COMPSCI 3MI3 - Principles of Programming Languages

Types for references and memory

J. Carette

McMaster University

Fall 2023

Adapted from “Types and Programming Languages” by Benjamin C. Pierce

About :=

The left-hand-side of := is a **location** of a memory cell.

About :=

The left-hand-side of := is a **location** of a memory cell.
When these can be stored to multiple times (without allocation), called **mutable references**.

About :=

The left-hand-side of := is a **location** of a memory cell.
When these can be stored to multiple times (without allocation), called **mutable references**.
In general, three operations:

- Memory allocation, aka creating a **reference**.
- A "store" operation, aka **assignment**.
- A "retrieve" operation, aka **dereferencing**.

Depending on the programming language, some or all of these operations may be implicit in the grammar.

- Python hides allocation and retrieval, but storage is explicit.
- C/C++ hides retrieval, with allocation and storage being explicit.
- In ML, all three operations are explicit.
- Haskell buries these very deeply in a library.

References

$$\langle t \rangle ::= \ldots$$
$$\mid \quad \text{ref t}$$
$$\mid \quad \text{!t}$$
$$\mid \quad \text{t := t}$$
$$\mid \quad l$$

$$\langle v \rangle ::= \lambda\, \text{x}:\langle T \rangle.\langle t \rangle$$
$$\mid \quad \text{unit}$$
$$\mid \quad l$$

$$\langle T \rangle ::= \ldots$$
$$\mid \quad \text{Ref } \langle T \rangle$$

Types (to be refined):

$$\frac{\Gamma \vdash t_1 : T_1}{\Gamma \vdash \texttt{ref}\ \ t_1 : \mathit{Ref}\ \ T_1} \qquad \text{(T-Ref)}$$

$$\frac{\Gamma \vdash t_1 : \mathit{Ref}\ \ T_1}{\Gamma \vdash !t_1 : T_1} \qquad \text{(T-Deref)}$$

$$\frac{\Gamma \vdash t_1 : \mathit{Ref}\ \ T_1 \qquad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1 := t_2 : \mathit{Unit}} \qquad \text{(T-Assign)}$$

A Sample Program

```
let x = ref 0 in
  let y = ref 0 in
  let z = ref 1 in
    x := 2;
    y := 3;
    z := !x + !y;
    !z

>> 5
```

`ref` is like `new` in Java.

Aliasing

```
let x = ref 5 in
  let y = x in
    x := 10;
    !y

>> 10
```

Sharing is not necessarily bad

Aliases cells as *implicit communication channels*:

```
let c = ref 0 in
let inc_c = λx : Unit. (c := succ (!c); !c) in
let dec_c = λx : Unit. (c := pred (!c); !c) in
  inc_c unit;
  inc_c unit;
  dec_c unit
```

The values of c are 1 then 2 then 1 again.

Shades of OO...

Heap / Store

Heap   array of (typed) values, 'memory store' with 'locations'. Let $\mathcal{L}$ denote some set of **store locations**. Use l to range over $\mathcal{L}$.

Heap / Store

Heap array of (typed) values, 'memory store' with 'locations'. Let $\mathcal{L}$ denote some set of **store locations**. Use l to range over $\mathcal{L}$.

A *memory store* is then a (partial) function from $\mathcal{L}$ to values.

Vocabulary:

- We will use μ to denote memory stores.
- References will be called **locations**.
- "memory store" wil be just **store**.

Store passing style

Attach μ directly to terms:

$$t \mid \mu$$

Evaluation might affect the store; change evaluation relation:

$$t \mid \mu \rightarrow t' \mid \mu'$$

New evaluation rules:

$$(\lambda x : T_{11}.t_{12})v_2 \mid \mu \rightarrow [x \mapsto v_2]t_{12} \mid \mu \qquad \text{(E-AppAbs)}$$

$$\frac{t_1 \mid \mu \rightarrow t_1' \mid \mu'}{t_1\ t_2 \mid \mu \rightarrow t_1'\ t_2 \mid \mu'} \qquad \text{(E-App1)}$$

$$\frac{t_2 \mid \mu \rightarrow t_2' \mid \mu'}{t_1\ t_2 \mid \mu \rightarrow t_1\ t_2' \mid \mu'} \qquad \text{(E-App2)}$$

Dereferencing

New evaluation rules for *dereferencing*.

$$\frac{\mu(l) = v}{!l \mid \mu \to v \mid \mu} \quad \text{(E-DerefLoc)}$$

$$\frac{\mu(l) = v}{!l \mid \mu \to v \mid \mu} \quad \text{(E-DerefLoc)}$$

$$\frac{t_1 \mid \mu \to t_1' \mid \mu'}{!t_1 \mid \mu \to !t_1' \mid \mu'} \quad \text{(E-Deref)}$$

- Dereferencing a location: if we have a value for that location, return it.
- Otherwise evaluate t_1 (possibly with an effect)

Note: ! 5 is *stuck*.

Assignment

$$l := v_2 \mid \mu \to unit \mid [l \mapsto v_2]\mu \qquad \text{(E-Assign)}$$

$$\frac{t_1 \mid \mu \to t_1' \mid \mu'}{t_1 := t_2 \mid \mu \to t_1' := t_2 \mid \mu'} \qquad \text{(E-Assign1)}$$

$$\frac{t_2 \mid \mu \to t_2' \mid \mu'}{v_1 := t_2 \mid \mu \to v_1 := t_2' \mid \mu'} \qquad \text{(E-Assign2)}$$

- $[l \mapsto v_2]\mu$ means "a store which maps l to v, with all other locations mapping to the same things as in μ."

Allocation

$$\frac{l \notin dom(\mu)}{ref\ v_1 \mid \mu \to l \mid \mu \oplus l \mapsto v_1} \quad \text{(E-RefV)}$$

$$\frac{t_1 \mid \mu \to t_1' \mid \mu'}{ref\ t_1 \mid \mu \to ref\ t_1' \mid \mu'} \quad \text{(E-Ref)}$$

- E-RefV: we select a *fresh location* l not already used in μ.
- Extend μ with the new mapping
- The term `ref v` evaluates to this fresh location l.

Typing the store

Skip entirely: a first “simple” approach that does not scale.

Typing the store

Skip entirely: a first "simple" approach that does not scale.

Recall: type of a location is derivable at allocation from the type of the instantiating value.

Typing the store

Skip entirely: a first "simple" approach that does not scale.

Recall: type of a location is derivable at allocation from the type of the instantiating value.

Create a **typing store** Σ in parallel with contexts Γ.

$$\frac{\Sigma(l) = T_1}{\Gamma \mid \Sigma \vdash l : \mathit{Ref}\ T_1} \qquad \text{(T-Loc)}$$

- Γ starts off empty, and has typings added as the program is traversed.
- Σ will be the same
- Write empty Γ and emptr Σ as $\emptyset$.

Typing Allocation

$$\frac{\Sigma(l) = T_1}{\Gamma \mid \Sigma \vdash l : \mathit{Ref}\ T_1} \qquad \text{(T-Loc)}$$

$$\frac{\Gamma \mid \Sigma \vdash t_1 : T_1}{\Gamma \mid \Sigma \vdash \mathit{ref}\ t_1 : \mathit{Ref}\ T_1} \qquad \text{(T-Ref)}$$

$$\frac{\Gamma \mid \Sigma \vdash t_1 : \mathit{Ref}\ T_{11}}{\Gamma \mid \Sigma \vdash !t_1 : T_{11}} \qquad \text{(T-Deref)}$$

$$\frac{\Gamma \mid \Sigma \vdash t_1 : \mathit{Ref}\ T_{11} \qquad \Gamma \mid \Sigma \vdash t_2 : T_{11}}{\Gamma \mid \Sigma \vdash t_1 := t_2 : \mathit{Unit}} \qquad \text{(T-Assign)}$$

Typed Stores

Definition

A store μ is said to be **well typed** *with respect to a typing context Γ and a store typing Σ if:*

- $dom(\mu) = dom(\Sigma)$
- $\forall l \in dom(\mu) \mid \mu(l) : \Sigma(l)$

We write this $\Gamma \mid \Sigma \vdash \mu$

Preservation?

$$
\begin{aligned}
& (\Gamma \mid \Sigma \vdash t : T) \\
\wedge\ & (t \mid \mu \rightarrow t' \mid \mu') \\
\wedge\ & (\Gamma \mid \Sigma \vdash \mu) \\
\Longrightarrow\ & (\Gamma \mid \Sigma \vdash t' : T)
\end{aligned}
$$

But stepping can change μ and thus also Σ.

THEOREM: [Preservation]

$$
\begin{aligned}
&(\Gamma \mid \Sigma \vdash t : T) \\
&\wedge (t \mid \mu \rightarrow t' \mid \mu') \\
&\wedge (\Gamma \mid \Sigma \vdash \mu) \\
\Longrightarrow\ &(\exists \Sigma' \supseteq \Sigma \mid \\
&\qquad\quad (\Gamma \mid \Sigma' \vdash t' : T) \\
&\qquad \wedge\ (\Gamma \mid \Sigma' \vdash \mu') \\
&)
\end{aligned}
$$

Technical lemmas

LEMMA: [Preservation Over Substitution]

$$(\Gamma, x : S \mid \Sigma \vdash t : T) \wedge (\Gamma \mid \Sigma \vdash s : S) \implies (\Gamma \mid \Sigma \vdash [x \mapsto s]t : T]) \tag{1}$$

LEMMA: [Preservation Over Storage]

$$(\Gamma \mid \Sigma \vdash \mu) \wedge (\Sigma(l) = T) \wedge (\Gamma \mid \Sigma \vdash v : T) \implies (\Gamma \mid \Sigma \vdash [l \mapsto v]\mu)) \tag{2}$$

LEMMA: [Weakening Over Typing Stores]

$$(\Gamma \mid \Sigma \vdash t : T) \wedge (\Sigma' \supseteq \Sigma) \implies (\Gamma \mid \Sigma' \vdash t : T) \tag{3}$$

Progress

THEOREM: [Progress]
Suppose $\emptyset \mid \Sigma \vdash t : T$ for some T and Σ. Then either t is a value, or else, for any store μ such that $\emptyset \mid \Sigma \vdash \mu$, there is some term t' and store μ' such that $t \mid \mu \rightarrow t' \mid \mu'$.

Proof Sketch

- Induction on typing derivations.
- The canonical forms lemma needs two additional cases, stating that all values of type *Ref T* are locations, and similarly for *Unit*.