On the Structure of Interpreters

Operational Semantics

- Operational semantics describe how a language operates.
- Given by a set of inference rules.
- Use a step relation $\Downarrow$

$$\frac{\langle e, E, S\rangle \Downarrow \langle v, S'\rangle \quad l = E(x)}{\langle (:= \ x \ \ e), E, S\rangle \Downarrow \langle v, S' + \{l \mapsto v\}\rangle} \ (\mathrm{Assgn})$$

Recursive calls

Helper relations

$$\frac{\langle e, E, S\rangle \Downarrow \langle v, S'\rangle \qquad l = E(x)}{\langle (\mathtt{:=}\ \ x\ \ e), E, S\rangle \Downarrow \langle v, S' + \{l \mapsto v\}\rangle}\ (\text{Assgn})$$

syntax

parameters

result

state

1. The state of the semantics are those things that appear on both sides of the stepping relation
2. Inputs appear only on the left
3. Outputs appear on the right

State

- The state abstracts those things that change over time as the program executes
  - For example the heap
- The state might contain zero, one, or many parts
  - The heap, the stack, the current handlers etc.

Interpreters

- Interpreters have more detail than operational semantics.
- They are always recursive over the syntax of the language
- Some things are only inputs, because they remain constant. E.g. the environment that maps names to locations, since the location never changes
- The state appears as both an input and an output. The output captures the change over time

Example 1

- The interpreter for the stack machine from HW1

- step:: Stack Int -> Instr -> Stack Int

syntax

state

Example 2

- The interpreter from HW3

```
interpE :: Env (Env Addr,[Vname],Exp)
        -> Env Addr
        -> State
        -> Exp
        -> IO(Value,State)
```

inputs

syntax

result

State

Example 3

- The Exception machine from HW #4

```
interpE :: Env (Stack,[Vname],Exp)
        -> Env Address
        -> State
        -> Exp
        -> IO(Value,State)

data State
  = State Stack Heap
  | Exception State Fname [Value]
```

Here the State is much more complicated, it even comes in two modes: Normal and Exceptional. Normal states have two components, a Stack and a Heap

Operations on States

- Operations on states propagate exceptional state.

```
delta f g (State st hp) = State (f st) (g hp)
delta f g (Exception st fname vs) =
      Exception st fname vs

alloc :: Value -> State -> (Address,State)
alloc v state | exceptional state =
     (error "Exception State in alloc",state)
alloc v state = (HAddr addr,deltaHeap f state)
   where addr = length (heap state)
         f heap = heap ++ [(v)]
```

Threading

```
run state (Add  x y) =
  do { (v1,state1)<- interpE funs vars state x
     ; (v2,state2)<- interpE funs vars state1 y
     ; return(numeric "+" (+) v1 v2,state2) }
```

Special Casing the state

- The interpreter may do special things on certain kinds of state

```
interpE funs vars state exp = traceG run
  state exp where
    run (state@(Exception s f vs)) exp
        = return(Bad,state)
    run state (Int n)
        = return(IntV n,state)
    run state (Char c)
        = return(CharV c,state)
```

…

Summary

- The shape and operations on the State of an interpreter can be used to encode many kinds of language features.
  - Assignment
  - Allocation
  - Exceptions
  - Continuations