编程语言的设计原理
Design Principles of Programming Languages

Haiyan Zhao, Di Wang

赵海燕，王迪

Peking University, Spring Term 2025

Chapter 13: Reference

Why reference

Evaluation

Typing

Store Typings

Safety

Reference

Basic operations

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

Is there any difference between the expressions of ?

- 5 + 3;
- r: = 8;
- (r:=succ(!r); !r)
- (r:=succ(!r); (r:=succ(!r); (r:=succ(!r); !r)

sequencing

Syntax

We added to $\lambda_{\rightarrow}$ (with Unit) syntactic forms for *creating*, *dereferencing*, and *assigning* reference cells, plus a new *type constructor* Ref.

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

Evaluation

What is the value of the expression ref 0 ?

Is

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

and

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

behave the same?

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

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 run-time *store* (also known as the *heap* or just the *memory*)

What is the **store**?

- *Concretely*: an array of *8-bit bytes*, indexed by 32/64-bit integers
- *More abstractly*: an array of *values,* abstracting away from the different sizes of the runtime representations of different values
- *Even more abstractly*: a *partial function* from ***locations*** to *values*
  - set of store locations

Locations

A reference is a *location*, an *abstract index* into the store

The result of evaluating *a ref expression* will be a fresh location

Syntax of *values*:

v ::=		*values*
	unit	*unit constant*
	λx: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, just as intermediate results of evaluation

- Enriching the "source language" to include some *runtime structures*, we can thus 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 (with references)

- *depends on the store* in which it is evaluated
- *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*** *as well as* ***a store*** to ***a reduced term*** and ***a new store***

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

To use the metavariable μ to *range over stores*

μ & μ' : states of the store before & after evaluation

Evaluation

A term of *the form* $\text{ref } t_1$

1. first *evaluates* inside t_1 *until it becomes a value* ...

$$\frac{t_1 \mid \mu \longrightarrow t_1' \mid \mu'}{\texttt{ref } t_1 \mid \mu \longrightarrow \texttt{ref } t_1' \mid \mu'} \qquad \text{(E-Ref)}$$

2. then evaluate ref itself, *chooses* (allocates) a *fresh location* l, *augments* the store with ***a binding*** from l to v_1 , and returns l :

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

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

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

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

Evaluation

An assignment $t_1 \coloneqq t_2$ first evaluates t_1 and t_2 in order *until they become values* ...

$$\frac{\mathtt{t}_1 \mid \mu \longrightarrow \mathtt{t}_1' \mid \mu'}{\mathtt{t}_1 \mathtt{:=} \mathtt{t}_2 \mid \mu \longrightarrow \mathtt{t}_1' \mathtt{:=} \mathtt{t}_2 \mid \mu'} \qquad \text{(E-Assign1)}$$

$$\frac{\mathtt{t}_2 \mid \mu \longrightarrow \mathtt{t}_2' \mid \mu'}{\mathtt{v}_1 \mathtt{:=} \mathtt{t}_2 \mid \mu \longrightarrow \mathtt{v}_1 \mathtt{:=} \mathtt{t}_2' \mid \mu'} \qquad \text{(E-Assign2)}$$

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

$$l \mathtt{:=} \mathtt{v}_2 \boxed{\mid \mu} \longrightarrow \mathtt{unit} \boxed{\mid [l \mapsto \mathtt{v}_2]\mu} \qquad \text{(E-Assign)}$$

Evaluation

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

$$\frac{\mathtt{t}_1 \mid \mu \longrightarrow \mathtt{t}_1' \mid \mu'}{\mathtt{t}_1\ \mathtt{t}_2 \mid \mu \longrightarrow \mathtt{t}_1'\ \mathtt{t}_2 \mid \mu'} \qquad \text{(E-APP1)}$$

$$\frac{\mathtt{t}_2 \mid \mu \longrightarrow \mathtt{t}_2' \mid \mu'}{\mathtt{v}_1\ \mathtt{t}_2 \mid \mu \longrightarrow \mathtt{v}_1\ \mathtt{t}_2' \mid \mu'} \qquad \text{(E-APP2)}$$

$$(\lambda \mathtt{x}{:}\mathtt{T}_{11}.\mathtt{t}_{12})\ \mathtt{v}_2 \boxed{\mid \mu} \longrightarrow [\mathtt{x} \mapsto \mathtt{v}_2]\mathtt{t}_{12} \boxed{\mid \mu} \quad \text{(E-APPABS)}$$

Aside

Garbage Collection

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

It may not be problematic for most *theoretical purposes*, whereas it is clear that for *practical purposes* some form of *deallocation* of unused storage must be provided

Pointer Arithmetic

```
p++;
```

Typing rules

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

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

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

- **type system**
  - ***a set of rules*** that *assigns a property* called *type* to the various "constructs" of a computer program, such as
  - *variables*, *expressions*, *functions* or *modules*

Store Typing

Typing Locations

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

Answer: Depends on the ***contents*** of the store!

e.g,

- in the store $(l_1 \mapsto \text{unit}, l_2 \mapsto \text{unit})$,
  the term $!\,l_2$ is evaluated to unit, having type Unit

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

Typing Locations — first try

Roughly, to find the type of a location l, first *look up* the current contents of l in the store, and calculate the type T_1 of the contents:

$$\frac{\Gamma \vdash \mu(l) : \mathtt{T_1}}{\Gamma \vdash l : \mathtt{Ref\ T_1}}$$

More precisely, to make *the type of a term depend on the store* (keeping a consistent state), we should change the *typing relation* from *three-place* to :

$$\frac{\Gamma \boxed{\mid \mu} \vdash \mu(l) : \mathtt{T_1}}{\Gamma \boxed{\mid \mu} \vdash l : \mathtt{Ref\ T_1}}$$

i.e., typing is now a *four-place relation* (about *contexts*, *stores*, *terms*, and *types*), though ***the store is a part of the context*** ……

Problems #1

However, this rule is not *completely satisfactory*, and is *rather inefficient*.

– it can make *typing derivations very large* (if a location *appears* ***many times*** in a term) !

– e.g.,

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

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

Problems #2

But wait... it *gets worse* if the store contains a *cycle*.

Suppose

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

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

Calculating a type for l_2 requires finding the type of l_1, which in turn involves l_2

Why?

What leads to the problems?

Our typing rule for locations requires us to *recalculate the type of a location every time it's* mentioned in a term, which *should not be necessary*

In fact, once a location is first created, *the type of the initial value* is **known**, and *the type will be kept* even if the values can be changed

Store Typing

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 Typing

E.g., for

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

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 Typing

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

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

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

Proviso: the typing rules ***accurately predict*** the results of evaluation *only if* the *concrete store* used during evaluation actually *conforms to* the store typing.

Final typing rules

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

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

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

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

Store Typing

Where do *these store typings* come from?

When we first typecheck a program, there will be *no explicit locations*, so we can use *an empty store typing*, since the locations arise only in terms that are *the intermediate results* of evaluation

So, when a new location is created during evaluation,

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

we can observe the type of $\mathtt{v}_1$ and *extend* the "*current store typing*" appropriately.

Store Typing

As evaluation proceeds and *new locations are created*, *the store typing is extended* by looking at the type of the initial values being placed in newly allocated cells

Σ only records the *association*

between

already-allocated storage cells and

their types

Safety

Coherence between

the statics and the dynamics

Well-formed programs are well-behaved

when executed

Preservation

the steps of evaluation

preserve typing

Preservation

How to express the statement of preservation?

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

Theorem(first try**):** if $\Gamma \mid \Sigma \vdash t : T$ and $t|\mu \longrightarrow t'|\mu'$,

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

Right??

Wrong! Why ?

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

Exercise: Construct an example that breaks this statement of preservation

Preservation

Definition: 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 $\Gamma \mid \Sigma \vdash \mu(l)\colon\ \Sigma(l)$ for every $l \in dom(\mu)$

Theorem (snd try) : if

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

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

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

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

Right this time?

Still wrong !

Why? Where? (E-REFV） 13.5.2

Preservation

Creation of a *new reference cell* ...

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

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

Since *the store can grow during evaluation*:

Creation of a new reference cell yields a store with a *larger domain* than the initial one, making the conclusion *incorrect*: if μ' includes a binding for ***a fresh location*** l , then l cann't be in the domain of Σ , and it will not be the case that t' ***is typable*** *under* $\boldsymbol{\Sigma}$

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.

What is Σ' ?

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

Preservation

Lemma (Substitution)

if $\Gamma, x{:}\, S \mid \Sigma \vdash t{:}\, T$ and $\Gamma \mid \Sigma \vdash s{:}\, S$, then $\Gamma \mid \Sigma \vdash [x \mapsto s]\, t : T$

Lemma (updating contents of a cell don't change the overall type of the store) : if

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

$\Sigma(l) = T$

$\Gamma \mid \Sigma \vdash v{:}\, T$

then $\Gamma \mid \Sigma \vdash [l \mapsto v]\mu$

Lemma (preserving type in the extended store)

if $\Gamma \mid \Sigma \vdash t{:}\, T$ and $\Sigma' \supseteq \Sigma$ then $\Gamma \mid \Sigma' \vdash t{:}\, T$

Progress

well-typed expressions are

either *values*

or can be *further evaluated*

Progress

Theorem:

Suppose t is a closed, well-typed term

(i.e., $\varnothing \mid \Sigma \vdash \mathrm{t} : \mathrm{T}$ for some T and Σ)

then either t is a *value* or else, for any store μ such that $\Gamma \mid \Sigma \vdash \mu$, there is some term t' and store μ' with

$$\mathrm{t} \mid \mu \longrightarrow \mathrm{t}' \mid \mu'$$

Safety

- Preservation and progress together constitute the proof of safety
  - progress theorem ensures that well-typed expressions don't get stuck in an ill-defined state, and
  - preservation theorem ensures that if a step is a taken the result remains well-typed (*with the same type*).
- These two parts ensure the *statics and dynamics* are coherent, and that no ill-defined states can ever be encountered while evaluating a well-typed expression

In summary …

Syntax

We added to $\lambda_{\rightarrow}$ (with Unit) syntactic forms for *creating*, *dereferencing*, and *assigning* reference cells, plus a new type constructor Ref.

$\mathtt{t}$ ::=		*terms*
	$\mathtt{unit}$	*unit constant*
	$\mathtt{x}$	*variable*
	$\lambda\mathtt{x}\!:\!\mathtt{T}.\mathtt{t}$	*abstraction*
	$\mathtt{t}\ \mathtt{t}$	*application*
	$\mathtt{ref}\ \mathtt{t}$	*reference creation*
	$\mathtt{!t}$	*dereference*
	$\mathtt{t}\!:=\!\mathtt{t}$	*assignment*
	l	store location

Evaluation

Evaluation relation: $t \mid \mu \longrightarrow t' \mid \mu'$

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

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

$$l := \mathtt{v}_2 \mid \mu \longrightarrow \texttt{unit} \mid [l \mapsto \mathtt{v}_2]\mu \qquad \text{(E-Assign)}$$

Typing

Typing becomes *a four-place* relation: $\Gamma \mid \Sigma \vdash \mathtt{t} : \mathtt{T}$

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

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

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

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

Preservation

Theorem: if

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

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

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

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

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

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

Progress

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

Others ...

Arrays

Fix-sized vectors of values.

All of the values must have the *same type*, and the fields in the array can be accessed and modified.

e.g., arrays can be created in Ocaml, as

$$[|e_1; \dots ; e_n|]$$

```
# let a = [|1;3;5;7;9|];;
val a : int array = [|1;3;5;7;9|]
#a;;
-: int array = [|1;3;5;7;9|]
```

Arrays

```
let  f a =
   for i = 1 to Array.length a - 1 do
       let val_i = a.(i) in
       let j = ref i in
       while !j > 0 && val_i < a.(!j - 1) do
          a.(!j)  <-  a.(!j - 1);
          j := !j - 1
      done;
      a.(!j) <- val_i
  done;;
```

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:

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

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

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

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:

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

Nontermination via references

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

For example:

$$t1 = \lambda r\!: \mathrm{Ref}\ (\mathrm{Unit} \to \mathrm{Unit}).\ \boxed{(r := (\lambda x\!: \mathrm{Unit}.\ (!\,r)\ x);\ (!\,r)\ \mathrm{unit});}$$

$$t2 = \mathrm{ref}\ (\lambda x\!: \mathrm{Unit}.\ x);$$

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

Homework☺

- Read chapter 13
- Read and chew over the codes of *fullref*.

- HW: 13.1.2 & 13.3.1