Computer Science 131
Programming Languages

October 3, 2000

Semantics for References

Adding References to NQSML

- So far we have used purely *language-based* models of program execution
  - Every intermediate state of a program can be represented as another program
  - Advantages:
    - Relatively direct
    - Don't have to worry about irrelevant machine details (memory layout, data representations)

Adding References to NQSML

- Now we need to maintain a notion of memory, memory locations, aliasing, etc.
  - Do we need to start thinking about bits, bytes, and data representation?
  - Or is there a middle ground?

An Abstract Notion of Memory

- Postulate an (infinite) set of *locations*
  - Denoted l_1, l_2, l_3, ...
  - Each location can hold a single value
    - An integer, or a function, or a pair of values, …
- Memory is a kind of "environment"
  - Associates locations with the values they contain
  - For example,
    - l_1=**<u>3</u>**, l_6 **=(<u>3</u>,<u>4</u>)**, l_{17} **=(<u>3</u>,(<u>4</u>,(<u>5</u>,<u>tt</u>)))**

Programs and Expressions

- We extend the notion of expressions by adding unit and locations as new values

```
v ::= ...
    | ()
    | l
```

and by adding three new expression forms

```
e ::= ...
    | mkref e
    | get e
    | set(e1,e2)
```

Example Expressions

```
let r be mkref(0) in
    let y = get(r) in
         set(x, y+1)
```

set(l_3**, get(**l_{12}**))**

Programs

- A program then consists of
  - a memory M (often called a "store")
  - an expression **e**

```
p ::= (M, e)
```

- Recall that M associates values with locations
- Evaluation is defined on *programs*

$p \rightarrow p'$ $\qquad\qquad\qquad$ $(M,e) \rightarrow (M',e')$

Dynamic Semantics

- All the language constructs we have seen before don't affect memory themselves.

$$\frac{}{(\mathrm{M},\mathbf{\underline{n}_1+\underline{n}_2}) \rightarrow (\mathrm{M},\mathbf{\underline{n_1+n_2}})}$$

- But their subexpressions might

$$\frac{(\mathrm{M},\mathbf{e_1}) \rightarrow (\mathrm{M}',\mathbf{e_1}')}{(\mathrm{M},\mathbf{e_1+e_2}) \rightarrow (\mathrm{M}',\mathbf{e_1'+e_2})}$$

$$\frac{(\mathrm{M},\mathbf{e_2}) \rightarrow (\mathrm{M}',\mathbf{e_2}')}{(\mathrm{M},\mathbf{v_1+e_2}) \rightarrow (\mathrm{M}',\mathbf{v_1+e_2'})}$$

Dynamic Semantics

- All the language constructs we have seen before don't affect memory themselves.

$$\frac{}{(\mathrm{M},\underline{\mathbf{n}}_1+\underline{\mathbf{n}}_2) \rightarrow (\mathrm{M},\underline{\mathbf{n_1+n_2}})}$$

- But their subexpressions might

$$\frac{(\mathrm{M},\mathbf{e}_1) \rightarrow (\mathrm{M}',\mathbf{e}_1')}{(\mathrm{M},\mathbf{e}_1+\mathbf{e}_2) \rightarrow (\mathrm{M}',\mathbf{e}_1'+\mathbf{e}_2)}$$

$$\frac{(\mathrm{M},\mathbf{e}_2) \rightarrow (\mathrm{M}',\mathbf{e}_2')}{(\mathrm{M},\mathbf{v}_1+\mathbf{e}_2) \rightarrow (\mathrm{M}',\mathbf{v}_1+\mathbf{e}_2')}$$

Dynamic Semantics

- Similar changes to the other rules
  - **`if-then-else`**?

Dynamic Semantics

$$\frac{(M,\mathbf{e}) \rightarrow (M',\mathbf{e'})}{(M,\mathbf{mkref\ e}) \rightarrow (M',\mathbf{mkref\ e'})}$$

$$\frac{(M,\mathbf{e}) \rightarrow (M',\mathbf{e'})}{(M,\mathbf{get\ e}) \rightarrow (M',\mathbf{get\ e'})}$$

$$\frac{(M,\mathbf{e_1}) \rightarrow (M',\mathbf{e_1'})}{(M,\mathbf{set(e_1,e_2)}) \rightarrow (M',\mathbf{set(e_1',e_2)})}$$

$$\frac{(M,\mathbf{e_2}) \rightarrow (M',\mathbf{e_2'})}{(M,\mathbf{set(v_1,e_2)}) \rightarrow (M',\mathbf{set(v_1,e_2')})}$$

Dynamic Semantics

$$\frac{l \notin \text{dom } M}{(M, \texttt{mkref v}) \rightarrow ((M, l\texttt{=v}), l)}$$

$$\frac{}{(M, \texttt{get } l) \rightarrow (M, M(l))}$$

$$\frac{}{(M, \texttt{set}(l, \texttt{v})) \rightarrow ((M, l\texttt{=v}), \texttt{()})}$$

Example

- Show the steps required to evaluate

```
let r be mkref(0) in
   let x be set(r, get(r) + 1) in
       get(r)
```

starting with an empty memory.

Static Semantics

- Two new types

```
t ::= ...
    | Unit
    | t Ref
```

Static Semantics

$$\frac{}{\Gamma \vdash \mathbf{()} : \mathbf{Unit}}$$

$$\frac{\Gamma \vdash \mathbf{e} : \mathbf{t}}{\Gamma \vdash \mathbf{mkref\ e} : \mathbf{t\ Ref}}$$

$$\frac{\Gamma \vdash \mathbf{e} : \mathbf{t\ Ref}}{\Gamma \vdash \mathbf{get\ e} : \mathbf{t}}$$

$$\frac{\Gamma \vdash \mathbf{e_1} : \mathbf{t\ Ref} \qquad \Gamma \vdash \mathbf{e_2} : \mathbf{t}}{\Gamma \vdash \mathbf{set(e_1,e_2)} : \mathbf{Unit}}$$

Type Safety

- We now have all the rules necessary to typecheck expressions that do not explicitly mention locations
  - That is, all programs we expect a programmer to write
- Can also evaluate all programs

- This may be enough for a language definition

Type Safety

- But can we still prove type safety?
  - The approach using preservation and progress can't be applied yet, because need to typecheck all intermediate states of a program
  - Consider

(Æ**, get(mkref 1)+2)** **(*** *well-typed* ***)**

$\rightarrow$ **(***l***=1, (get** *l***)+2)** **(* ??? *)**

$\rightarrow$ **(***l***=1, 1+2)** **(*** *well-typed* ***)**

$\rightarrow$ **(***l***=1, 3)** **(*** *well-typed* ***)**

Type Safety

- Intuition
  - Programs ought to step from "good programs" to "good programs"
  - The program **(*l*=1, (get *l*)+2)** is ok
  - The program **(*l*=tt, (get *l*)+2)** is not

- How can we make this precise?

Solution

- A type environment for the store
  - If the store has l**=1**, we want to remember that location l contains an integer.
    - Thus

      (get l**)+2**

      is ok
    - So is:

      set(l**,4)**
    - But not:

      (get l**)(2)**

Store Typing

- Associates types with locations
  - For example,

    l_1**:Int**, l_6 **:Int*Int**
  - Denoted Δ
- Typing Judgment must change

$$\Delta,\Gamma \vdash \mathbf{e} : \mathbf{t}$$

- Only one rule actually uses the store typing

$$\frac{}{\Delta,\Gamma \vdash l \;:\; \Delta(l)\ \mathbf{Ref}}$$

Type Soundness

- A program **`(M,e)`** is well-formed if there is a store typing Δ such that
  - Every location l in **`M`** contains a value of type $\Delta(l)$
  - There is a proof $\Delta,\varnothing \vdash$ **`e : t`**
- Claim:
  - If **`p`** is well-formed and **`p`**$\rightarrow$**`p'`** then so is **`p'`**
  - If p is well-formed then either it is of the form **`(M,v)`** [or **`(M,fail)`**] or else **`p`**$\rightarrow$**`p'`**

Assign-once variables

- Specification:

```
type 'a oneshot
exception Oneshot
val  new : unit -> 'a oneshot
val  get : 'a oneshot -> 'a
val  set : 'a oneshot * 'a -> unit
```

Assign-once variables

```
type 'a oneshot = 'a option ref
exception Oneshot

(* new : unit -> 'a oneshot *)
fun new () = ???
```

Assign-once variables

```
type 'a oneshot = 'a option ref
exception Oneshot

(* new : unit -> 'a oneshot *)
fun new () = ref NONE
```

Assign-once variables

```
type 'a oneshot = 'a option ref
exception Oneshot
fun new () = ref NONE

(* get : 'a oneshot -> 'a *)
fun get os = ???
```

Assign-once variables

```
type 'a oneshot = 'a option ref
exception Oneshot
fun new () = ref NONE

(* get : 'a oneshot -> 'a *)
fun get os =
  (case !os of
     NONE   => raise Oneshot
   | SOME v => v)
```

Assign-once variables

```
type 'a oneshot = 'a option ref
exception Oneshot
fun new () = ref NONE

(* get : 'a oneshot -> 'a *)
fun get (ref NONE)     = raise Oneshot
  | get (ref (SOME v)) = v
```

Assign-once variables

```
type 'a oneshot = 'a option ref
exception Oneshot
fun new () = ref NONE
fun get (ref NONE)     = raise Oneshot
  | get (ref (SOME v)) = v
(* set : 'a oneshot * 'a -> unit *)
fun set (os as ref NONE, v) = ???
  | set (os as ref (SOME v'), v) = ???
```

Assign-once variables

```
type 'a oneshot = 'a option ref
exception Oneshot
fun new () = ref NONE
fun get (ref NONE)      = raise Oneshot
  | get (ref (SOME v)) = v
(* set : 'a oneshot * 'a -> unit *)
fun set (os as ref NONE, v) =
      (os := SOME v)
  | set (os as ref (SOME v'), v) =
      raise Onshot
```

Quick Quiz

```
val x1 : int list = [1,2,3]
val _ = f1(x1)
   length(x1) = ???        hd(x1) = ???

val x2 : int list ref = ref [1,2,3]
val _ = f2(x2)
   length(!x2) = ???      hd(!x2) = ???

val x3 : int ref list= [ref 1, ref 2, ref 3]
val _ = f3(x3)
   length(x3) = ???       !hd(x3) = ???
```

Quick Quiz

```
val x1 = [1,2,3]
val _ = f1(x1)
   length(x1) = 3          hd(x1) = 1

val x2 = ref [1,2,3]
val _ = f2(x2)
   length(!x2) = Unknown  !hd(x1) = Unknown
                            (may raise exception)
val x3 = [ref 1, ref 2, ref 3]
val _ = f3(x3)
   length(x3) = 3         !hd(x1) = Unknown
                            (but no exception)
```

Pure vs. Imperative Interfaces

- Persistent environments

```
type 'a env
val empty : 'a env
val insert: 'a env * string * 'a -> 'a env
val lookup: 'a env * string -> 'a option
```

- Ephemeral environments

```
type 'a env
val empty : unit -> 'a env
val insert: 'a env * string -> unit
val lookup: 'a env * string -> 'a
val copy  : 'a env -> 'a env
```