1 Denotational Semantics

- Denotational semantics gives the meaning of a program in terms of mathematical objects: integers, booleans, tuples, and functions.

- The basic idea is to associate a mathematical object with each phrase of the language:
  - The phrase denotes the mathematical object.
  - The object is the denotation of the phrase.

- Definitions in Denotational Semantics are compositional:
  - The denotation of a language construct is defined in the denotations of its sub-phrases.

2 Meaning Brackets

- We use the emphatic (or Strachey or meaning) brackets to enclose pieces of abstract syntax, as in $[P]$.

- If $p$ is a phrase in the language, we define a mapping meaning such that $\text{meaning}[p]$ is a mathematical entity that models the semantics of $p$.

3 Meaning Brackets — Examples

- Addition in an imperative language:

  $evaluate \ [E_1 + E_2]_{sto} = compute(m, \text{plus}, n)$
  where $m = \text{evaluate} \ [E_1]_{sto}$
  $n = \text{evaluate} \ [E_2]_{sto}$
• The expressions 2 * 4, (5 + 3), 008, 8 all denote the same abstract object, 8:

\[
\begin{align*}
\text{meaning}[2 * 4] & = \text{meaning}[(5 + 3)] = \\
\text{meaning}[008] & = \text{meaning}[8] = 8
\end{align*}
\]

4 Denotational Specification

• A denotational specification consists of five parts:
  1. Syntactic categories
  2. Abstract production rules
  3. Semantic domains
  4. Semantic functions
  5. Semantic equations.

Example — A Language of Numerals

5 Denotational Specification

Syntactic Domains:

• \( N \) : Numeral
• \( D \) : Digit

Abstract Production Rules:

\[
\begin{align*}
\text{Numeral} & ::= \text{Digit} \mid \text{Numeral Digit} \\
\text{Digit} & ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{align*}
\]

Semantic Domains:

• Number = \{0,1,2,3,4,…\}

6 Denotational Specification...

Semantic Functions:

\[\text{value} : \text{Numeral} \rightarrow \text{Number}\]
\[\text{digit} : \text{Digit} \rightarrow \text{Number}\]
Semantic Equations:
\[
\begin{align*}
value[N, D] &= 10 \cdot value[N] + digit[D] \\
value[D] &= digit[D] \\
digit[0] &= 0 \\
& \vdots \\
digit[9] &= 9
\end{align*}
\]

7 Example

- Let’s see how the meaning of the phrase 65 would be derived:
\[
= 10 \cdot digit[5] + 5 \\
= 10 \cdot 6 + 5 \\
= 60 + 5 = 65
\]

8 Example...

- And the meaning of the phrase 088:
\[
value[088] = 10 \cdot value[00] + digit[8] \\
= 10 \cdot (10 \cdot value[0] + digit[0]) + 8 \\
= 10 \cdot (10 \cdot digit[0] + 0) + 8 \\
= 10 \cdot (10 \cdot 0 + 0) + 8 \\
= 8
\]

- Note that
\[
\]

The Semantics of Wren

9 Imperative Languages

- Wren is an imperative language.
- Programs consist of commands (statements).
- Commands alter a store, a global data structure simulating computer memory.
- The program updates the store until the required result is reached.
- The most important command is the assignment statement which modifies the store.
- Basic program control consists of sequencing, selection, and iteration (; if while).
10 Abstract Syntactic Domains

These are the abstract syntactic domains of Wren:

- **P**: Program
- **C**: Command
- **D**: Declaration
- **T**: Type
- **E**: Expression
- **O**: Operator
- **N**: Numeral
- **I**: Identifier

11 Abstract Syntax of Wrens

\[
\text{Program} ::= \text{program identifier is Declaration* begin Command end}
\]

\[
\text{Declaration} ::= \text{var Identifier : Type}
\]

\[
\text{Type} ::= \text{integer | boolean}
\]

\[
\text{Command} ::= \text{command | Command ; Command | variable := Expression | skip | read read Identifier | write Expression | while Expression do Command | if Expression then Command | if Expression then Command else Command}
\]

\[
\text{Expression} ::= \text{Numeral | Identifier | true | false | Expression Operator Expression | not ( Expression)}
\]

\[
\text{Operator} ::= \leq | \leq | \geq | \geq | \langle | \langle | \rangle | \rangle | + | * | / | \text{and} | \text{or}
\]

12 Semantic Domains of Wren

- **SV** (storable values) represents the values that may be placed in the store.
- **EV** (expressible or first-class values) represents the values that expressions can produce.

\[
\text{Integer} = \{\ldots -2, -1, 0, 1, 2, \ldots\}
\]

\[
\text{Boolean} = \{\text{true, false}\}
\]

\[
\text{EV} = \text{Integer + Boolean}
\]

\[
\text{SV} = \text{Integer + Boolean}
\]

\[
\text{Store} = \text{Identifier \to (SV + undefined)}
\]
13 Semantic Functions of Wren

- The value of an expression depends on the values of variables in the store:
  
  \[ evaluate : \text{Expression} \rightarrow (\text{Store} \rightarrow \text{EV}) \]

- Commands (statements) can modify the store:
  
  \[ execute : \text{Command} \rightarrow (\text{Store} \rightarrow \text{Store}) \]

- The meaning of a program is its resulting store:
  
  \[ meaning : \text{Program} \rightarrow \text{Store} \]

- The meaning of a number is handled elsewhere:
  
  \[ value : \text{Numeral} \rightarrow \text{EV} \]

## Semantic Equations

14 Commands

- The semantics of sequenced commands:
  
  \[ execute \{ C_1; C_2 \} = execute \{ C_2 \} \circ execute \{ C_1 \} \]
  
  This could also be written as
  
  \[ execute \{ C_1; C_2 \} = execute \{ C_2 \} (execute \{ C_1 \} \text{sto}) \]

- `skip` does not affect the store:
  
  \[ execute \{ \text{skip} \} \text{sto} = \text{sto} \]

- The assignment statement evaluates the right-hand-side and produces an updated store:
  
  \[ execute \{ I := E \} \text{sto} = \text{updateSto(sto, I, (evaluate \{ E \} \text{sto}))} \]

15 Commands...

- Conditionals:
  
  \[ execute \{ \text{if } E \text{ then } C \} \text{sto} = \text{if } p \text{ then} \]
  
  \[ execute \{ C \} \text{sto} \]

  \[ \text{else} \text{sto} \]

  \[ \text{where } p = \text{evaluate \{ E \} \text{sto}} \]

  \[ execute \{ \text{if } E \text{ then } C_1 \text{ else } C_2 \} \text{sto} = \text{if } p \text{ then} \]

  \[ execute \{ C_1 \} \text{sto} \]

  \[ \text{else } execute \{ C_2 \} \text{sto} \]

  \[ \text{where } p = \text{evaluate \{ E \} \text{sto}} \]
16 Commands...

- Loops:

execute while $E$ do $C$ sto = loop
where loop sto = if $p$ then
  loop(execute $[C]$ sto)
else sto
where $p$ = evaluate $[E]$ sto

- Here we have factored out the looping behavior into a special recursive function $\text{loop}$.

17 Expressions

evaluate $[I]$ sto = if $v$ = Undefined then error else $v$
where $v$ = applySto(sto, $I$)
evaluate $[N]$ sto = value $[N]$
evaluate [true] sto = true
evaluate [false] sto = false
evaluate $[E_1 + E_2]$ sto = compute(m, plus, n)
where $m$ = evaluate $[E_1]$ sto
  $n$ = evaluate $[E_2]$ sto

18 Expressions...

evaluate $[E_1 / E_2]$ sto = if $n = 0$ then error
  else compute(m, div, n)
where $m$ = evaluate $[E_1]$ sto
  $n$ = evaluate $[E_2]$ sto
evaluate $[E_1 < E_2]$ sto = if $n < m$ then true else false
where $m$ = evaluate $[E_1]$ sto
  $n$ = evaluate $[E_2]$ sto
evaluate $[E_1 \&\& E_2]$ sto = if $p$ then $q$ else false
where $p$ = evaluate $[E_1]$ sto
  $q$ = evaluate $[E_2]$ sto

A Haskell Prototype
19 Abstract Syntax

\texttt{type Num = Rational}
\texttt{data SV = IVal Num | BVal Bool | Undefined}
\texttt{type Identifier = String}
\texttt{data Operator = Add | Sub | Mul | Minus | Div | Not | Or | And | Lt | Gt | Eq | Ne | Le | Ge}
\texttt{data Expression = Id String | LitInt Num | TrueVal | FalseVal | Unary Operator Expression | Binary Expression Operator Expression}

20 Abstract Syntax...

\texttt{data Program = Prog [Declaration] Command}
\texttt{data Declaration = Var [Identifier] Type}
\texttt{data Type = IntType | BoolType}
\texttt{data Command = Skip | Assign String Expression | Read String | Write Expression | IfThen Expression Command | IfThenElse Expression Command Command | While Expression Command | Seq Command Command}

21 Expressions

\texttt{bcompute :: SV -> Operator -> SV -> SV}
\texttt{bcompute (IVal a) Add (IVal b) = (IVal (a + b))}
\texttt{bcompute (IVal a) Mul (IVal b) = (IVal (a * b))}
\texttt{bcompute (IVal a) Div (IVal b) =}
\texttt{\quad if b==0 then error "Division by 0"}
\texttt{\quad else (IVal (toRational(a / b)))}
\texttt{bcompute (IVal a) Sub (IVal b) = (IVal (a - b))}
\texttt{bcompute (BVal a) And (BVal b) = (BVal (a && b))}
\texttt{bcompute (BVal a) Or (BVal b) = (BVal (a || b))}
\texttt{bcompute (IVal a) Lt (IVal b) = (BVal (a < b))}
\texttt{bcompute (IVal a) Gt (IVal b) = (BVal (a > b))}
\texttt{bcompute (IVal a) Le (IVal b) = (BVal (a <= b))}
\texttt{bcompute (IVal a) Ge (IVal b) = (BVal (a >= b))}
bcompute (IVa l a) Eq (IVa l b) = (BVal (a == b))
bcompute (IVa l a) Ne (IVa l b) = (BVal (not (a == b)))

22 Expressions...

ucompute :: Operator -> SV -> SV
ucompute Minus (IVa l b) = (IVa l (- b))
ucompute Not (BVal b) = (BVal (not b))

23 Expressions...

evaluate :: Expression -> Store -> SV
evaluate (Id id) sto =
  if val == Undefined then val else val
  where val = applySto sto id
evaluate (LitInt n) sto = (IVa l n)
evaluate (TrueVal) sto = (BVal True)
evaluate (FalseVal) sto = (BVal False)
evaluate (Unary op r) sto = ucompute op n
  where n = evaluate r sto
evaluate (Binary l op r) sto = bcompute m op n
  where m = evaluate l sto
  n = evaluate r sto

24 Expressions — Examples

> s1
[("b",True),("a",5 % 1)]

> evaluate (Binary (LitInt 5) Add (LitInt 6)) s1
11 % 1

> evaluate (Binary (LitInt 5) Add (Id "a")) s1
10 % 1

> evaluate (Binary (Binary (LitInt 6) Mul (LitInt 2)) Add (Id "a")) s1
17 % 1

25 Commands

execute :: Command -> Store -> Store
execute (Skip) sto = sto
execute (Assign id e) sto = updateSto sto id (evaluate e sto)
ex e c l u x e (Seq c1 c2) sto = execute c2 (execute c1 sto)
ex e x e (I f t h e n b c) sto =
  if (evaluate b sto) == (BVal True) then
    execute c sto
  else sto
execute (IfThenElse b c1 c2) sto =
if (evaluate b sto) == (BVal True) then
  execute c1 sto
else execute c2 sto
execute (While b c) sto = loop sto
  where loop sto = if (evaluate b sto) == (BVal True) then
                  (loop (execute c sto)) else sto

26 Commands — Examples

> s1
[("b",True),("a",5 % 1)]

> execute (Assign "a" (LitInt 9)) s1
[("a",9 % 1),("b",True)]

> execute (IfThen (Unary Not (Id "b")) (Assign "a" (LitInt 9))) s1
[("b",True),("a",5 % 1)]

> execute (While (Binary (Id "a") Lt (LitInt 10)) (Assign "a" (Binary (Id "a") Add (LitInt 1)))) s1
[("a",10 % 1),("b",True)]

27 Store

type Store = [(Identifier,SV)]

emptySto:: Store
emptySto = []

updateSto:: Store -> Identifier -> SV -> Store
updateSto env id val =
  (id,val) : (filter (\ (x,_) -> not(id==x)) env)

applySto:: Store -> Identifier -> SV
applySto env id =
  snd (foldl (\ r c -> if id==(fst r) then r else c) ("",Undefined) env)

28 Store — Examples

s1 = updateSto (updateSto emptySto "a" (IVal 5)) "b" (BVal True)

> s1
[("b",True),("a",5 % 1)]

> applySto s1 "a"
5 % 1

> applySto s1 "b"
True

> applySto emptySto "a"
29  Readings and References


30  Acknowledgments