Administrivia

◊ [Use of homework solutions]
◊ [Study groups?]

References, continued

Final example

NatArray = Ref (Nat → Nat);

newarray = λa:Unit. ref (λn:Nat. 0);
    : Unit → NatArray

lookup = λa:NatArray. λn:Nat. (!a) n;
    : NatArray → Nat → Nat

       let oldf = !a in
       a := (λn:Nat. if equal m n then v else oldf n);
    : NatArray → Nat → Nat → Unit
### Syntax

\[ t ::= \]

- unit
- \( x \)
- \( \Lambda x : T . t \)
- \( t t \)
- ref \( t \)
- \( !t \)
- \( t : = t \)

... plus other familiar types, in examples.

### Typing Rules

\[ \Gamma \vdash t_1 : T_1 \]  
\[ \Gamma \vdash \text{ref } t_1 : \text{Ref } T_1 \]  
\[ (\text{T-REF}) \]

\[ \Gamma \vdash t_1 : \text{Ref } T_1 \]
\[ \Gamma \vdash !t_1 : T_1 \]  
\[ (\text{T-DEREF}) \]

\[ \Gamma \vdash t_1 : \text{Ref } T_1 \]
\[ \Gamma \vdash t_2 : T_1 \]
\[ \Gamma \vdash t_1 := t_2 : \text{Unit} \]  
\[ (\text{T-ASSIGN}) \]

### Evaluation

What is the **value** of the expression `ref 0`?

Crucial observation: evaluating `ref 0` must do something.

Otherwise,

\[ r = \text{ref } 0 \]
\[ s = \text{ref } 0 \]

and

\[ r = \text{ref } 0 \]
\[ s = r \]

would behave the same.
**Evaluation**

What is the value of the expression `ref 0`?

Crucial observation: evaluating `ref 0` must do something.

Otherwise,

\[
\begin{align*}
  r &= \text{ref } 0 \\
  s &= \text{ref } 0
\end{align*}
\]

and

\[
\begin{align*}
  r &= \text{ref } 0 \\
  s &= r
\end{align*}
\]

would behave the same.

Specifically, evaluating `ref 0` should allocate some storage and return a reference (or pointer) to that storage.

---

**The Store**

A reference is a pointer into the memory (the heap or store).

What is the store?

- Concretely: An array of 8-bit bytes, indexed by 32-bit integers.
The Store

A reference is a pointer into the memory (the heap or store).

What is the store?

◊ Concretely: An array of 8-bit bytes, indexed by 32-bit integers.
◊ More abstractly: an array of values

Locations

Syntax of values:

\[
\begin{align*}
\text{v} &::= \text{values} \\
& \quad \text{unit} \\
& \quad \Lambda x:T.t \\
& \quad l \\
\end{align*}
\]

... and since all values are terms...

Syntax of Terms

\[
\begin{align*}
\text{t} &::= \text{terms} \\
& \quad \text{unit} \\
& \quad x \\
& \quad \Lambda x:T.t \\
& \quad t \triangleright t \\
& \quad \text{ref } t \\
& \quad t := t \\
& \quad l \\
\end{align*}
\]

terms
unit constant
variable
abstraction
application
reference creation
dereference
assignment
store location
Aside

Does this mean we are going to allow programmers to write explicit locations in their programs?
No: This is just a modeling trick. We are enriching the language of terms to include some run-time structures, so that we can continue to formalize the evaluation relation as a relation between terms.

Evaluation

The result of evaluating a term now depends on the store in which it is evaluated. Moreover, the result of evaluating a term is not just a value — we must also keep track of the changes that get made to the store.
I.e., the evaluation relation should now map a term and a store to a reduced term and a new store.

\[ t \mid \mu \rightarrow t' \mid \mu' \]

We use the metavariable \( \mu \) to range over stores.

Evaluation

Evaluation rules for function abstraction and application are augmented with stores, but don't do anything with them.

\[
\frac{\tau_1 \mid \mu \rightarrow \tau'_1 \mid \mu'}{t_1 \mid t_2 \mid \mu \rightarrow \tau'_1 \mid t_2 \mid \mu'}
\]

(E-APP1)

\[
\frac{\tau_2 \mid \mu \rightarrow \tau'_2 \mid \mu'}{v_1 \mid t_2 \mid \mu \rightarrow v_1 \mid \tau'_2 \mid \mu'}
\]

(E-APP2)

\[
(\lambda x : T_{11} \cdot t_{12}) \mid v_{2} \mid \mu \rightarrow [x \mapsto v_{2}] t_{12} \mid \mu
\]

(E-APPABS)

A term \( !t_1 \) first evaluates in \( t_1 \) until it becomes a value...

\[
\frac{\tau_1 \mid \mu \rightarrow \tau'_1 \mid \mu'}{t_1 \mid \mu \rightarrow !t'_1 \mid \mu'}
\]

(E-DEREF)

... and then looks up this value (which must be a location, if the original term was well typed) and returns its contents in the current store:

\[
\frac{\mu(1) = v}{!1 \mid \mu \rightarrow v \mid \mu}
\]

(E-DEREFLOC)
Typing Locations

Q: What is the type of a location?

A: It depends on the store! E.g., in the store \( (l_1 \mapsto \text{unit}, l_2 \mapsto \text{unit}) \), the term \(!l_2\) has type \(\text{Unit}\). But in the store \( (l_1 \mapsto \text{unit}, l_2 \mapsto \lambda x: \text{Unit}.x) \), the term \(!l_2\) has type \(\text{Unit} \to \text{Unit}\).
Typing Locations — first try

Roughly:

\[
\Gamma \vdash \mu(I) : T_1 \\
\Gamma \vdash I : \text{Ref } T_1
\]

More precisely:

\[
\Gamma \mid \mu \vdash \mu(I) : T_1 \\
\Gamma \mid \mu \vdash I : \text{Ref } T_1
\]

I.e., typing is now a four-place relation (between contexts, stores, terms, and types).

Problem

However, this rule is not completely satisfactory. For one thing, it can make typing derivations very large!

E.g., if

\[
(\mu - I_1 \mapsto \lambda x : \text{Nat.} \ 999,
I_2 \mapsto \lambda x : \text{Nat.} \ \! I_1 \ (\! I_1 \ x),
I_3 \mapsto \lambda x : \text{Nat.} \ \! I_2 \ (\! I_2 \ x),
I_4 \mapsto \lambda x : \text{Nat.} \ \! I_3 \ (\! I_3 \ x),
I_5 \mapsto \lambda x : \text{Nat.} \ \! I_4 \ (\! I_4 \ x)),
\]

then how big is the typing derivation for \( \! I_5 \)?
Store Typings

Observation: a given location in the store is always used to hold values of the same type.

These intended types can be collected into a store typing --- a partial function from locations to types.

\[ \begin{align*}
\text{E.g., for} & \\
\mu &= (l_1 \mapsto \lambda x : \text{Nat}. \, 999, \\
& \quad l_2 \mapsto \lambda x : \text{Nat}. \, l_1 \, (l_1 \, x), \\
& \quad l_3 \mapsto \lambda x : \text{Nat}. \, l_2 \, (l_2 \, x), \\
& \quad l_4 \mapsto \lambda x : \text{Nat}. \, l_3 \, (l_3 \, x), \\
& \quad l_5 \mapsto \lambda x : \text{Nat}. \, l_4 \, (l_4 \, x),
\end{align*} \]

A reasonable store typing would be

\[ \Sigma = (l_1 \mapsto \text{Nat} \rightarrow \text{Nat}, \\
& \quad l_2 \mapsto \text{Nat} \rightarrow \text{Nat}, \\
& \quad l_3 \mapsto \text{Nat} \rightarrow \text{Nat}, \\
& \quad l_4 \mapsto \text{Nat} \rightarrow \text{Nat}, \\
& \quad l_5 \mapsto \text{Nat} \rightarrow \text{Nat}) \]

Now, suppose we are given a store typing \( \Sigma \) describing the store \( \mu \) in which we intend to evaluate some term \( t \). Then we can use \( \Sigma \) to look up the types of locations in \( t \) instead of calculating them from the values in \( \mu \).

\[ \frac{\Sigma (l) = T_1}{\Gamma, l \vdash t : \text{Ref} \, T_1} \]

\[ (T-\text{Loc}) \]

i.e., typing is now a four-place relation between contexts, store typings, terms, and types.

Final typing rules

\[ \frac{\Sigma (l) = T_1}{\Gamma, l \vdash t : \text{Ref} \, T_1} \quad (T-\text{Loc}) \]

\[ \frac{\Gamma \vdash t_1 : T_1}{\Gamma \vdash \text{ref} \, t_1 : \text{Ref} \, T_1} \quad (T-\text{REF}) \]

\[ \frac{\Gamma \vdash t_1 : \text{Ref} \, T_1}{\Gamma \vdash t_1 : T_1} \quad (T-\text{DEREF}) \]

\[ \frac{\Gamma \vdash t_1 : \text{Ref} \, T_1}{\Gamma \vdash t_1 \, : = t_2 : \text{Unit}} \quad (T-\text{ASSIGN}) \]
Aside: garbage collection

[...]

Aside: pointer arithmetic

[...]

Exceptions