Denotational semantics
(An introduction)

Ralf Lämmel
Software Language Engineering Team
University of Koblenz-Landau
http://www.softlang.org/
Dependencies:
- Tree-based abstract syntax
- Basic of interpretation
- Big-step operational semantics
Denotational semantics
= **functional** semantics

**Semantic domains** = **function** types of meanings

\[
\begin{align*}
\text{store}T &= \text{store} \rightarrow \text{store} \quad // \text{Type of store transformation} \\
\text{store}O &= \text{store} \rightarrow \text{value} \quad // \text{Type of store observation}
\end{align*}
\]

**Semantic **functions** = mappings from syntax to semantics

\[
\begin{align*}
S & : \text{stmt} \rightarrow \text{store}T \quad // \text{Semantics of statements} \\
E & : \text{expr} \rightarrow \text{store}O \quad // \text{Semantics of expressions}
\end{align*}
\]

N.B.: The semantic functions are to be defined **compositionally**.
Denotational semantics obeys **compositionality**: define meaning of compound construct in terms of the meanings of constituents without reference to syntax.

For comparison:
Big-step operational semantics of imperative programs

\[
\begin{align*}
m_0 & \vdash s_1 \rightarrow m_1 \\
m_1 & \vdash s_2 \rightarrow m_2 \\
\hline
m_0 & \vdash \text{seq}(s_1, s_2) \rightarrow m_2 \\
\end{align*}
\]

\[
\begin{align*}
m & \vdash \text{if}(e, \text{seq}(s, \text{while}(e, s)), \text{skip}) \rightarrow m' \\
\hline
m & \vdash \text{while}(e, s) \rightarrow m' \\
\end{align*}
\]

N.B.: [SEQ] is compositional, but [WHILE] is not, as the meaning of a while-loop is defined here in terms of a constructed phrase (which, by the way, contains the while-loop under definition).
The compositional scheme

\[
\begin{align*}
storeT &= \text{store} \rightarrow \text{store} \quad // \text{Type of store transformation} \\
storeO &= \text{store} \rightarrow \text{value} \quad // \text{Type of store observation} \\
S : \text{stmt} &\rightarrow \text{storeT} \quad // \text{Semantics of statements} \\
\mathcal{E} : \text{expr} &\rightarrow \text{storeO} \quad // \text{Semantics of expressions} \\
S[\text{skip}] &= \text{skip} \\
S[\text{assign}(x,e)] &= \underline{\text{assign}} \ x \ (\mathcal{E}[e]) \\
S[\text{seq}(s_1,s_2)] &= \underline{\text{seq}} \ (S[s_1]) \ (S[s_2]) \\
S[\text{if}(e,s_1,s_2)] &= \underline{\text{if}} \ (\mathcal{E}[e]) \ (S[s_1]) \ (S[s_2]) \\
S[\text{while}(e,s)] &= \underline{\text{while}} \ (\mathcal{E}[e]) \ (S[s]) \\
\mathcal{E}[\text{intconst}(i)] &= \underline{\text{intconst}} \ i \\
\mathcal{E}[\text{var}(x)] &= \underline{\text{var}} \ x \\
\mathcal{E}[\text{unary}(o,e)] &= \underline{\text{unary}} \ o \ (\mathcal{E}[e]) \\
\mathcal{E}[\text{binary}(o,e_1,e_2)] &= \underline{\text{binary}} \ o \ (\mathcal{E}[e_1]) \ (\mathcal{E}[e_2])
\end{align*}
\]

N.B.: The underlined functions are the **semantic combinators**. They combine meanings — no syntax is involved.
Types of semantic combinators

\[
\begin{align*}
\text{skip} & : storeT \\
\text{assign} & : \text{string} \to \text{storeO} \to \text{storeT} \\
\text{seq} & : \text{storeT} \to \text{storeT} \to \text{storeT} \\
\text{if} & : \text{storeO} \to \text{storeT} \to \text{storeT} \to \text{storeT} \\
\text{while} & : \text{storeO} \to \text{storeT} \to \text{storeT} \\
\text{intconst} & : \text{int} \to \text{storeO} \\
\text{var} & : \text{string} \to \text{storeO} \\
\text{unary} & : \text{uo} \to \text{storeO} \to \text{storeO} \\
\text{binary} & : \text{bo} \to \text{storeO} \to \text{storeO} \to \text{storeO}
\end{align*}
\]

N.B.: The combinators combine meanings — no syntax is involved.

Well, ints, strings, and operators are hybrids.
Semi-formal definitions of semantic combinators

// The identity function for type store
\texttt{skip} \texttt{m} \; \texttt{=} \; \texttt{m}

// Pointwise store update
\texttt{assign} \; x \; f \; m \; \texttt{=} \; m[\texttt{x} \mapsto (f \; m)] \texttt{, if } f \; m \texttt{ is defined}

// Function composition for type \texttt{storeT}
\texttt{seq} \; f \; g \; m \; \texttt{=} \; g (f \; m)

// Select either branch for Boolean value
\texttt{if} \; f \; g \; h \; m \; \texttt{=} \; \begin{cases} g \; m, \text{if } f \; m = \texttt{true} \\ h \; m, \text{if } f \; m = \texttt{false} \\ \texttt{undefined, otherwise} \end{cases}

\textbf{N.B.: What about } \texttt{while}?
Suppose \textbf{while} \( fg = t \). Then, it would hold that:

\[
  t \equiv \text{if } f (\text{seq } g \ t) \text{ skip}
\]

Let us capture the expression as \( h \) parametrized in \( t \):

\[
h \ t = \text{if } f (\text{seq } g \ t) \text{ skip}
\]

Now consider the following progression of applications of \( h \):

\[
  \begin{align*}
    &h \text{ undefined} \\
    &h \ (h \text{ undefined}) \\
    &h \ (h \ (h \text{ undefined})) \\
    &\vdots
  \end{align*}
\]

The meaning of the while-loop can be understood as the repeated application of \( h \) to an undefined meaning while the number of repetitions is unbounded.

Fixed-point semantics II/II

The meaning of the while-loop is thus:

\[
\text{while } f \ g = \text{fix } h
\]

where

\[
h \ t = \text{if } f \ (\text{skip } g \ t) \ \text{skip}
\]

The fixed-point combinator is ‘defined’ as follows:

\[
\text{fix } k = k \ (\text{fix } k)
\]

N.B.: For such a fixed-point semantics to make sense, denotational semantics definitions need to meet some constraints studied in domain theory, which we skip here (and in the book as well).
Ingredients in need of encoding:

- Abstract syntax
- Semantic **domains**
- Semantic **functions**
- Semantic **combinators**
- **fix**
Denotational interpreter in Haskell: semantic domains and functions

```haskell
-- Results of expression evaluation
type Value = Either Int Bool
-- Stores as maps from variable ids to values
type Store = Map String Value
-- Store transformers (semantics of statements)
type StoreT = Store → Store
-- Store observers (semantics of expressions)
type StoreO = Store → Value

execute :: Stmt → StoreT
execute Skip = skip'
execute (Assign x e) = assign' x (evaluate e)
execute (Seq s1 s2) = seq' (execute s1) (execute s2)
execute (If e s1 s2) = if' (evaluate e) (execute s1) (execute s2)
execute (While e s) = while' (evaluate e) (execute s)

evaluate :: Expr → StoreO
evaluate (IntConst i) = intconst' i

...```

© 2017 Ralf Lämmel, Software Languages Team http://www.softlang.org/, University of Koblenz-Landau. All rights reserved.
Denotational interpreter in Haskell: semantic combinators

skip' :: StoreT
skip' = id
assign' :: String → StoreO → StoreT
assign' x f m = insert x (f m) m
seq' :: StoreT → StoreT → StoreT
seq' = flip (.)
if' :: StoreO → StoreT → StoreT → StoreT
if' f g h m = let Right v = f m in if v then g m else h m
while' :: StoreO → StoreT → StoreT
while' f g = fix h where h t = if' f (seq' g t) skip'
intconst' :: Int → StoreO
intconst' i _ = Left i

... 

The fixed-point combinator at hand

fix :: (a → a) → a
fix f = f (fix f)
Online resources

YAS’ GitHub repository contains all code.
YAS (Yet Another SLR (Software Language Repository))
http://www.softlang.org/yas
See here specifically:
https://github.com/softlang/yas/tree/master/languages/BIPL/Haskell

The Software Languages Book
http://www.softlang.org/book
The book discusses denotational semantics in more detail. This includes continuation style and program analysis by abstract interpretation atop denotational semantics.