Environments, Stores, and Interpreters
Overview

• As we study languages we will build small languages that illustrate language features

• We will use two tools
  – Observational semantic judgements
  – Small interpreters

• These tools convey the same information at different levels of detail.
To describe the machine’s operation, we give rules of inference that state when a judgment can be derived from judgments about sub-expressions.

The form of a rule is

\[
\frac{\text{premises}}{\text{conclusion}} \quad \text{(Name of rule)}
\]

We can view evaluation of the program as the process of building an inference tree.

This notation has similarities to axiomatic semantics: the notion of derivation is essentially the same, but the content of judgments is different.
ENvironments and Stores, Formally

- We write $E(x)$ means the result of looking up $x$ in environment $E$. (This notation is because an environment is like a function taking a name as argument and returning a meaning as result.)

- We write $E + \{x \mapsto v\}$ for the environment obtained from existing environment $E$ by extending it with a new binding from $x$ to $v$. If $E$ already has a binding for $x$, this new binding replaces it.

The domain of an environment, $\text{dom}(E)$, is the set of names bound in $E$. Analogously with environments, we’ll write

- $S(l)$ to mean the value at location $l$ of store $S$

- $S + \{l \mapsto v\}$ to mean the store obtained from store $S$ by extending (or updating) it so that location $l$ maps to value $v$.

- $\text{dom}(S)$ for the set of locations bound in store $S$.

Also, we’ll write

- $S - \{l\}$ to mean the store obtained from store $S$ by removing the binding for location $l$. 

**Evaluation Rules (1)**

\[
\frac{l = E(x) \quad v = S(l)}{\langle x, E, S \rangle \Downarrow \langle v, S \rangle} \quad \text{(Var)}
\]

\[
\frac{\langle i, E, S \rangle \Downarrow \langle i, S \rangle}{\quad \text{(Int)}}
\]

\[
\frac{\langle e_1, E, S \rangle \Downarrow \langle v_1, S' \rangle \quad \langle e_2, E, S'' \rangle \Downarrow \langle v_2, S''' \rangle}{\langle (+ e_1 e_2), E, S \rangle \Downarrow \langle v_1 + v_2, S''' \rangle} \quad \text{(Add)}
\]

\[
\frac{\langle e_1, E, S \rangle \Downarrow \langle v_1, S' \rangle \quad l \notin \text{dom}(S')}{\langle e_2, E + \{x \mapsto l\}, S' + \{l \mapsto v_1\} \rangle \Downarrow \langle v_2, S''' \rangle} \quad \text{(Local)}
\]

\[
\frac{\langle e, E, S \rangle \Downarrow \langle v, S' \rangle \quad l = E(x)}{\langle (: = x \; e), E, S \rangle \Downarrow \langle v, S' + \{l \mapsto v\} \rangle} \quad \text{(Assign)}
\]
\[
\begin{align*}
\langle e_1, E, S \rangle \downarrow \langle v_1, S' \rangle & \quad v_1 \neq 0 & \langle e_2, E, S' \rangle \downarrow \langle v_2, S'' \rangle & \quad (\text{If-nzero}) \\
& \quad (\text{if } e_1 \ e_2 \ e_3, E, S) \downarrow \langle v_2, S''' \rangle \\
\langle e_1, E, S \rangle \downarrow \langle 0, S' \rangle & \quad \langle e_3, E, S' \rangle \downarrow \langle v_3, S'' \rangle & \quad (\text{If-zero}) \\
& \quad (\text{if } e_1 \ e_2 \ e_3, E, S) \downarrow \langle v_3, S''' \rangle
\end{align*}
\]

\[
\begin{align*}
\langle e_1, E, S \rangle \downarrow \langle v_1, S' \rangle & \quad v_1 \neq 0 & \langle e_2, E, S' \rangle \downarrow \langle v_2, S'' \rangle & \quad (\text{While-nzero}) \\
& \quad (\text{while } e_1 \ e_2, E, S''') \downarrow \langle v_3, S''' \rangle \\
& \quad (\text{while } e_1 \ e_2, E, S) \downarrow \langle v_3, S''' \rangle
\end{align*}
\]

\[
\begin{align*}
\langle e_1, E, S \rangle \downarrow \langle 0, S' \rangle & \quad (\text{While-zero}) \\
& \quad (\text{while } e_1 \ e_2, E, S) \downarrow \langle 0, S' \rangle \\
\end{align*}
\]
The structure of the rules guarantees that at most one rule is applicable at any point.

The store relationships constrain the order of evaluation.

If no rules are applicable, the evaluation gets stuck; this corresponds to a runtime error in an interpreter.

We can view the interpreter for the language as implementing a bottom-up exploration of the inference tree. A function like

\[
\text{Value eval(Exp e, Env env) \{ 
\ldots 
\}}
\]

returns a value \( v \) and has side effects on a global store such that

\[
\langle e, env, \text{store before} \rangle \Downarrow \langle v, \text{store after} \rangle
\]

The implementation of \texttt{eval} dispatches on the syntactic form of \( e \), chooses the appropriate rule, and makes recursive calls on \texttt{eval} corresponding to the premises of that rule.

Question: how deep can the derivation tree get?
Interpreters

• Programs that detail the same issues as an observational semantics
  – Operations on environments and stores
    • $E(x)$
    • $E\{x \rightarrow v\}$
    • $\text{Dom}(E)$
    • $S(l)$
    • $S\{l \rightarrow v\}$
    • $\text{Dom}(S)$
Values

type Addr = Int

data Value
  = IntV Int    -- An Int
  | PairV Addr  -- Or an address
    -- into the heap
Tables in hw3.hs

• Tables are like dictionaries storing objects indexed by a key.

– A table maps keys to objects

data Table key object = Tab [(key,object)]
type Env a = Table String a  -- A Table
  where the key is a String
Lookup and Searching Tables

-- When searching an Env one returns a Result
data Result a = Found a | NotFound

search :: Eq key => key -> [(key, a)] -> Result a
search x [] = NotFound
search x ((k,v):rest) =
    if x==k then Found v else search x rest

-- Search a Table
lookUp :: Eq key => Table key a -> key -> Result a
lookUp (Tab xs) k = search k xs
Updating Tables

• Update is done by making a new changed copy

```haskell
-- update a Table at a given key.
update n u ((m,v):rest)
    | n==m = (m,u):rest
update n u (r:rest) = r : update n u rest
update n u [] = error
    ("Unknown address: "+show n++
     " in update")
```
Environments in hw3.hs

-- A Table where the key is a String
type Env a = Table String a

-- Operations on Env
emptyE = Tab []               -- Ø

extend key value (Tab xs) =  -- E+{x → v}
    Tab ((key,value):xs)

-- E+{x₁ → v₁, x₂ → v₂, ..., xₙ → vₙ}
push pairs (Tab xs) = Tab(pairs ++ xs)
Stores and Heaps

• In language E3, the store is implemented by a heap. Heaps are indexed by addresses (int)

```haskell
type Heap = [Value]

-- State contains just a Heap
data State = State Heap

-- Access the State for the Value
-- at a given Address    S(n)
access n (State heap) = heap !! n
```

(list !! n) is the get element at position n. The first element is at position 0.
Allocating on the heap

\[ S + \{ l \rightarrow v \} \]

-- Allocate a new location in the heap. Initialize it with a value, and return its Address and a new heap.

\[
\text{alloc} :: \text{Value} \rightarrow \text{State} \rightarrow (\text{Addr},\text{State})
\]

\[
\text{alloc value (State heap) } = \text{ (addr,State (heap ++ \[ value \])})
\]

where \( \text{addr} = \text{length heap} \)

Note that allocation creates a new copy of the heap with one more location
Multiple allocations

\[
\text{(fun f (x y z) (+ x (* y z)))}
\]
\[
(@ f 3 5 8)
\]

• We need to create 3 new locations on the heap and note where the formal parameters \((x, y, z)\) are stored

• \(E \{x \rightarrow l_1, y \rightarrow l_2, z \rightarrow l_3\}\)

• \(S \{l_1 \rightarrow 3, l_2 \rightarrow 5, l_3 \rightarrow 8\}\)
bind :: [String] -> [Value] -> State -> ([(Vname,Addr)],State)
bind names objects state =
    loop (zip names objects) state
where loop [] state = ([],state)
    loop ((nm,v):more) state =
        ((nm,ad):xs,state3)
    where (ad,state2) = alloc v state
        (xs,state3) = loop more state2
Example

bind \([a, b, c]\) 
\([\text{IntV } 3, \text{IntV } 7, \text{IntV } 1]\) 
\((\text{State } [\text{IntV } 17])\)

• returns the pair

\(( [(a, 1), (b, 2), (c, 3)] , \text{State } [\text{IntV } 17, \text{IntV } 3, \text{IntV } 7, \text{IntV } 1] \) \)
Heap update

• Makes a new copy of the heap with a different object at the given address.

```haskell
-- Update the State at a given Address
stateStore addr u (State heap) =
  State (update addr heap)
where update 0 (x:xs) = (u:xs)
update n (x:xs) = x : update (n-1) xs
update n [] =
  error ("Address "++show addr++
          " too large for heap.")
```
Example

Allocate 1 (St [IntV 3, IntV 7] )

Returns

(2, St [IntV 3, IntV 7, IntV 1] )
The interpreter

• It implements the observational rules but has more detail.
• It also adds the ability to trace a computation.
interpE :: Env (Env Addr,[Vname],Exp)  -- The function name space
   -> Env Addr                    -- the variable name space
   -> State                      -- the state, a heap
   -> Exp                        -- the Exp to interpret
   -> IO(Value,State)            -- (result,new_state)

interpE funs vars state exp =
   (traceG vars) run state exp where
   run state (Var v) =
      case lookUp vars v of
         Found addr ->
            return(access addr state,state)
         NotFound ->
            error ("Unknown variable: "++v++" in lookup.")

   --- ... many more cases

\[ l = E(x) \quad v = S(l) \]
\[ \langle x, E, S \rangle \Downarrow \langle v, S \rangle \]  (Var)
Constant and assignment case

run state (Int n) = return(IntV n,state)
run state (Asgn v e) =
  do { (val,state2) <- interpE funs vars state e
       ; case lookUp vars v of
         Found addr ->
           return(val,stateStore addr val state2)
         NotFound -> error
           ("Unknown variable: " ++
            v ++" in assignment.") } }

(\langle i, E, S \rangle \downarrow \langle i, S \rangle) (Int)

\langle e, E, S \rangle \downarrow \langle v, S' \rangle \quad l = E(x)
\frac{\langle (:= x e), E, S \rangle \downarrow \langle v, S' + \{ l \mapsto v \} \rangle}{(Assgn)}
Notes on pairs

• Pairs are allocated in consecutive locations on the heap

run state (Pair x y) =
  do { (v1,s1) <- interpE funs vars state
       ; (v2,22) <- interpE funs vars s1 y
       ; let (a1,s3) = alloc v1 s2
           (a2,s4) = alloc v2 s3
       ; return(PairV a1,s4)}
Runtime checking of errors

• Numeric operations (+, *, <=, etc) only operate on (IntV n) and must raise an error on (PairV a)

```haskell
class Numeric st name fun (v@(PairV _)) _ = error ("First arg of " ++ name ++ " is not an Int. " ++ showV (v, st))
class Numeric st name fun _ (v@(PairV _)) = error ("Second arg of " ++ name ++ " is not an Int. " ++ showV (v, st))
```