properties

This can be written for example in Hoare logic as

\{P\} Prog \{Q\}

Thus for example,

\{true\} P1/P2 : \ a = 1; b = 0; \ \{a \geq 0 \land b \geq 0\}
\{true\} P3 : \ b = 0; a = 1; \ \{a \geq 0 \land b \geq 0\}
\{true\} P4 : \ a = 1; b = 1; \ \{a \geq 0 \land b \geq 0\}

are all equivalent.
Relation between different semantics

Two syntactic equivalent programs are operational equivalent.
Two operational equivalent programs are denotational equivalent.
Two denotational equivalent programs are axiomatic equivalent.

The converse implications are in general false.
Operational Semantics

\[ \text{program} = \text{transition system} \]

Rewriting systems are a natural tool to model transitions between objects.
Example I: String rewriting

Rewriting system:

```
vert  \implies orange
orange \implies rouge
rouge \implies vert
```

Reduction sequence:

```
vert \rightarrow orange \rightarrow rouge \rightarrow vert \rightarrow orange \rightarrow \ldots
```
Example II: String rewriting

Rewriting system:

\[
\begin{align*}
\circ \bullet & \Rightarrow \bullet \circ \\
\bullet \bullet & \Rightarrow \bullet \bullet \\
\bullet \circ & \Rightarrow \circ \bullet
\end{align*}
\]

Reduction Sequence:

\[
\begin{align*}
\circ \bullet \bullet \circ \bullet \bullet \circ \bullet \\
\downarrow^* \\
\bullet \bullet \circ \circ \circ \bullet \bullet \bullet \bullet
\end{align*}
\]
Example III: Term rewriting

Rewriting system for Peano arithmetic:

\[
\begin{align*}
0 + Y & \mapsto Y \\
\text{s}(X) + Y & \mapsto \text{s}(X + Y) \\
0 \ast Y & \mapsto 0 \\
\text{s}(X) \ast Y & \mapsto (X \ast Y) + Y
\end{align*}
\]

Reduction sequence:

\[
\begin{align*}
\text{“}2 \ast 3\text{”} &= \text{s}(\text{s}(0)) \ast \text{s}(\text{s}(\text{s}(0))) \\
& \rightarrow \\
\text{s}(\text{s}(0)) \ast \text{s}(\text{s}(\text{s}(0))) + \text{s}(\text{s}(\text{s}(0))) \\
& \rightarrow \\
0 \ast \text{s}(\text{s}(\text{s}(0))) + \text{s}(\text{s}(\text{s}(0))) + \text{s}(\text{s}(\text{s}(0))) \\
& \rightarrow \\
0 + \text{s}(\text{s}(\text{s}(0))) + \text{s}(\text{s}(\text{s}(0))) \\
& \rightarrow \\
\text{s}(\text{s}(\text{s}(0))) + \text{s}(\text{s}(\text{s}(0))) \\
& \rightarrow \\
\text{s}(\text{s}(\text{s}(0)) + \text{s}(\text{s}(\text{s}(0)))) \\
& \rightarrow \\
\text{s}(\text{s}(\text{0 + s}(\text{s}(\text{s}(0)))))) \\
& \rightarrow \\
\text{s}(\text{s}(\text{0 + s}(\text{s}(\text{s}(\text{s}(0))))))) \\
& \rightarrow \\
\text{s}(\text{s}(\text{s}(\text{s}(\text{s}(\text{s}(0))))))) = \text{“}6\text{”}
\end{align*}
\]
Example IV: Equational Programming

Rewriting system:

\[
\begin{align*}
nil[X\backslash Y] & \rightarrow nil \\
\text{cons}(H, T)[H\backslash I] & \rightarrow \text{cons}(I, T[H\backslash I]) \\
\text{cons}(J, T)[H\backslash I] & \rightarrow \text{cons}(J, T[H\backslash I])
\end{align*}
\]

Reduction sequence:

\[
\begin{align*}
\text{cons}(a, \text{cons}(b, \text{cons}(a, \text{cons}(b, \text{nil}))))[a\backslash c][b\backslash d] & \rightarrow^* \\
\text{cons}(c, \text{cons}(d, \text{cons}(c, \text{cons}(d, \text{nil}))))
\end{align*}
\]
Example V: Logical reasoning

Rewriting system:

\[
egin{align*}
P & \supset Q \iff \neg P \lor Q \\
\neg(P \land Q) & \iff \neg P \lor \neg Q \\
\neg(P \lor Q) & \iff \neg P \land \neg Q \\
\neg \neg P & \iff P
\end{align*}
\]

Reduction sequence:

\[
\neg(\neg(\neg r \supset s) \supset t) \rightarrow \\
\neg(\neg(\neg \neg r \lor s) \supset t) \rightarrow \\
\neg(\neg(\neg r \lor s) \supset t) \rightarrow \\
\neg((\neg r \land \neg s) \supset t) \rightarrow \\
\neg((\neg \neg r \land \neg s) \lor t) \rightarrow \\
\neg((\neg r \land \neg s) \lor t) \rightarrow \\
\neg((\neg r \lor \neg s) \lor t) \rightarrow \\
\neg((r \lor \neg s) \lor t) \rightarrow \\
\neg((r \lor s) \lor t) \rightarrow \\
\neg((r \lor s) \land \neg t) \rightarrow \\
(\neg r \land \neg s) \land \neg t
\]
Example VI: Functional Programming

The $\lambda$-calculus [Church]

$(\beta)$ \hspace{1cm} $(\lambda x. M) N \mapsto M \{x \leftarrow N\}$

$(\eta)$ \hspace{1cm} $\lambda x. M \; x \mapsto M$

$(S\; P)$ \hspace{1cm} $\langle \pi_1(M), \pi_2(M) \rangle \mapsto M$
Example VII: Object-Oriented Programming

The $\varsigma$-calculus [Abadi & Cardelli]

An objet $O$ is a collection of methods $\langle l_i \equiv \varsigma \text{self}_i.B_i \rangle_{i \in 1...n}$.
Method invocation is given by the rewriting rules:

$$O.l_j \mapsto B_j\{\text{self}_j\backslash O\}$$
Example VIII: Mathematical reasoning

Formulae:

\[ \exists \alpha. \forall \beta. X \iff \forall \beta. \exists \alpha. X \]

Sets:

\[ T \in \{ \alpha : A \mid P \} \iff T \in A \land P(\alpha \backslash T) \]
Example IX: OCAMML Programs

let rec length l = match l with
  | []  -> 0
  | h::t -> length t + 1 ;;

let rec append l1 l2 = match l1 with
  | []  -> l2
  | h::t -> h::append t l2 ;;

let rec map l f = match l with
  | []  -> []
  | h::t -> (f h):: map t f ;;
Example X: Concurrent Programming

The π-calculus [Milner & Parrow & Walker]

$$\overline{\alpha}(t).P \mid \alpha(x).Q \quad \rightarrow \quad P \mid Q\{x\backslash t\}$$
Typical questions concerning a rewriting model

Consider a rewriting sequence

\[ t_1 \xrightarrow{R} t_2 \xrightarrow{R} t_3 \xrightarrow{R} \ldots \]

- Is this computation terminating? (always? sometimes?)
- Is there a result (e.g. canonical form)?
- Is there uniqueness of results?
(Part of the) History

(Thue-1914) String rewrite systems
(Church-1936) Lambda calculus
(Gorn-1967) Term rewrite systems
(Klop-1980) Combinatory reduction systems
Plan du cours

■ Introduction
■ Quelques notions mathématiques
■ Calculs fonctionnels
  ■ Introduction au lambda calcul
  ■ Propriétés: Confluence, standardisation, résidus, etc.
  ■ Lambda calcul typé: type, jugement de type, dérivation de typage
  ■ Propriétés : préservation du typage, normalisation forte, etc.
  ■ Sémantique opérationnelle: appel par nom, appel par valeur, appel par nécessité, call-by-push-value, sémantique à petits pas et à grands pas.
  ■ Machines à environnement.
  ■ Polymorphisme, inférence de types, types intersection.
■ Calculs algébriques
  ■ Termes et Sigma algèbres
  ■ Théorème de Birkhoff
  ■ Réécriture
  ■ Quelques techniques pour montrer la confluence
  ■ Quelques techniques pour montrer la terminaison
■ Sémantique dénotationnelle
  ■ L’approche dénotationnelle de la sémantique
  ■ Sémantique d’un langage fonctionnel simple
Bibliographie

*Transparents et tableau*
(consulter [www.irif.fr/~kesner](http://www.irif.fr/~kesner) régulièrement)

Pour les calculs algébriques :
*Term Rewriting and All That*. 1998.
Franz Baader, Tobias Nipkow. Cambridge University Press
Pour les calculs fonctionnels :
