Chapter 13: Reference

Why reference

Typing

Evaluation

Store Typings

Safety

Notes

References

Mutability

So far, what we discussed does not include computational effects (also known as *side effects*) . In particular, whenever we defined function, we never changed variables or data. Rather, we always computed new data.

- For instance, the operations to insert an item into the data structure didn't effect the old copy of the data structure. Instead, we always built a new data structure with the item appropriately inserted.
- For the most part, programming in a functional style (i.e., without side effects) is a "good thing" because it's easier to reason locally about the behavior of the program.

Mutability

- In most programming languages, variables are mutable — i.e., a variable provides both
  - a name that refers to a previously calculated value, and
  - the possibility of overwriting this value with another (which will be referred to by the same name)
- In some languages (e.g., OCaml), these features are separate:
  - variables are only for naming — the binding between a variable and its value is immutable
  - introduce a new class of mutable values (called reference cells or references)
  - at any given moment, a reference holds a value (and can be dereferenced to obtain this value)
  - a new value may be assigned to a reference

Mutability

Writing values into memory locations is the fundamental mechanism of imperative languages such as Pascal or C.

Mutable structures are required to implement many *efficient* algorithms. They are also very convenient to represent the current state of a state machine.

Basic Examples

r = ref 5

!r

r := 7

(r:=succ(!r); !r)

(r:=succ(!r); r:=succ(!r); r:=succ(!r); r:=succ(!r); !r)

i.e.,

((((r:=succ(!r); r:=succ(!r)); r:=succ(!r)); :=succ(!r)); !r)

Reference

Basic operations:

- allocation : ref (operator)
- dereferencing : !
- assignment: :=

Is there any difference between?:

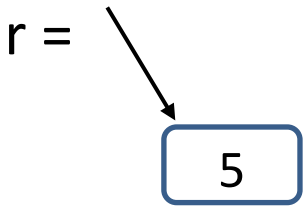
5 + 8;

r: =7;

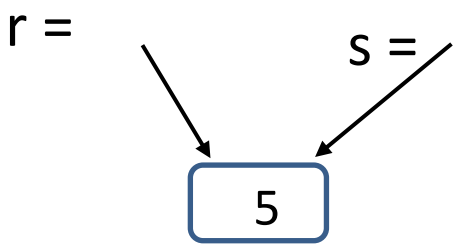
(r:=succ(!r); !r)

Aliasing

A value of type $\mathrm{ref}\ \mathrm{T}$ is a *pointer* to a cell holding a value of type T.



If this value is "copied" by assigning it to another variable, the cell pointed to is not copied. (r and s are *aliases*)



So we can change r by assigning to s:

(s:=10; !r)

Aliasing all around us

Reference cells are not the only language feature that introduces the possibility of aliasing.

- arrays
- communication channels
- I/O devices (disks, etc.)

The difficulties of aliasing

The possibility of aliasing invalidates all sorts of useful forms of reasoning about programs, both by programmers...

e.g., function

$$\lambda r{:}\,Ref\ Nat.\,\lambda s{:}\,Ref\ Nat.\ \ (r := 2;\ s := 3;\ !r)$$

always returns 2 unless r and s are aliases.

...and by compilers:

Code motion out of loops, common subexpression elimination, allocation of variables to registers, and detection of uninitialized variables all depend upon the compiler knowing which objects a load or a store operation could reference.

High-performance compilers spend significant energy on *alias analysis* to try to establish when different variables cannot possibly refer to the same storage.

The benefits of aliasing

The problems of aliasing have led some language designers simply to disallow it (e.g., Haskell).

But there are good reasons why most languages do provide constructs involving aliasing:

- efficiency (e.g., arrays)
- "action at a distance" (e.g., symbol tables)
- shared resources (e.g., locks) in concurrent systems
- etc.

Example

$c = ref\ 0$

$\text{incc} = \lambda x{:}\,Unit.\,(c := succ(!\,c);!\,c)$

$\text{decc} = \lambda x{:}\,Unit.\,(c := pred(!\,c);!\,c)$

$\text{incc}\ unit$

$decc\ unit$

$\text{o} = \{\text{i} = incc, d = decc\}$

$let\ newcounter = o$

$\lambda_{.Unit}\cdot$

$let\ c\ =\ ref\ 0\ in$

$\text{let incc} = \lambda x{:}\,Unit.\,(c := succ(!\,c);!\,c)\ \text{in}$

$\text{let decc} = \lambda x{:}\,Unit.\,(c := pred(!\,c);!\,c)$

$\text{let o} = \{\text{i} = \text{incc}, d = decc\}\ \text{in}$

o

Syntax

t ::=		*terms*
	unit	*unit constant*
	x	*variable*
	λx:T.t	*abstraction*
	t t	*application*
	ref t	*reference creation*
	!t	*dereference*
	t:=t	*assignment*

... plus other familiar types, in examples.

Typing rules

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

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

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

Example

```
NatArray = Ref (Nat→Nat);

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

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

update = λa:NatArray. λm:Nat. λv:Nat.
           let oldf = !a in
           a := (λn:Nat. if equal m n then v else oldf n);
       : NatArray → Nat → Nat → Unit
```

Evaluation

What is the value of the expression ref 0 ?

Crucial observation: evaluating ref 0 must *do* something ?

Otherwise,

```
r = ref 0
s = ref 0
```

and

```
r = ref 0
s = r
```

would behave the same.

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

So *what* is a reference?

The store

A reference names a *location* in the *store* (also known as the *heap* or just the *memory*).

What is the store?

- *Concretely*: An array of 8-bit bytes, indexed by 32-bit integers.
- *More abstractly*: an array of values.
- *Even more abstractly*: a partial function from locations to values.

Locations

Syntax of values:

v ::=		*values*
	unit	*unit constant*
	$\lambda x{:}T.t$	*abstraction value*
	l	*store location*

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

Syntax of Terms

t ::=		*terms*
	unit	*unit constant*
	x	*variable*
	λx:T.t	*abstraction*
	t t	*application*
	ref t	*reference creation*
	!t	*dereference*
	t:=t	*assignment*
	l	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 "source language" to include some run-time structures, so that we can continue to formalize evaluation as a relation between source terms.

Aside: If we formalize evaluation in the big-step style, then we can add locations to *the set of values* (results of evaluation) without adding them to the set of 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 μ to range over stores.

Evaluation

An assignment $t_1 := t_2$ first evaluates t_1 and t_2 until they become values...

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

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

... and then returns unit and updates the store:

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

Evaluation

A term of the form $\mathtt{ref}\, \mathtt{t}_1$ first evaluates inside $\mathtt{t}_1$ until it becomes a value...

$$\frac{\mathtt{t}_1 \mid \mu \longrightarrow \mathtt{t}_1' \mid \mu'}{\mathtt{ref}\ \mathtt{t}_1 \mid \mu \longrightarrow \mathtt{ref}\ \mathtt{t}_1' \mid \mu'} \qquad (\text{E-REF})$$

... and then chooses (allocates) a fresh location l, augments the store with a binding from l to $\mathtt{v}_1$, and returns l :

$$\frac{l \notin dom(\mu)}{\mathtt{ref}\ \mathtt{v}_1 \mid \mu \longrightarrow l \mid (\mu, l \mapsto \mathtt{v}_1)} \qquad (\text{E-REFV})$$

Evaluation

A term $!t_1$ first evaluates in t_1 until it becomes a value...

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

... 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

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

Evaluation

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

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

$$\frac{t_2 \mid \mu \longrightarrow t_2' \mid \mu'}{v_1\ t_2 \mid \mu \longrightarrow v_1\ t_2' \mid \mu'} \quad \text{(E-App2)}$$

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

Aside

Garbage Collection

Note that we are not modeling garbage collection — the store just grows without bound.

Pointer Arithmetic

We can’t do any!

Store Typings

Typing Locations

Question: What is the *type of a location*?

Answer: It depends on the store!

e.g., in the store $(l_1 \longmapsto \text{unit}, l_2 \longmapsto \text{unit})$, the term $!\,l_2$ has type Unit.

But in the store $(l_1 \longmapsto \text{unit}, l_2 \longmapsto \lambda x{:}\,\text{Unit}.\,x)$, the term $!\,l_2$ has type $\text{Unit} \to \text{Unit}$.

Typing Locations — first try

Roughly:

$$\frac{\Gamma \vdash \mu(l) : \mathtt{T}_1}{\Gamma \vdash l : \mathtt{Ref}\ \mathtt{T}_1}$$

More precisely:

$$\frac{\Gamma \mid \mu \vdash \mu(l) : \mathtt{T}_1}{\Gamma \mid \mu \vdash l : \mathtt{Ref}\ \mathtt{T}_1}$$

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

$$
\begin{aligned}
(\mu = {} & l_1 \mapsto \lambda \mathtt{x{:}Nat.}\ \ 999, \\
& l_2 \mapsto \lambda \mathtt{x{:}Nat.}\ \ !l_1\ (!l_1\ \mathtt{x}), \\
& l_3 \mapsto \lambda \mathtt{x{:}Nat.}\ \ !l_2\ (!l_2\ \mathtt{x}), \\
& l_4 \mapsto \lambda \mathtt{x{:}Nat.}\ \ !l_3\ (!l_3\ \mathtt{x}), \\
& l_5 \mapsto \lambda \mathtt{x{:}Nat.}\ \ !l_4\ (!l_4\ \mathtt{x})),
\end{aligned}
$$

then how big is the typing derivation for $!\,l_5$?

Problem

But wait... it gets worse. Suppose

$$(\mu = l_1 \mapsto \lambda \mathtt{x}{:}\mathtt{Nat}.\ \ !\, l_2\ \ \mathtt{x},$$
$$l_2 \mapsto \lambda \mathtt{x}{:}\mathtt{Nat}.\ \ !\, l_1\ \ \mathtt{x}),$$

how big is the typing derivation for $!\, l_2$?

Store Typings

Observation: The typing rules we have chosen for references guarantee that 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.

Store Typings

E.g., for

$$\begin{aligned}\mu = (&l_1 \mapsto \lambda x{:}\mathtt{Nat}.\ 999,\\ &l_2 \mapsto \lambda x{:}\mathtt{Nat}.\ !l_1\ (!l_1\ x),\\ &l_3 \mapsto \lambda x{:}\mathtt{Nat}.\ !l_2\ (!l_2\ x),\\ &l_4 \mapsto \lambda x{:}\mathtt{Nat}.\ !l_3\ (!l_3\ x),\\ &l_5 \mapsto \lambda x{:}\mathtt{Nat}.\ !l_4\ (!l_4\ x)),\end{aligned}$$

A reasonable store typing would be

$$\begin{aligned}\Sigma = (&l_1 \mapsto \mathtt{Nat}\to\mathtt{Nat},\\ &l_2 \mapsto \mathtt{Nat}\to\mathtt{Nat},\\ &l_3 \mapsto \mathtt{Nat}\to\mathtt{Nat},\\ &l_4 \mapsto \mathtt{Nat}\to\mathtt{Nat},\\ &l_5 \mapsto \mathtt{Nat}\to\mathtt{Nat})\end{aligned}$$

Store Typings

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

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

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

Final typing rules

$$\frac{\Sigma(l) = \mathtt{T}_1}{\Gamma \mid \Sigma \vdash l : \mathtt{Ref}\ \mathtt{T}_1} \qquad \text{(T-Loc)}$$

$$\frac{\Gamma \mid \Sigma \vdash \mathtt{t}_1 : \mathtt{T}_1}{\Gamma \mid \Sigma \vdash \mathtt{ref}\ \mathtt{t}_1 : \mathtt{Ref}\ \mathtt{T}_1} \qquad \text{(T-Ref)}$$

$$\frac{\Gamma \mid \Sigma \vdash \mathtt{t}_1 : \mathtt{Ref}\ \mathtt{T}_{11}}{\Gamma \mid \Sigma \vdash !\mathtt{t}_1 : \mathtt{T}_{11}} \qquad \text{(T-Deref)}$$

$$\frac{\Gamma \mid \Sigma \vdash \mathtt{t}_1 : \mathtt{Ref}\ \mathtt{T}_{11} \qquad \Gamma \mid \Sigma \vdash \mathtt{t}_2 : \mathtt{T}_{11}}{\Gamma \mid \Sigma \vdash \mathtt{t}_1 := \mathtt{t}_2 : \mathtt{Unit}} \qquad \text{(T-Assign)}$$

Store Typings

Question: Where do these store typings come from?

Answer: When we first typecheck a program, there will be no explicit locations, so we can use *an empty store typing*.

So, when a new location is created during evaluation,

$$\frac{l \notin dom(\mu)}{\texttt{ref}\ \ v_1 \mid \mu \longrightarrow l \mid (\mu, l \mapsto v_1)} \qquad (\text{E-RefV})$$

we can observe the type of v_1 and extend the "current store typing" appropriately.

Safety

Preservation

First attempt: just add *stores* and *store typings* in the appropriate places.

Theorem(?): if $\Gamma \mid \Sigma \vdash t : T$ and if $t|\mu \longrightarrow t'|\mu'$, then $\Gamma \mid \Sigma \vdash t' : T$

Right?? Wrong!

Why is this wrong?

Because Σ and μ here are not constrained to have anything to do with each other!

Preservation

A store μ is said to be *well typed* with respect to a typing context Γ and a store typing Σ, written $\Gamma \mid \Sigma \vdash \mu$, if $dom(\mu) = dom(\Sigma)$ and $\Gamma \mid \Sigma \vdash \mu(l) : \Sigma(l)$ for every $l \in dom(\mu)$.

Theorem (?) : if

$\Gamma \mid \Sigma \vdash t : T$

$t \mid \mu \longrightarrow t' \mid \mu'$

$\Gamma \mid \Sigma \vdash \mu$

then $\Gamma \mid \Sigma \vdash t' : T$

Still wrong !

Preservation

Creation of a new reference cell...

$$\frac{l \notin dom(\mu)}{\text{ref } \mathrm{v}_1 \mid \mu \longrightarrow l \mid (\mu, l \mapsto \mathrm{v}_1)} \qquad \text{(E-RefV)}$$

... breaks the correspondence between the store typing and the store.

Preservation

Theorem: if

$$\Gamma \mid \Sigma \vdash t : T$$

$$\Gamma \mid \Sigma \vdash \mu$$

$$t \mid \mu \longrightarrow t' \mid \mu$$

then, for some $\Sigma' \supseteq \Sigma$,

$$\Gamma \mid \Sigma' \vdash t' : T$$

$$\Gamma \mid \Sigma' \vdash \mu'.$$

A correct version.

Proof: Easy extension of the preservation proof for $\lambda_{\rightarrow}$.

Progress

Theorem: Suppose t is a closed, well-typed term (that is, $\emptyset \mid \Sigma \vdash t : T$ 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' with $t \mid \mu \longrightarrow t' \mid \mu'$.

Nontermination via references

There are well-typed terms in this system that are not strongly normalizing. For example:

$$\begin{aligned} t1 = {} & \lambda r\text{: Ref (Unit} \to \text{Unit).} \\ & (r := (\lambda x\text{: Unit. (! r)x}); \\ & (!\, r)\ \text{unit}); \\ t2 = {} & \text{ref}\ (\lambda x\text{: Unit. x}); \end{aligned}$$

Applying t1 to t2 yields a (well-typed) divergent term.

Recursion via references

Indeed, we can define arbitrary recursive functions using references.

1. Allocate a ref cell and initialize it with a dummy function of the appropriate type:

$$\text{fact}_{ref} = \text{ref}\ (\lambda \text{n: Nat. } 0)$$

2. Define the body of the function we are interested in, using the contents of the reference cell for making recursive calls:

$$\text{fact}_{body} =$$
$$\lambda \text{n: Nat.}$$
$$\text{if iszero n then 1 else times n } ((!\,\text{fact}_{ref})(\text{pred n}))$$

3. "Backpatch" by storing the real body into the reference cell:

$$\text{fact}_{ref} := \text{fact}_{body}$$

4. Extract the contents of the reference cell and use it as desired:

$$\text{fact} = !\,\text{fact}_{ref}$$
$$\text{fact } 5$$

Homework☺

- Read chapter 13
- Preview chapter 14
- Read and chew over the codes of chap 10.

- HW: 13.5.2