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 2026

Chapter 13: Reference

Why reference

Evaluation

Typing

Store Typings

Safety

Why & What References

Computational Effects

Also known as *side effects*.

A *function* or *expression* is said to have a **side effect** if, in addition to returning a value, it also ***modifies some state*** or has an ***observable interaction with*** calling functions or the outside world.

- modify a *global variable* or *static variable*, modify *one of its arguments*,
- *raise an exception,*
- *write* data to *a display* or file, *read* data, or
- call other *side-effecting functions*.

In the presence of side effects, a program's behavior may depend on *history*; i.e., the *order of evaluation* matters.

Computational Effects

Side effects are the *most common way* that a program *interacts with the outside world* (*people*, *file systems*, *other computers on networks*).

The degree to which *side effects are used* depends on the *programming paradigm*.

- *Imperative programming* is known for *its frequent utilization* of side effects.
- In *functional programming*, side effects are rarely used.
  - Functional languages like *Standard ML*, *Scheme* and *Scala* do not restrict side effects, but it is customary for programmers to avoid them.
  - The functional language *Haskell* expresses side effects such as I/O and other stateful computations using *monadic* actions.

Mutability

So far, what we have discussed does not yet include *side effects* .

In particular, whenever we defined function, we *never changed variables or data*. Rather, we always computed *new data*.

- E.g., 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

Writing values into memory locations is the **fundamental mechanism** of imperative languages such as C/C++.

Mutable structures are

- required to implement many *efficient algorithms*.
- also very convenient to represent the *current state of a state machine*.

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

Basic Examples

```
#let r = ref 5
val r : int ref = {contents = 5}
//  The value of r is a reference to a cell that always contain a number.

# r := !r +3
 ??
# !r
-: int = 8

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

Basic Examples

```
# let flag = ref true;;
-val flag: bool ref = {contents = true}

# if !flag then 1 else 2;;
-: int = 1
```

Reference

Basic operations

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

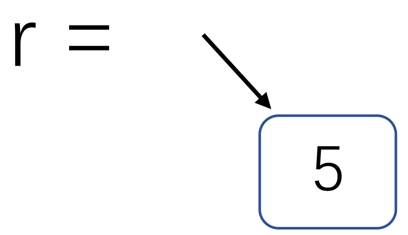
Is there any difference between the expressions of ?

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

sequencing

Reference

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



Exercise 13.1.1 :

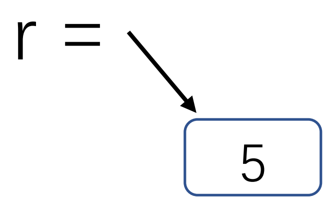
Draw a similar diagram showing the effects of evaluating the expressions

a = {ref 0, ref 0}

b = (λx:Ref Nat. {x, x}) (ref 0)

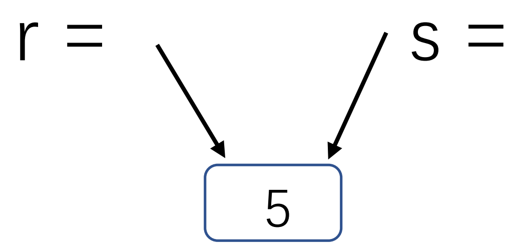
Aliasing

A value of type ref T is *a pointer* to a cell holding a value of type T



If this value is "*copied*" by assigning it to another variable: s = r;

the cell pointed to is not copied.   (*r* and *s* are *aliases*)



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 (shared state—between different parts of a program.)
- 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., $\lambda r{:}Ref\ Nat.\lambda s{:}Ref\ Nat.(r := 2;\ s := 3;\ !r)$

  always returns 2 unless r and s are aliases

  and *by compilers* :

  Code motion out of loops, *common sub-expression 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).

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

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

Example

$c = ref\ 0$

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

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

$\text{incc}\ unit$

$decc\ unit$

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

$let\ newcounter = o$

$\quad \lambda_{.Unit}.\ \text{let}\ c\ =\ ref\ 0\ \text{in}$

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

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

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

$\qquad\quad \text{o}$

Example

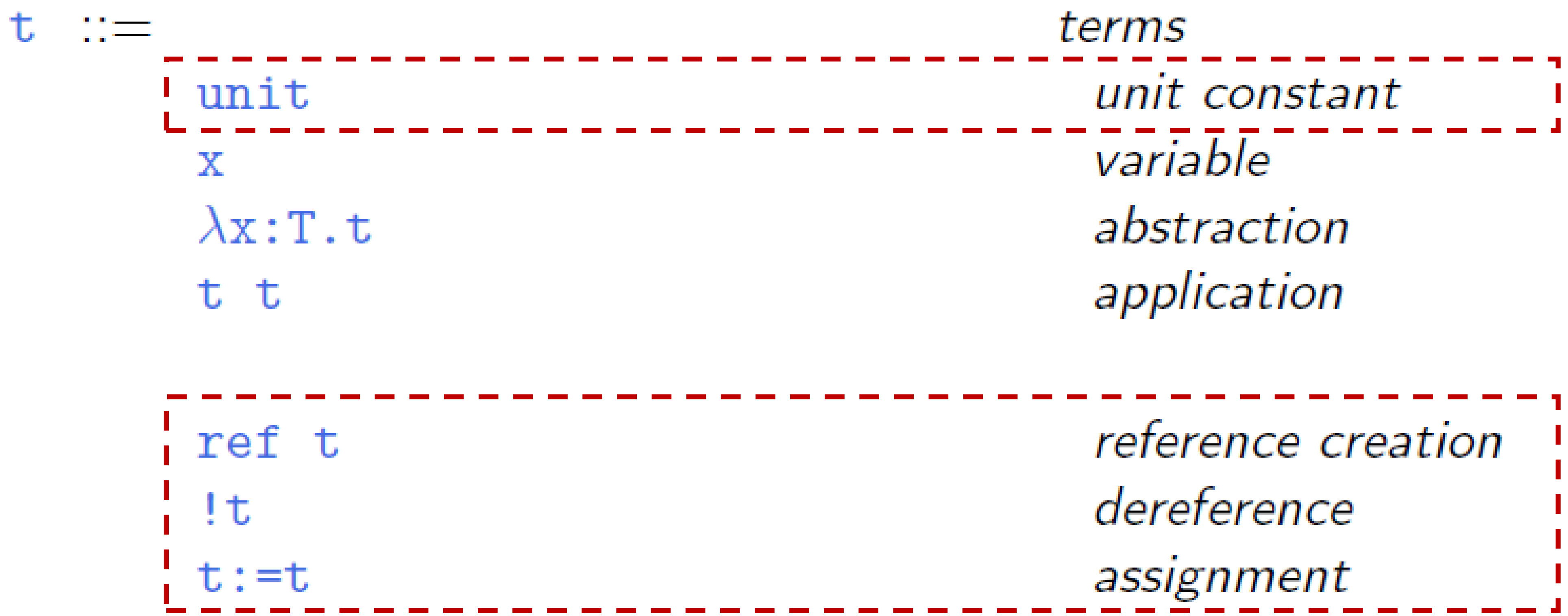
- Reference values of any type, including functions.

```
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
```

How to enrich the language with the new mechanism?

Syntax

... plus other familiar types, in examples



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

Typing rules

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

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

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

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

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 storagea

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

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***

$$\mathrm{t} \mid \mu \;\rightarrow\; \mathrm{t}' \mid \mu'$$

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:

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

Evaluation

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

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

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

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

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++;
```

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 $(l_1 \longmapsto \text{unit}, l_2 \longmapsto \text{unit})$,
  the term $!\,l_2$ is evaluated to unit, having type $\mathbf{Unit}$
- in the store $(l_1 \longmapsto \text{unit}, l_2 \longmapsto \lambda x\!:\!\text{Unit}.\,x)$,
  the term $!\,l_2$ has type $\mathbf{Unit} \rightarrow \mathbf{Unit}$

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:

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

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:

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

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.,

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

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

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

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

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

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

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

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

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,

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

we can observe the type of $\texttt{v}_1$ 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

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 $\Gamma \mid \Sigma \vdash \mu(l){:}\ \Sigma(l)$ for every $l \in dom(\mu)$

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* ...

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

... *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

$\Gamma \mid \Sigma \vdash \mathrm{t} : \mathrm{T}$

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

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

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

$\Gamma \mid \Sigma' \vdash \mathrm{t}' : \mathrm{T}$

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

A correct version.

What is Σ' ?

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

Preservation

Lemma (Substitution)

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

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

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

$\Sigma(l) = \mathrm{T}$

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

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

Lemma (preserving type in the extended store)

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

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

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

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

```
# 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:

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

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

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

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

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

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

$$\mathrm{fact} \ = \ !\,\mathrm{fact}_{ref}$$

Nontermination via references

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

For example:

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

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

Applying $\mathrm{t1}$ to $\mathrm{t2}$ yields a (well-typed) divergent term.

Metatheory: Key Properties

- **Expressive Power**: from pure to impure
  - **Pure Functions:** No side effects, Deterministic: Same input always gives same output.
  - **Impure Functions:** May have side effects (state changes). Non-deterministic output possible.

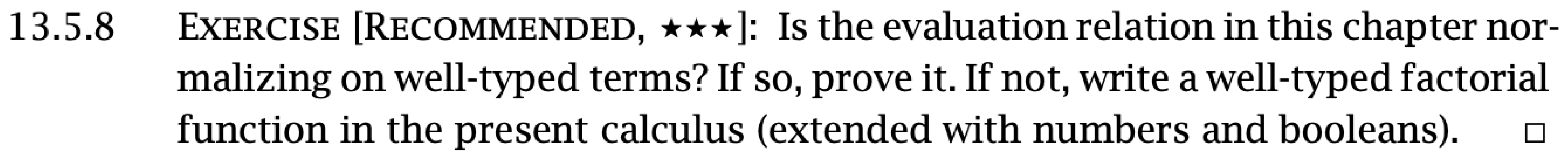
- **Loss of Strong Normalization**
  - The language is **no longer strongly normalizing, we** can now write non-terminating programs (e.g., infinite loops with recursion).

- **Fundamental Trade-off**
  - We gain expressiveness (mutable state) at the cost of losing strong normalization, a classic trade-off in language design.

Homework☺

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

- HW: 13.5.8



13.5.8 EXERCISE [RECOMMENDED, ★★★]: Is the evaluation relation in this chapter normalizing on well-typed terms? If so, prove it. If not, write a well-typed factorial function in the present calculus (extended with numbers and booleans). □