CS571

Notes 17
Denotational Semantics of an Imperative Language

A Language with a non-terminating command

- We cannot handle the (possibly non-terminating) loop right now
- Use a language with:
  - Assignment
  - Arithmetic
  - Booleans
  - Command sequence
  - Conditional
  - `diverge` which will stand in for a non-terminating loop

Syntax

- `P` ∈ Program
- `C` ∈ Command
- `E` ∈ Expression
- `B` ∈ Boolean-expr
- `I` ∈ Identifier
- `N` ∈ Numeral

`P ::= C`  
`C ::= C1;C2 | if B then C | if B then C else C2 | I := E | diverge`

`E ::= E1 + E2 | I | N`

`B ::= E1 = E2 | ~B`

Th Store Function

- Model the binding of identifiers to values with the `store` function:
  - The `store` function:
    - `Store`: `Id → Nat`
    - `Nat = ℕ`
    - `Id = identifiers` (e.g. `x, y, z` etc.)
    - As a domain:
      - Domain `Store` (`Id → Nat`)
    - Operations:
      - `newstore`: `Store`  
        - `newstore = λi.zero`
        - `access`: `λi.λs.s(i)`
        - `update`: `λid.λls.s(i) → Store → Store`  
        - `update = i.i.λn.λs.[i → n]s`

The non-terminating command
Lifted domains and strict functions

- To handle non-termination we add a distinguished element to a domain.
- \( \bot \), pronounced "bottom".
- \( A_{\bot} = A \cup \{\bot\} \)

Example of strict and non-strict function application

- Strict
  - \( (\lambda x. e) : A_{\bot} \rightarrow B_{\bot} \)
    - \( (\lambda x. e) \bot = \bot \)
    - \( (\lambda x. e) a = [a/x]e \)
- Non-strict
  - \( (\lambda x. \text{zero})((\lambda x. \text{one})\bot) \)
    - \( (\lambda x. \text{zero}) \bot = \bot \)
    - (using strictness twice)
    - (because this is a constant function that does not check for bottom)

Semantic algebras

- \( \text{Nat} = \mathbb{N} \)
  - Operations for constants, plus
- \( \text{Tr} = \mathbb{B} \)
  - Operations for true, false, equals, \( _{-} \rightarrow _{-} \)
- \( \text{Id} = \text{Identifier} = \{x, y, z, \ldots\} \)
  - No operations
- \( \text{Store} = \text{Id} \rightarrow \text{Nat} \)
  - Operations newstore, access, update

Valuation functions

- \( \text{N}: \text{Numeral} \rightarrow \text{Nat} \)
- \( \text{B}: \text{Boolean-expr} \rightarrow \text{Store} \rightarrow \text{Tr} \)
  - \( B[E_1 \equiv E_2] = \lambda s. E[E_1] s \text{ equals } E[E_2] s \)
  - \( B[\neg B] = \lambda s. \text{not } B[B] s \)
- \( \text{E}: \text{Expression} \rightarrow \text{Store} \rightarrow \text{Nat} \)
  - \( E[E_1 + E_2] = \lambda s. E[E_1] s \text{ plus } E[E_2] s \)
  - \( E[\text{[]}] = \lambda s. \text{access } \text{[]} s \)
  - \( E[N] = \lambda s. N[N] \)
**Valuation function for commands**

- **C**: Command → Store ⊥ → Store ⊥
  - **C[if B then C else C]** = λ s. B[B]s → C[C]s ⊸ s
  - **C[I := E] = λ s. update [I] (E[E]s) s**
  - **C[diverge] = λ s.⊥**

**NOTE**: these are all strict functions

**Lifted domains**

**diverge always returns bottom**

**Sample derivation**

- **Program**:
  - \(Z := 1;\) if \(A = 0\) then diverge;
  - \(Z := 3\)
  - **P[Z := 1; if A = 0 then diverge; Z := 3 ](two)**
    = let \(s' = C[Z] s\) in
      access [Z] s'

**Valuation function for programs**

- **P**: Program → Nat → Nat ⊥
  - **P[C] = λ n.let s = (update [A] n newstore) in**
    let \(s' = C[C]s\) in
      access [Z] s'

**Derivation continued**

- **Write** \(s' = (update [A] two newstore) = [[A] → two]newstore\)
- **Derivation is**
  - let \(s' = C[C]s\) in
    access [Z] s'
  = \(C[I := E] = λ s. update [I] (E[E]s) s\)
  - **s' = C[Z := 1; if A = 0 then diverge; Z := 3 ]s'**
    = (λ s. C[I := E] s) s'
  - **C[if A = 0 then diverge; Z := 3](C[Z := 1]s) s'**
  = C[I := E] s'}

**Not bottom**
Derivation continued

- C[Z := 1]s₁
  = (λs. update [Z] (E[s] s) s₁) s₁
  = update [Z] (E[s₁]) s₁
  = update [Z] (N[s₁]) s₁
  = update [Z] one s₁
  = [[Z] → one][[A] → two]newstore

Write this as s₂, the store for the conditional
- C[if A = 0 then diverge; Z := 3]s₂
  = (λs. C[Z := 3]s) s₂
  = C[Z := 3](λs. B[A = 0]s → C(diverge)[s ∩ s] s₂)
  = C[Z := 3]([B]A = 0) s₂ → C(diverge)[s₂ ∩ s₂]

Derivation finished

- So C[if A = 0 then diverge]s₂ = s₂

Then C[Z := 3]s₂
  = (λs. update [Z] (E[3] s) s) s₂
  = update [Z] (E[3]) s₂
  = update [Z] three s₂
  = [[Z] → three][[Z] → one][[A] → two]newstore
  = s₃

Final denotation is
  * let s' = [[[Z] → three][[Z] → one][[A] → two]newstore in access [Z] s'
  = [[[Z] → three][[Z] → one][[A] → two]newstore] [Z]
  = three

Derivation when A = 0

- The difference is in the store for Z := 3
  * C[if A = 0 then diverge; Z := 3]s₂
  = (λs. C[Z := 3]s) s₂
  = C[Z := 3](λs. B[A = 0]s → C(diverge)[s ∩ s] s₂)
  = C[Z := 3]([B]A = 0) s₂ → C(diverge)[s₂ ∩ s₂]
  = C[Z := 3](true → C(diverge)[s₂ ∩ s₂])
  = C[Z := 3](C(diverge)[s₂])
  = C[Z := 3](false)
  = (λs. update [Z] (E[3] s) s) ⊥
  = ⊥, because of strictness

- The final denotation is
  * let s' = ⊥ in access [Z] s'
  = ⊥, from the definition of let

However
- B[A = 0]s₂
  = (λs. E[s] s) equals E[0]s₂
  = E[s₂] equals E[0]s₂
  = access [A] s₂, equals zero
  = s₂ · (A) equals zero
  = two equals zero
  = false