References and Exceptions

CS 565
Lecture 12
3/22/06
References

- In most languages, variables are mutable:
  - it serves as a name for a location
  - the contents of the location can be overwritten, and still be referred to by the same name

- In ML, variables only name values:
  - bindings are immutable
  - introduce a new class of values called references.
  - A variable bound to mutable location will have type $\text{Ref } \tau$
Basic Operations

- **Create a reference:**
  - \texttt{ref s}: returns a reference to a location that contains the value denoted by \texttt{s}.
  - If \texttt{s} has type \(\tau\), then \texttt{ref s : \tau} ref

- **Dereference:**
  - \texttt{!r}: returns the contents of the location referenced by \texttt{r}
  - If \texttt{r} has type \(\tau\) ref, then \texttt{!r : \tau}

- **Assignment:**
  - \texttt{r := s}: changes the contents of the location referenced by \texttt{r} to hold the value denoted by \texttt{s}.
  - If \texttt{r} has type \(\tau\) ref, and \texttt{s} has type \(\tau\), then \texttt{r := s} has value unit of type \texttt{Unit}.

- **No explicit deallocation operation.**
References and Stores

- Distinction between a reference and the location pointed to by that reference:
  - \( r = s \): binds a reference to the location pointed to by \( r \) to \( s \).
  - Thus,
    \[
    r = s \\
    s := 13
    \]
  - \( r \) and \( s \) are aliases for the same location
References and Shared State

- Implement implicit communication channels:
  
  \[ c = \text{ref} \ 0 \]
  
  \[ \text{incc} = \lambda \ x: \text{Unit.} \ (c := \text{succ} \ (!c); \ !c) \]
  
  \[ \text{decc} = \lambda \ x: \text{Unit.} \ (c := \text{pred}(!c); \ !c) \]

  \[ \text{incc unit} \rightarrow 1 \]
  
  \[ \text{decc unit} \rightarrow 0 \]

- Package both operations together:
  
  \[ o = \{ \ i = \text{incc}, \ d = \text{decc} \} \]

  We have now have a simple form of object:
  
  a collection of operations that share access to common state
References to Complex Types

A location can hold values of any type

Example:

\[
\text{newarray} = \lambda z : \text{unit. ref (} \lambda n : \text{Nat. 0)} \\
\text{newarray : Unit} \rightarrow (\text{Nat} \rightarrow \text{Nat}) \text{ ref} \\
\text{lookup} = \lambda a : \text{Ref (Nat} \rightarrow \text{Nat). } \lambda n : \text{Nat. } (!a) n; \\
\text{lookup: (Nat} \rightarrow \text{Nat) ref} \rightarrow \text{Nat} \rightarrow \text{Nat} \\
\text{update} = \lambda a : (\text{Nat} \rightarrow \text{Nat}) \text{ ref.} \\
\quad \lambda m : \text{Nat. } \lambda v : \text{Nat.} \\
\quad \quad \text{let oldf = !a} \\
\quad \quad \quad \text{in a := (} \lambda n : \text{Nat. if equal m n then } v \text{ else oldf n)} \\
\text{update: (Nat} \rightarrow \text{Nat) ref} \rightarrow \text{Nat} \rightarrow \text{Nat} \rightarrow \text{Unit}
\]
Typing Rules

\[ \Gamma \vdash e_1 : \tau_1 \]
\[ \Gamma \vdash \text{ref } e_1 : \tau_1 \text{ ref} \]

\[ \Gamma \vdash e_1 : \tau_1 \text{ ref} \]
\[ \Gamma \vdash !e_1 : \tau_1 \]

\[ \Gamma \vdash e_1 : \tau_1 \text{ ref} \quad \Gamma \vdash e_2 : \tau_2 \]
\[ \Gamma \vdash e_1 := e_2 : \text{Unit} \]
Evaluation

☐ How do we capture the operational (runtime) behavior of reference operations?
  ◦ What does it mean to “allocate” storage?
  ◦ What does it mean to “assign” to a location?

☐ Think of the store as an array of values
  ◦ rather than think of references as addresses (numbers), think of them as elements of a set $L$ of store locations
Evaluation Rules

Change evaluation rules to deal with stores:

\[ \lambda \mathbf{x} : \tau . \mathbf{e}, \mu \downarrow \lambda \mathbf{x} : \tau . \mathbf{e}, \mu \]

\[ \mathbf{e}_1, \mu \downarrow \lambda \mathbf{x} : \tau . \mathbf{e}_1', \mu' \]

\[ \mathbf{e}_2, \mu' \downarrow \mathbf{v}_2, \mu'' \]

\[ [\mathbf{v}_2/x] \mathbf{e}_1', \mu'' \downarrow \mathbf{v}, \mu''' \]

\[ \mathbf{e}_1 \mathbf{e}_2, \mu \downarrow \mathbf{v}, \mu''' \]
Evaluation (cont)

- Values: \( v ::= n | \text{true} | \text{false} | \lambda x: \tau. e | \text{unit} | ! \\
- Terms: \( e ::= x | \lambda x: \tau. e | e e | \text{unit} | \text{ref } e | !e | e := e | ! \\

Dereference:

\[
\begin{align*}
\text{e, } \mu & \downarrow \text{! } \mu' & \mu'(!) = v \\
\text{!e, } \mu & \downarrow v \mu'
\end{align*}
\]

Assignment:

\[
\begin{align*}
\text{e}_1, \mu & \downarrow \text{! } \mu' & \mu'' & \Downarrow & \text{v, } \mu'' \\
\text{e}_1 := & \text{e}_2, \mu & \downarrow \text{unit, } \mu'' & \Downarrow & \text{v} \Rightarrow \mu
\end{align*}
\]

Allocation:

\[
\begin{align*}
\text{e, } \mu & \downarrow \text{v, } \mu' & \mu'' = \mu' \Downarrow & \text{v} \Rightarrow \mu & \mu' \not\in \text{Dom}(\mu') \\
\text{ref e, } \mu & \downarrow \text{l, } \mu''
\end{align*}
\]
Locations

Extend typing rules to accommodate locations:

\[
\frac{\Gamma, \mu \vdash \mu(l) : \tau}{\Gamma, \mu \vdash l : \tau \text{ ref}}
\]

The type of a location depends upon the contents of the store:

- If \( \mu = [l_1 \mapsto \text{Unit}, l_2 \mapsto \text{Unit}] \), then \( l_2 \) has type \( \text{unit} \)
- If \( \mu = [l_1 \mapsto \text{unit}, l_2 \mapsto \text{Unit} \to \text{Unit}] \), then \( l_2 \) has type \( \text{Unit} \to \text{Unit} \)
This type rule isn't very satisfactory
1. large type derivations
2. doesn't handle cycles in the store

Suppose the store is defined by:

\[
[ l_1 \mapsto \lambda x: \text{Nat.} 99, \\
l_2 \mapsto \lambda x: \text{Nat.} (l_1l_2) x, \\
l_3 \mapsto \lambda x: \text{Nat.} (l_2l_3) x, \\
\ldots ]
\]

Now, typing \( l_n \) requires calculating types of \( l_1, \ldots, l_{n-1} \)

Suppose we have:

\[
[ l_1 \mapsto \lambda x: \text{Nat.} (l_1l_2) x, \\
l_2 \mapsto \lambda x: \text{Nat.} (l_2l_1) x]
\]
Issues

- How do we create cycles?
  
  ```
  let r1 = ref (λ x:Nat.0)
  r2 = ref (λ x:Nat.(!r1) x)
  in (r1 := λ x:Nat.(!r2) x;
  r2)
  ```

- Unnecessary for us to recalculate the type of a location everytime it is mentioned:
  - we know its type at the point it is declared
  - all other values stored in the location must share that type
Store Typings

For a store that contains:

\[
\begin{align*}
\ll_1 & \mapsto \lambda x : \text{Nat}. 99, \\
\ll_2 & \mapsto \lambda x : \text{Nat}. (\ll_1) \times, \\
\ll_3 & \mapsto \lambda x : \text{Nat}. (\ll_2) \times, \\
& \quad \ldots \]
\end{align*}
\]

A reasonable typing would be:

\[
\begin{align*}
\ll_1 & \mapsto \text{Nat} \rightarrow \text{Nat}, \\
\ll_2 & \mapsto \text{Nat} \rightarrow \text{Nat}, \\
\ll_3 & \mapsto \text{Nat} \rightarrow \text{Nat}, \\
& \quad \ldots \\
\end{align*}
\]
Store Typings

A store typing $\Sigma$ describes the store $\mu$ in which we intend to evaluate term $e$. We use $\Sigma$ to lookup the types of locations referenced in $e$.

$$\Sigma(l) : \tau$$

$\Gamma, \Sigma \vdash l : \tau \, \text{ref}$

Need to propagate $\Sigma$ to all the other type rules defined earlier.
Store Typings

A given store may have multiple store typings:

Suppose \( \mu = [ l \mapsto \lambda x: \text{Unit}. (!!) x ] \)

Then,

\[ \Sigma_1 = l \mapsto \text{Unit} \rightarrow \text{Unit} \]

\[ \Sigma_2 = l \mapsto \text{Unit} \rightarrow \text{Unit} \rightarrow \text{Unit} \]
Exceptions

- An exception is a construct that allows programmers deal with exceptional conditions (e.g., errors)
  - exception handler: code that is associated with an exception that is invoked when an exception is raised.
  - raising an exception causes the computation to transfer control to the closest enclosing handler (in the dynamic context).
First step: Errors

An error is a special term that when evaluated stops evaluation of the term.

- Values: \( v ::= n \mid \text{true} \mid \text{false} \mid \lambda \, x : \tau. \, e \mid \)
- Terms: \( e ::= x \mid \lambda \, x : \tau. \, e \mid e \mid e \mid \text{error} \)

Evaluation rules (Contextual)
\[ E ::= [] \mid E \mid (\lambda \, x : \tau. \, e') \, E \]

- error \( e \rightarrow \text{error} \)
- \( v \, \text{error} \rightarrow \text{error} \)

\[ \Gamma \vdash \text{error} : \tau \quad \text{(an error can be of any type)} \]

What difficulties do we face in expressing error using a big-step semantics?
Typing

- Since error can be of any type, it breaks uniqueness property of types:
  - subtyping: allow error to be “promoted” to any other type as necessary by defining it the “minimal” type
  - polymorphism: give error the polymorphic type $\forall X.X$ that can be “instantiated” to any other type as necessary

- Why not just use annotations? Consider:
  $$(\lambda x:\text{Nat. } x) \ ((\lambda y:\text{Bool. } 13) \ (\text{error as Bool}))$$
Exceptions

- Evaluating error “unwinds” the call-stack until all frames have been discarded, and evaluation returns to the top-level.

- Generalizing to exceptions, allows handlers to be inserted between activation frames in the call-stack
  - control reverts to the handler that handles the exception raised
  - use the first matching handler
Error Handling

- Values: \( v ::= n \mid \text{true} \mid \text{false} \mid \lambda x: \tau. \ e \mid \)
- Terms: \( e ::= x \mid \lambda x: \tau.e \mid e \ e \mid \text{error} \mid \)
  \( \text{try } e \text{ with } e \)

Contexts and Reduction Rules:
- \( E ::= \ldots \mid \text{try } E \text{ with } e \)
- \( e ::= \ldots \mid \text{try error with } e \mid \text{try } v \text{ with } e \)

- \( \text{try } v \text{ with } e \rightarrow v \)
- \( \text{try error with } e \rightarrow e \)

Type rule:
\[
\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \text{try } e_1 \text{ with } e_2 : \tau}
\]
Exception-Carrying Values

- Suppose we want to send information to a handler about the unusual event that triggered the exception
- Allow exceptions to carry values
- When an exception is raised, supply a value that is an argument to the handler.
Evaluation Rules

Values: \( v ::= n \mid true \mid false \mid \lambda x: \tau. e \mid \)

Terms: \( e ::= x \mid \lambda x: \tau.e \mid e \ e \mid try \ e \ with \ e \mid \)

\( raise \ e \)

Evaluation contexts and Reduction Rules:

\( E ::= \ldots \mid try \ E \ with \ e \mid raise \ E \)

\( try \ v \ with \ e \rightarrow v \)

\( (raise \ v) \ e \rightarrow raise \ v \)

\( raise \ (raise \ v) \rightarrow raise \ v \)

\( try \ (raise \ v) \ with \ e \rightarrow e \ v \)

\( v \ (raise \ v_1) \rightarrow raise \ v_1 \)
Typing Rules

\[
\begin{align*}
\Gamma \vdash e : \tau_{\text{exn}} & \quad \Gamma \vdash e : \tau & \quad \Gamma \vdash e_h : \tau_{\text{exn}} \rightarrow \tau \\
\Gamma \vdash \text{raise } e : \tau & \quad \Gamma \vdash \text{try } e \text{ with } e_h : \tau
\end{align*}
\]
Exception Types

What type should $\tau_{\text{exn}}$ be?

- Take it to be Nat. Corresponds to errno convention in Unix.
- Take it to be String.
- Take it to be a variant type:

$$\tau_{\text{exn}} = \text{divideByZero} : \text{Unit} + \text{overflow} : \text{Unit} + \text{fileNotFound} : \text{String}$$

  Not particularly flexible

- Assume $\tau_{\text{exn}}$ is an extensible variant:
  - In ML, there is a single extensible variant type called exn.
  - exception E of T means “make sure E is different from any other tag present in the variant type T”
Continuations

- Exceptions and errors are instantiations of a more general control feature that allows non-local transfer of control from point in the program to another.
  - structured jumps or gotos
- Can we generalize (or reify) this notion into our core language?
  - result is a continuation: a reified representation (in the form of an abstraction) of a program’s control-stack.
Continuations

- Define a new primitive call/cc:
  - Takes as its argument a procedure
    \[(\text{lambda} \,(k) \,e)\]
    and binds to \(k\) a reified representation of
    the call-stack at the point of evaluation.
  - Can transfer control to this point via
    application.
Examples

\texttt{call/cc (λ k. (k 3) + 2) + 1 \rightarrow 4}

\texttt{ref (λ v. 0)}
\texttt{call/cc (λ k. (r := k; (k 3) + 2)) + 1 \rightarrow 4}
\texttt{(r 4) \rightarrow 5}

\texttt{let f = call/cc (λ k. λ x. k (λ y. x + y)}
\texttt{in f 6 \rightarrow}
\texttt{ 12}
Evaluation and Typing Rules

- First, consider the evaluation rule in an untyped setting:

\[ E[\text{call/cc } e] = E[e (\lambda v. \text{abort } (E[v]))] \]

where abort represents the “initial” continuation.

- Typing is a bit harder because continuations bound by call/cc can be invoked in several different contexts.
An Example in ML

1 + call/cc (fn k => hd (if b
    then [(k 3) + 1]
    else 5 :: (k 4)))

- k is invoked in two contexts:
  - one expects an integer
  - other expects a list
- Since continuations never return, how do we choose the result type?
- One possible type: \((\tau \text{ cont} \rightarrow \tau) \rightarrow \tau\)
  - Will revisit this issue when we consider polymorphism.