References and Mutable Data Structures

Principles of Programming Languages

CSE 526

1 Syntax

2 Semantics
References

Syntax:

\[ t ::= \ldots \]
\[ \mid \text{ref } t \quad \text{Creation of boxed value} \]
\[ \mid ! t \quad \text{Dereference of a boxed value} \]
\[ \mid t := t \quad \text{Assignment} \]
\[ \mid t ; t \quad \text{Sequence} \]

Values:

\[ v ::= \ldots \]
\[ \mid \ell \quad \text{location values} \]

Types:

\[ T ::= \ldots \]
\[ \mid \text{Ref } T \quad \text{Reference type} \]
Examples use of references

ref 0

Create a new location, initialized to 0

let r = ref λ x. x
    in let p = ref (! r) 0
        in (! r) (! p)

Get a location r initialized to the identity function; get its value, apply it to 0 and store it in a new location (p); finally apply the value at p to the value at r.

let f = λ x. (x := succ ! x)
    in let r = ref 0
        in (f r); !r

Define function f that increments the value at a given location; get a location r initialized to 0; finally apply f to r and then return the updated value at r.
We will use an abstract structure, called store, that associates locations with values.
We will use an abstract structure, called **store**, that associates **locations** with **values**.

Each expression that creates or modifies locations will change the store.
We will use an abstract structure, called \textit{store}, that associates \textit{locations} with \textit{values}.

Each expression that creates or modifies locations will change the store.

Each expression that uses values at some locations will access the store.
We will use an abstract structure, called **store**, that associates \textit{locations} with \textit{values}.

Each expression that creates or modifies locations will change the store.

Each expression that uses values at some locations will access the store.

**Major change:** Since any expression may now contain \texttt{"ref t"} or \texttt{"! t"}, the small step semantics...
Store

- We will use an abstract structure, called **store**, that associates *locations* with *values*.
- Each expression that creates or modifies locations will change the store.
- Each expression that uses values at some locations will access the store.

**Major change:** Since any expression may now contain "ref \( t \)" or "! \( t \)" , the small step semantics
  - that was of the form \( t \rightarrow t' \)
Store

- We will use an abstract structure, called store, that associates locations with values.
- Each expression that creates or modifies locations will change the store.
- Each expression that uses values at some locations will access the store.

**Major change:** Since any expression may now contain “ref t” or “! t”, the small step semantics
  - that was of the form \( t \rightarrow t' \)
  - will now be of the form \( \mu|t \rightarrow \mu'|t' \)
We will use an abstract structure, called **store**, that associates *locations* with *values*.

Each expression that creates or modifies locations will change the store.

Each expression that uses values at some locations will access the store.

**Major change:** Since any expression may now contain “`ref t`” or “`! t`”, the small step semantics

- that was of the form $t \rightarrow t'$
- will now be of the form $\mu|t \rightarrow \mu'|t'$
- where $\mu$ and $\mu'$ represent stores.
Small-step semantics with Eager Evaluation

- **R–1:**
  \[
  \frac{\mu | t_1 \rightarrow \mu' | t'_1}{\mu | \text{ref } t_1 \rightarrow \mu' | \text{ref } t'_1}
  \]

- **R–2:**
  \[
  \frac{\ell \notin \text{dom}(\mu), \; \mu' = \mu[\ell \mapsto v_1]}{\mu | \text{ref}(v_1) \rightarrow \mu' | \ell}
  \]

- **D–1:**
  \[
  \frac{\mu | t_1 \rightarrow \mu' | t'_1}{\mu | ! t_1 \rightarrow \mu' | ! t'_1}
  \]

- **D–2:**
  \[
  \frac{\mu | ! \ell \rightarrow \mu | \mu(\ell)}{}
  \]

- **When locations are created, the initial value is first evaluated (R–1).**
- **A store with domain \( \mu[l \mapsto v] \) is a store that is same as \( \mu \), except that it has value \( v \) at location \( l \).**
- **In a dereference expression of the form \( ! t \), the inner expression \( t \) is completely evaluated to a location before the dereference operation is done.**
Small-step semantics with Eager Evaluation

**Assign–1:**

\[
\begin{align*}
\mu \mid t_1 & \rightarrow \mu' \mid t_1' \\
\mu \mid t_1 : &= t_2 \rightarrow \mu' \mid t_1' := t_2
\end{align*}
\]

**Assign–2:**

\[
\begin{align*}
\mu \mid t_2 & \rightarrow \mu' \mid t_2' \\
\mu \mid v_1 : &= t_2 \rightarrow \mu' \mid v_1 := t_2'
\end{align*}
\]

**Assign–3:**

\[
\mu' = \mu[\ell \mapsto v] \\
\mu \mid \ell : = v \rightarrow \mu' \mid \text{unit}
\]

- In an assignment of the form \( t_1 := t_2 \), \( t_1 \) is first completely evaluated, then \( t_2 \) is evaluated, and the assignment is finally made.

- The “value” of an assignment expression is \text{unit}, independent of the r.h.s. value that is assigned to l.h.s. This behavior is similar to SML/OCAML, and differs from the “cascading” assignment semantics used in C/C++/Java family of languages.
Small-step semantics with Eager Evaluation

\[
\text{SEQ-1:} \quad \frac{\mu | t_1 \rightarrow \mu' | t'_1}{\mu | t_1 ; t_2 \rightarrow \mu' | t'_1 ; t_2}
\]

\[
\text{SEQ-2:} \quad \frac{\mu | \text{unit} ; t_2 \rightarrow \mu | t_2}{\}
\]

- In a sequence expression of the form \( t_1 ; t_2 \), \( t_1 \) is first completely evaluated.
- Once \( t_1 \) is evaluated to a value, that value is discarded, and the sequence simply results in \( t_2 \).
- As in OCAML, \( t_1 \) should evaluate to unit, otherwise the evaluation will be stuck.
Types

**T–Ref:**

\[
\frac{\Gamma \vdash t_1 : T_1}{\Gamma \vdash \text{ref } t_1 : \text{Ref } T_1}
\]

**T–Deref:**

\[
\frac{\Gamma \vdash t_1 : \text{Ref } T_1}{\Gamma \vdash ! t_1 : T_1}
\]

**T–Assign:**

\[
\frac{\Gamma \vdash t_1 : \text{Ref } T_1 \quad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1 := t_2 : \text{Unit}}
\]

**T–Seq:**

\[
\frac{\Gamma \vdash t_1 : \text{Unit} \quad \Gamma \vdash t_2 : T_2}{\Gamma \vdash t_1 ; t_2 : T_2}
\]
Impact of Mutability

- When we introduce mutable structures, it can have significant effect on the rest of the language as well.
- Every expression now has the potential to change the store.
- This changes all semantic rules:

<table>
<thead>
<tr>
<th>Old</th>
<th>New</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t_1 \to t'_1 )</td>
<td>( \mu</td>
</tr>
<tr>
<td>( \text{if}(t_1, t_2, t_3) \to \text{if}(t'_1, t_2, t_3) )</td>
<td>( \mu</td>
</tr>
<tr>
<td>( \text{if(true, } t_2, t_3) \to t_2 )</td>
<td>( \mu</td>
</tr>
<tr>
<td>( \text{if(false, } t_2, t_3) \to t_3 )</td>
<td>( \mu</td>
</tr>
</tbody>
</table>