Semantics

- Semantics = “Assigning Meaning to Programs” (Robert W. Floyd)

  - Operational Semantics
    - Defining behavior of a programming language based on a simple abstract machine for it.
    - (E.g., state of the machine is a term, machine operates on a transition function)
  
  - Denotational Semantics
    - Instead of describing a term as a sequence of machine states, the meaning of a term is taken to be some mathematical object
    - E.g., number or functions as semantic domains

- Axiomatic Semantics

  - Defining behavior of a language directly in terms of laws.

Syntax vs. Semantics

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Semantic</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 = 1</td>
<td>1 = 1</td>
</tr>
<tr>
<td>x = 1</td>
<td>x = 1</td>
</tr>
<tr>
<td>2 + 2 = 4</td>
<td>2 + 2 = 4</td>
</tr>
<tr>
<td>1 + 3 = 2 + 2</td>
<td>1 + 3 = 2 + 2</td>
</tr>
</tbody>
</table>

It was originally believed that operational semantics were inferior (< 1980)

- “quick and dirty”

- They have become more accepted due to concepts that showed to be difficult to express in the other forms.
  - Axiomatic semantics struggled with procedures
  - Denotational semantics struggled with non-determinism and concurrency
  - More recent work (1980-2000) introduces more elegant formalism and managed to match many of the mathematical techniques developed in the context of denotational in operational semantics.
Java Try/Catch

Assume you wanted to check if a monitor held within a try/catch block is automatically released if an exception occurs within the block...

11.1 The Causes of Exceptions

An exception is thrown for one of three reasons:
- The evaluation of an expression violates the normal semantics of the language, such as an integer divide by zero, as summarized in §15.6.

Exceptions are represented by instances of the class Throwable and instances of its subclasses. These classes are, collectively, the

11.2 Compile-Time Checking of Exceptions

A compiler for the Java programming language checks, at compile time, that a program contains handlers for:

Some expression of the argument list can throw the overridden declarations (§9.4).

The method to be invoked is of the form

\[ E \]

iff either:
- \( E \) is listed in the throws clause of the type of the constructor that is invoked; or
- \( E \) and some instance initializer block or instance variable initializer expression in the

11.2.3 Exception Checking

No preceding catch block of the immediately enclosing try statement catches

Those unchecked exception classes which are the (Error and its subclasses) are exempted from compile-time checking because they can occur at many points in the program and recovery from them is difficult or impossible. A program declaring such exceptions would be cluttered, pointlessly.

For example, certain code might implement a circular data structure that, by construction, can never involve null... that is needed to establish such global properties of data structures is beyond the scope of this specification.

The paper by Marc Feeley, Proc. 1993 Conference on Functional Programming and Computer Architecture, Copenhagen, Denmark, pp. 179-187, is recommended as further reading.

11.4 An Example of Exceptions

divide null not test it produces the output:

catches each exception that the thrower throws. Whether the invocation of thrower completes normally or abruptly, a message is printed describing what happened.

The Java virtual machine throws an object that is an instance of a subclass of LinkageError when a loading, linkage, preparation, verification or initialization error occurs:

The loading process is described in §12.2.

The class verification process is described in §12.3.1.

The class preparation process is described in §12.3.2.

The class initialization process is described in §12.4.

11.5.2 Virtual Machine Errors

The exception mechanism of the Java SE platform is integrated with its synchronization model, so that monitors are unlocked as synchronized statements and invocations of synchronized methods (§8.4.3.6, §15.12)

complete abruptly.

Asynchronous exceptions are rare. They occur only as a result of:
- An invocation of the stop methods of class Thread or ThreadGroup

The paper

Inference Rules in Sequent Calculus

\[ \rho_1, \rho_2, \ldots, \rho_n \rightarrow C \]

\( (\rho_1 \land \rho_2 \land \ldots) \rightarrow C \)

Inference: Act of drawing conclusions

Notation: Gentzen
Example: Natural Deduction

\[
\begin{align*}
A & \quad B & (\wedge i) & \frac{A \wedge B}{A} & (\wedge e) & \frac{A \wedge B}{B} & (\wedge e) & \frac{A}{A \rightarrow B} & (MP) \\
A & \quad \frac{A}{A \lor B} & (\lor i) & \frac{B}{A \lor B} & (\lor i) & \frac{A \lor B}{A \rightarrow C} & \frac{A}{B \rightarrow C} & \frac{B}{C} & (\lor e) \\
& \quad \frac{A \rightarrow B}{A} & (\rightarrow e) & \frac{A \rightarrow B}{B} & (\rightarrow e) & \frac{A \rightarrow B}{B \rightarrow A} & (\rightarrow i) & \frac{A \rightarrow B}{B} & (\rightarrow e) \\
& \frac{A \rightarrow \bot}{A} & (\rightarrow i) & \frac{A \rightarrow \bot}{B} & (\rightarrow e) & \frac{A}{\bot} & (\bot i) & \frac{A}{A \lor \bot} & (\lor e) & \frac{A}{A \lor \bot} & (\lor e) \\
& \frac{A}{A \lor \bot} & (\lor i) & \frac{\bot}{A} & (\bot e) & \frac{A}{A \lor \bot} & (TND)
\end{align*}
\]

Soundness and Completeness

- **Soundness**: If \( \Gamma \vdash P \) then \( \Gamma \models P \) in \( T \).
  - If the conclusion is provable from the premise, then the premise logically entails the conclusion.

- **Completeness**: If \( \Gamma \models P \) then \( \Gamma \vdash P \) in \( T \).
  - If the premise logically entails the conclusion, then the conclusion is provable from the premise.

  A premise logically entails the conclusion iff every interpretation that satisfies the premise also satisfies the conclusion.

Semantics Revisited

- **Denotational semantics**:
  - \( [TRUE] = true \)
  - \( [FALSE] = false \)
  - \( [AND(e_1, e_2)] = [e_1] \land [e_2] \)
Semantics Revisited

- **Small-Step Operational Semantics**
  
  \[
  \text{AND(\text{TRUE, } e) } \rightarrow e \quad (\text{AND1})
  \]
  
  \[
  \text{AND(\text{FALSE, } e) } \rightarrow \text{FALSE} \quad (\text{AND2})
  \]
  
  \[
  e_1 \rightarrow e'_1 \quad \text{AND}(e_1, e_2) \rightarrow \text{AND}(e'_1, e_2) \quad (\text{AND3})
  \]

- **Big-Step Operational Semantics**
  
  eval(\text{AND}(\text{e1, e2}))
  
  \[
  \begin{cases}
  \text{if (eval(e1))} & \text{return (eval(e2));} \\
  \text{else} & \text{return false;}
  \end{cases}
  \]
  
  \[
  \text{eval(AND(e1, e2))} := \text{eval(e1)} \ ? \text{eval(e2)} : \text{false};
  \]

Term Algebra

- Let \( \mathcal{F}^i \) be a set of function symbols with arity \( i \).
- We call functions with 0 arguments (nullary functions) constants.
- Let \( \mathcal{F} \) be the set of all function symbols: \( \mathcal{F} = \bigcup_{i=0}^{\infty} \mathcal{F}^i \).
- Let \( \mathcal{X} \) denote a set of (free) variable symbols.
- Given a set of function symbols \( \mathcal{F} \) and disjoint set of variable symbols \( \mathcal{X} \), the tuple \( \mathcal{T}(\mathcal{F}, \mathcal{X}) \) denotes a Term Algebra (or Term Language).

Example

- \( \mathcal{F} = \{0, \text{succ}(t)\}, \mathcal{X} = \{x\} \)
  
  \[
  \begin{align*}
  0 & \in \mathcal{T}(\mathcal{F}, \mathcal{X}) \\
  1 & \notin \mathcal{T}(\mathcal{F}, \mathcal{X}) \\
  \text{succ}(0) & \in \mathcal{T}(\mathcal{F}, \mathcal{X}) \\
  \text{succ}(\text{succ}(0)) & \in \mathcal{T}(\mathcal{F}, \mathcal{X}) \\
  \text{succ}(x) & \in \mathcal{T}(\mathcal{F}, \mathcal{X}) \\
  \text{succ}(y) & \notin \mathcal{T}(\mathcal{F}, \mathcal{X})
  \end{align*}
  \]
Ground Terms
- A term $t \in \mathcal{T} \cap \mathcal{X}$ containing no variables is called a Ground Term and we can write $t \in \mathcal{T} \cap \mathcal{X}$.

Matching
- Given two terms $t_1 \in \mathcal{T} \cap \mathcal{X}$ and $t_2 \in \mathcal{T} \cap \mathcal{X}$. The goal of matching is to decide whether the free variables in $t_1$ can be bound in such a way that the new term $t'_1 = t_2$ (syntactical equivalence).

Example:
- $t_1 = x = 0$
- $t_2 = x = \min(1, 2)$
- $t_1 = x = \min(1, 4)$
- $t_2 = x = \min(1, 4)$
- $t_1 = \min(x, y) = \min(\text{succ}(0), 5)$
- $t_2 = x = \text{succ}(0)$, $y = 5$

Substitution
- $\sigma$ is the substitution of the form
  - $\{ \text{var}_1 \rightarrow t_1 \} ... \{ \text{var}_n \rightarrow t_n \}$

  $\sigma(f(t_1, t_2, ..., t_n)) = f(\sigma(t_1), \sigma(t_2), ..., \sigma(t_n))$

  $\sigma(\text{var}) = \sigma(t)$ if $\exists t : \{ \text{var} \rightarrow t \} \in \sigma$

  $\sigma(\text{var}) = \text{var}$ if $\nexists t : \{ \text{var} \rightarrow t \} \in \sigma$

Example:
- $\sigma := \{ x \rightarrow \text{succ}(0) \}, \{ y \rightarrow \min(1, 2) \}$
  - $\min(x, 1) \rightarrow \min(\text{succ}(0, 1))$
  - $\min(x, y) \rightarrow \min(\text{succ}(0), \min(1, 2))$
  - $\min(z) \rightarrow \min(z)$

Equational Reasoning / Term Rewriting
- $\begin{align*}
  (s = t) & \in E \\
  E \vdash s = t
  \end{align*}$

- $\begin{align*}
  E \vdash s = t & \rightarrow E \vdash s = t \\
  E \vdash s = t & \rightarrow E \vdash s = u
  \end{align*}$

- $\begin{align*}
  E \vdash s = t & \rightarrow E \vdash \sigma(s) = \sigma(t) \\
  E \vdash f(s_1, ..., s_n) & = f(t_1, ..., t_n)
  \end{align*}$
The Matching Problem for Ground Terms

- Let
  \[ \text{match}(t_{\text{vars}}, t_{\text{ground}}) = \begin{cases} \sigma & \text{if } \exists \sigma : \sigma(t_{\text{vars}}) = t_{\text{ground}} \\ \text{fail} & \text{if } \exists \sigma \ldots \end{cases} \]
  be match expressions.

- We can now perform equational reasoning on (simple) terms as a matching problem of the form
  \[ \Theta = \{ \text{match}(t_1, s_1), \text{match}(t_2, s_2), \ldots \text{match}(t_n, s_n) \} \]

- What about
  \[ \min(x, y) ≡ \min(succ(z), y) = \min(succ(0), y) \]
  Subterms!

Unification

- Unification is a generalization of matching where both terms can contain variables

- Given two terms \( t_1 \in T(F, X) \) and \( t_2 \in T(F, X) \). The goal of unification is to discover a (most general) substitution \( \sigma \) such that \( \sigma(t_1) = \sigma(t_2) \)

- Most general unifier:
  \[ \Theta = \{ \text{unifies}(t_1, s_1), \text{unifies}(t_2, s_2), \ldots \text{unifies}(t_n, s_n) \} \]
  Most general unifier: Let \( \Theta \) be a MGU for \( t_1 = t_2 \), then every other valid unifier \( \Theta' \) can be expressed as \( \Theta' = \Theta \sigma' \).

Example

- \[ \min(x, y) ≡ \min(succ(0), y) \]
  \[ \{ \min(x, y) ≡ \min(succ(0), y) \} \]
  \[ \{ x ≡ succ(0), y ≡ y \} \quad (1) \]
  \[ \{ x ≡ succ(0) \} \quad (3) \]
  done: \( \Theta = \{ x \rightarrow succ(0) \} \)

- \[ \min(x, y) ≡ \min(succ(x), y) \]
  \[ \{ \min(x, y) ≡ \min(succ(x), y) \} \]
  \[ \{ x ≡ succ(x), y ≡ y \} \quad (1) \]
  \[ \{ x ≡ succ(x) \} \quad (3) \]
  fail \quad (6)
Example

- $\text{foo}(f(x,1),g(y)) \doteq \text{foo}(f(g(2),1),x)$
  - $\{\text{foo}(f(x,1),g(y)) \doteq \text{foo}(f(g(2),1),x)\}$
  - $\{f(x,1) \doteq f(g(2),1), g(y) \doteq x\}$ (1)
  - $\{x \doteq g(2), 1 \doteq 1, g(y) \doteq x\}$ (1)
  - $\{x \doteq g(2), g(y) \doteq x\}$ (3)
  - $\{g(y) \doteq g(2)\}$ (5) $\sigma = \{x \rightarrow g(2)\}$
  - $\{y \doteq 2\}$ (1)

- done $\Theta = \{x \rightarrow g(2), y \rightarrow 2\}$

- Unification: $\text{foo}(f(g(2),1),g(2))$