Denotational Semantics

A4M36TPJ, 2015/2016
Operational Semantics

• Usually well suited for reasoning about whole programs, less than ideal for reasoning about program fragments.

• Sometimes tends to overspecify the implementation of certain language features (e.g. evaluation order).

• Tends to put emphasis on syntax (rather than semantics) of the language.
Denotational Semantics

• Meaning of a program can be determined from the meaning of its parts.

• Unlike an operational semantics, a denotational semantics emphasizes what the meaning of a phrase is, not how the phrase is evaluated.
Denotational Semantics

- Consists of three parts:
  - Syntactic Algebra
  - Semantic Algebra
  - Meaning Function
Syntactic Algebra

- Describes **abstract syntax of the language**.
- Can be **specified by a grammar**.
Semantic Algebra

• Models the meaning of program phrases.

• Consists of a collection of semantic domains along with functions that manipulate these domains.

• The meaning of a program may be as simple as an element of a primitive semantic domain like Int, the domain of integers.

• More typically, the meaning of a program is an element of a function domain that maps context domains to an answer domain.
Context Domains

- **Denotational analogue** of state components.

- Model such entities as **name/value associations**, **contents of memory**, and **control information**.
Answer Domains

• Represent the **possible meanings of programs**.

• Include components that model **context information** that was transformed.
Meaning Function

- Maps elements of the **syntactic algebra** (i.e., nodes in the abstract syntax trees) to their meanings in the **semantic algebra**.

- Usually a collection of so-called **valuation** functions, one for each **syntactic domain** defined by the **abstract syntax** for the language.

- The function must be a **homomorphism** between the syntactic algebra and the semantic algebra.
Meaning Function

Suppose $M$ is a meaning function and $t$ is a node in an abstract syntax tree, with children $t_1, \ldots, t_k$. Then

$$(M t) = (f_t (M t_1) \ldots (M t_k))$$

where $f_t$ is a function that is determined by the syntactic class of $t$.

The reason to restrict meaning functions to homomorphisms is that their structure-preserving behavior greatly simplifies reasoning.
This is just the technical condition that constrains the meaning of an abstract syntax tree node to be determined from the meaning of its subnodes. It can be stated more formally as follows: Suppose $M$ is a meaning function and $t$ is a node in an abstract syntax tree, with children $t_1, \ldots, t_k$. Then $(M_t)$ must equal $(f_t (M_{t_1}) \ldots (M_{t_k}))$ where $f_t$ is a function that is determined by the syntactic class of $t$.

The reason to restrict meaning functions to homomorphisms is that their structure-preserving behavior greatly simplifies reasoning. This design choice accounts for a property of denotational semantics we call compositionality that is summarized by the motto “the meaning of the whole is composed out of the meaning of the parts.” A key consequence of compositionality is that the meaning of a program remains the same when one of its phrases is replaced by another phrase with the same meaning.

Compositionality also facilitates the implementation of programming languages. The core syntactic processing procedures of interpreters and translators based on denotational semantics have a natural recursive structure that mimics the recursive structure of the valuation functions and the abstract syntax trees they manipulate. For example, parser generators like Yacc [Joh75] allow grammar descriptions to specify semantic actions that are performed when an abstract syntax tree node is recognized during the parsing of a program. Typically, these
Expression Language
Syntax

\[
Expr ::= Num \mid \\
\triangle Expr \mid \\
Expr \circ Expr
\]
Semantics

Semantic domain: \( N \).

\[
[n] = n 
\]

\[
[\Delta e] = [\Delta]([e]) 
\]

\[
[\Delta] = \lambda x. - x \text{ (i.e. unary minus)}
\]

\[
[e_1 \odot e_2] = [\odot]([e_1], [e_2])
\]

\[
[\odot] = \lambda x, y. x + y \text{ (i.e. plus)}
\]
Syntax

\[ \text{Formula ::= true} \mid \false \mid \neg \text{Formula} \mid \text{Formula BinaryConnective Formula} \]

\[ \text{BinaryConnective ::= } \land \mid \lor \]
Semantics

Semantic domain: \{0, 1\}.

\[
\begin{align*}
[true] &= 1 \\
[false] &= 0 \\
[\neg f] &= [\neg][f] \\
[\neg] &= \lambda x.1 - x
\end{align*}
\]
Semantics

\[
[f_1 \ c \ f_2] = [c]([f_1], [f_2]) \quad (c \in BinaryConnective)
\]

\[
[\land] = \lambda x, y. x \cdot y
\]

\[
[\lor] = \lambda x, y. (x + y) - (x \cdot y)
\]
Regular Expressions
Syntax

```
RegExp ::= \emptyset \mid 
\epsilon \mid 
A \mid 
RegExp^* \mid 
RegExp BinOp RegExp

BinOp ::= + \mid \cdot 
```

where $A$ is a predefined set of characters (alphabet).
Semantics

Semantic domain: $A^*$.

$$[\emptyset] = \{\}$$

$$[\epsilon] = \{\epsilon\}$$

$$[a] = \{a\}$$

$$[e^*] = [*]([e])$$
Semantics

\[ \text{[*]} = \lambda L. \{ l_1 \cdot \ldots \cdot l_n | n \in N \land l_i \in L \} \quad \text{(note: including } \epsilon) \]

\[ [e_1 \circ e_2] = [o]([e_1], [e_2]) \quad (o \in \text{BinOp}) \]

\[ [+\] = \cup \]

\[ [\cdot] = \lambda A, B. \{ a \cdot b | a \in A \land b \in B \} \]
Lambda Calculus
Syntax

\[ Expr ::= X \mid \lambda X.Expr \mid Expr \; Expr \]

Semantic domain:

- \( J^\times K \)
- \( J \; K = \{ \; \} \)  
- \( J a K = \{ a \} \)  
- \( J^\times K = J^\times K (J e K) \)  
- \( J^\times K = L \; \{ l_1 \cdot \ldots \cdot l_n \mid n \in \mathbb{N} \} \) (note: including \( \varepsilon \) )

1.6 Denotational Semantics of Lambda Calculus

1.6.1 Syntax

\[ Expr ::= X \mid X.Expr \mid Expr \; Expr \]

1.6.2 Semantics

Semantic domains:

- \( \text{env} = \text{string} \)
- \( \text{fcn} = \text{function} \)

Notational conventions:

\[ e_2 \text{env}, f, f_2 \in \text{fcn}, E, E_0 \in \text{Expr} \]

\[ J x K = e.e(x) \]

\[ J x.E K = e.p.J E K (e[x \leftarrow p]) \]

\[ J E_1 E_2 K = e.(J E_1 K (e)) (J E_2 K (e)) \]
Semantics

Semantic domains: \( env = \text{string} \rightarrow \text{function}, \ fcn = fcn \rightarrow fcn; \) notational conventions \( e \in env, f, f' \in fcn, E, E' \in Expr \)

\[
[x] = \lambda e.e(x) \quad (26)
\]

\[
[\lambda x.E] = \lambda e.\lambda p.[E](e[x \mapsto p]) \quad (27)
\]

\[
[E_1 \ E_2] = \lambda e.([E_1](e))([E_2](e)) \quad (28)
\]