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 $l_1, l_2, l_3, \ldots$
  - 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,
    • $l_1=3, \ l_6=(3,4), \ l_{17}=(3,(4,(5,tt)))$
Programs and Expressions

- We extend the notion of expressions by adding unit and locations as new values
  \[ v ::= \ldots \]
  \[ | () \]
  \[ | l \]
  and by adding three new expression forms
  \[ e ::= \ldots \]
  \[ | \text{mkref } e \]
  \[ | \text{get } e \]
  \[ | \text{set}(e_1,e_2) \]
Example Expressions

\begin{align*}
\text{let } r & \text{ be mkref}(0) \text{ in} \\
& \quad \text{let } y = \text{get}(r) \text{ in} \\
& \quad \quad \text{set}(x, y+1) \\
\text{set}(l_3, \text{get}(l_{12}))
\end{align*}
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

\[ p \rightarrow p' \quad (M,e) \rightarrow (M',e') \]
Dynamic Semantics

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

\[
(M, n_1 + n_2) \rightarrow (M, n_1 + n_2)
\]

• But their subexpressions might

\[
\begin{align*}
(M, e_1) & \rightarrow (M', e_1') \\
(M, e_1 + e_2) & \rightarrow (M', e_1' + e_2) \\
(M, e_2) & \rightarrow (M', e_2') \\
(M, v_1 + e_2) & \rightarrow (M', v_1 + e_2')
\end{align*}
\]
Dynamic Semantics

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

\[
(M, n_1+n_2) \rightarrow (M, n_1+n_2)
\]

• But their subexpressions might

\[
(M, e_1) \rightarrow (M', e_1')
\]

\[
(M, e_1+e_2) \rightarrow (M', e_1'+e_2)
\]

\[
(M, e_2) \rightarrow (M', e_2')
\]

\[
(M, v_1+e_2) \rightarrow (M', v_1+e_2')
\]
Dynamic Semantics

• Similar changes to the other rules
  – if–then–else?
Dynamic Semantics

\[
\begin{aligned}
(M, e) & \rightarrow (M', e') \\
(M, \text{mkref } e) & \rightarrow (M', \text{mkref } e') \\
(M, e) & \rightarrow (M', e') \\
(M, \text{get } e) & \rightarrow (M', \text{get } e') \\
(M, e_1) & \rightarrow (M', e_1') \\
(M, \text{set}(e_1, e_2)) & \rightarrow (M', \text{set}(e_1', e_2)) \\
(M, e_2) & \rightarrow (M', e_2') \\
(M, \text{set}(v_1, e_2)) & \rightarrow (M', \text{set}(v_1, e_2'))
\end{aligned}
\]
Dynamic Semantics

\[ l \notin \text{dom } M \]

\[ (M, \text{mkref } v) \rightarrow ((M, l=v), l) \]

\[ (M, \text{get } l) \rightarrow (M, M(l)) \]

\[ (M, \text{set}(l, v)) \rightarrow ((M, l=v), ()) \]
Example

• Show the steps required to evaluate

\[
\text{let } r \text{ be mkref(0) in}
\]

\[
\text{let } x \text{ be set}(r, \text{get}(r) + 1) \text{ in}
\]

\[
\text{get}(r)
\]

starting with an empty memory.
Static Semantics

• Two new types

\[
\begin{align*}
t & ::= \ldots \\
    & \mid \text{Unit} \\
    & \mid t \text{ Ref}
\end{align*}
\]
Static Semantics

\[ \Gamma \vdash () : \text{Unit} \]

\[ \Gamma \vdash e : t \]

\[ \Gamma \vdash \text{mkref } e : t \text{ Ref} \]

\[ \Gamma \vdash e : t \text{ Ref} \]

\[ \Gamma \vdash \text{get } e : t \]

\[ \Gamma \vdash e_1 : t \text{ Ref} \quad \Gamma \vdash e_2 : t \]

\[ \Gamma \vdash \text{set}(e_1,e_2) : \text{Unit} \]
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
    
    $$(\emptyset, \text{get}(\text{mkref } 1)+2) \quad (*\text{ well-typed }*)$$
    $$\rightarrow (l=1, (\text{get } l)+2) \quad (*\text{ ??? }*)$$
    $$\rightarrow (l=1, 1+2) \quad (*\text{ well-typed }*)$$
    $$\rightarrow (l=1, 3) \quad (*\text{ well-typed }*)$$
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
      \[(\text{get } l) + 2\]
      is ok
    • So is:
      \[\text{set}(l, 4)\]
    • But not:
      \[(\text{get } l)(2)\]
Store Typing

• Associates types with locations
  - For example,
    \[ l_1: \texttt{Int}, \ l_6: \texttt{Int*Int} \]
  - Denoted \( \Delta \)

• Typing Judgment must change
  \[ \Delta, \Gamma \vdash \texttt{e : t} \]

• Only one rule actually uses the store typing
  \[ \Delta, \Gamma \vdash l : \Delta(l) \quad \texttt{Ref} \]
Type Soundness

• A program \((M,e)\) is well-formed if there is a store typing \(\Delta\) such that
  - Every location \(l\) in \(M\) contains a value of type \(\Delta(l)\)
  - There is a proof \(\Delta,\emptyset \vdash e : t\)

• 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 \((M,\text{fail})\)] or else \(p \rightarrow p'\)
Assign-once variables

• Specification:

```ocaml
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 Oneshot
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