



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 2024



Recap

Simply typed lambda calculus

Syntax

 $t ::=$

$$\lambda x:T. t$$

$$t t$$

terms:

variable

abstraction

application

 $v ::=$

$$\lambda x:T. t$$

values:

abstraction value

 $T ::=$

$$T \rightarrow T$$

types:

type of functions

 $\Gamma ::=$ \emptyset $\Gamma, x:T$

contexts:

empty context

term variable binding

Assume: all variables in Γ are different
via renaming/internal

Evaluation

 $t \rightarrow t'$

$$\frac{t_1 \rightarrow t'_1}{t_1 t_2 \rightarrow t'_1 t_2} \quad (\text{E-APP1})$$

$$\frac{t_2 \rightarrow t'_2}{v_1 t_2 \rightarrow v_1 t'_2} \quad (\text{E-APP2})$$

$$(\lambda x:T_{11}. t_{12}) v_2 \rightarrow [x \mapsto v_2]t_{12} \quad (\text{E-APPABS})$$

Typing

 $\Gamma \vdash t : T$

$$\frac{x:T \in \Gamma}{\Gamma \vdash x : T} \quad (\text{T-VAR})$$

$$\frac{\Gamma, x:T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x:T_1. t_2 : T_1 \rightarrow T_2} \quad (\text{T-ABS})$$

$$\frac{\Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \quad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1 t_2 : T_{12}} \quad (\text{T-APP})$$

The Unit Type

- It is the singleton type (like void in C).

New syntactic forms

$t ::= \dots$
`unit`

terms:
constant unit

$v ::= \dots$
`unit`

values:
constant unit

$T ::= \dots$
`Unit`

types:
unit type

New typing rules

$\Gamma \vdash \text{unit} : \text{Unit}$

$\Gamma \vdash t : T$

(T-UNIT)

New derived forms

$t_1 ; t_2 \stackrel{\text{def}}{=} (\lambda x : \text{Unit}. t_2) t_1$
where $x \notin FV(t_2)$

- Application: Unit-type expressions care more about “**side effects**” rather than “results”.

Derived Form: Sequencing $t_1 ; t_2$

- A direct extension λ^E

- $t ::= \dots$

- $t_1 ; t_2$

- New evaluation relation rules

$$\frac{t_1 \rightarrow t'_1}{t_1 ; t_2 \rightarrow t'_1 ; t_2} \quad (\text{E-SEQ})$$

$$\text{unit} ; t_2 \rightarrow t_2 \quad (\text{E-SEQNEXT})$$

- New typing rules

$$\frac{\Gamma \vdash t_1 : \text{Unit} \quad \Gamma \vdash t_2 : T_2}{\Gamma \vdash t_1 ; t_2 : T_2} \quad (\text{T-SEQ})$$



Ascription

New syntactic forms

$t ::= \dots$
 $t \text{ as } T$

terms

ascription

New evaluation rules

$v_1 \text{ as } T \longrightarrow v_1$

(E-ASCRIIBE)

$$\frac{t_1 \longrightarrow t'_1}{t_1 \text{ as } T \longrightarrow t'_1 \text{ as } T}$$

(E-ASCRIIBE1)

New typing rules

$$\frac{\Gamma \vdash t_1 : T}{\Gamma \vdash t_1 \text{ as } T : T}$$

(T-ASCRIIBE)

verification

Ascription as a derived form

$t \text{ as } T \stackrel{\text{def}}{=} (\lambda x:T. x) t$

Let Bindings

- Give names to some of its subexpressions

New syntactic forms

$t ::= \dots$
 $\text{let } x=t \text{ in } t$

terms
let binding

New evaluation rules

$\text{let } x=v_1 \text{ in } t_2 \longrightarrow [x \mapsto v_1]t_2$ (E-LETV)
 $\frac{t_1 \longrightarrow t'_1}{\text{let } x=t_1 \text{ in } t_2 \longrightarrow \text{let } x=t'_1 \text{ in } t_2}$ (E-LET)

New typing rules

$\frac{\Gamma \vdash t_1 : T_1 \quad \Gamma, x:T_1 \vdash t_2 : T_2}{\Gamma \vdash \text{let } x=t_1 \text{ in } t_2 : T_2}$ (T-LET)

Records

New syntactic forms

$t ::= \dots$
 $\{\lambda_i = t_i \mid i \in 1..n\}$
 $t.l$

terms:
 record
 projection

$v ::= \dots$
 $\{\lambda_i = v_i \mid i \in 1..n\}$

values:
 record value

$T ::= \dots$
 $\{\lambda_i : T_i \mid i \in 1..n\}$

types:
 type of records

New evaluation rules

$t \rightarrow t'$

$\{\lambda_i = v_i \mid i \in 1..n\}.l_j \rightarrow v_j$ (E-PROJRCD)

$\frac{t_1 \rightarrow t'_1}{t_1.l \rightarrow t'_1.l}$ (E-PROJ)

$\frac{t_j \rightarrow t'_j}{\{\lambda_i = v_i \mid i \in 1..j-1, \lambda_j = t_j, \lambda_k = t_k \mid k \in j+1..n\} \rightarrow \{\lambda_i = v_i \mid i \in 1..j-1, \lambda_j = t'_j, \lambda_k = t_k \mid k \in j+1..n\}}$ (E-RCD)

New typing rules $\Gamma \vdash t : T$

$\frac{\text{for each } i \quad \Gamma \vdash t_i : T_i}{\Gamma \vdash \{\lambda_i = t_i \mid i \in 1..n\} : \{\lambda_i : T_i \mid i \in 1..n\}}$ (T-RCD)

$\frac{\Gamma \vdash t_1 : \{\lambda_i : T_i \mid i \in 1..n\}}{\Gamma \vdash t_1.l_j : T_j}$ (T-PROJ)

Question: $\{\text{partno}=5524, \text{cost}=30.27\} = \{\text{cost}=30.27, \text{partno}=5524\}?$

Variants

New syntactic forms

$t ::= \dots$

$\langle l=t \rangle \text{ as } T$

$\text{case } t \text{ of } \langle l_i=x_i \rangle \Rightarrow t_i^{i \in 1..n}$

terms:

tagging

case

$T ::= \dots$

$\langle l_i : T_i^{i \in 1..n} \rangle$

types:

type of variants

New evaluation rules

$t \rightarrow t'$

$\text{case } (\langle l_j=v_j \rangle \text{ as } T) \text{ of } \langle l_i=x_i \rangle \Rightarrow t_i^{i \in 1..n} \rightarrow [x_j \mapsto v_j]t_j$

(E-CASEVARIANT)

$$\frac{t_0 \rightarrow t'_0}{\text{case } t_0 \text{ of } \langle l_i=x_i \rangle \Rightarrow t_i^{i \in 1..n} \rightarrow \text{case } t'_0 \text{ of } \langle l_i=x_i \rangle \Rightarrow t_i^{i \in 1..n}}$$

(E-CASE)

$$\frac{t_i \rightarrow t'_i}{\langle l_i=t_i \rangle \text{ as } T \rightarrow \langle l_i=t'_i \rangle \text{ as } T}$$

(E-VARIANT)

New typing rules

$\Gamma \vdash t : T$

$$\frac{\Gamma \vdash t_j : T_j}{\Gamma \vdash \langle l_j=t_j \rangle \text{ as } \langle l_i : T_i^{i \in 1..n} \rangle : \langle l_i : T_i^{i \in 1..n} \rangle}$$

(T-VARIANT)

$$\frac{\Gamma \vdash t_0 : \langle l_i : T_i^{i \in 1..n} \rangle \text{ for each } i \quad \Gamma, x_i : T_i \vdash t_i : T}{\Gamma \vdash \text{case } t_0 \text{ of } \langle l_i=x_i \rangle \Rightarrow t_i^{i \in 1..n} : T}$$

(T-CASE)

General Recursions

- Introduce “fix” operator : $\text{fix } f = f (\text{fix } f)$
 - It cannot be defined as a derived form in simply typed lambda calculus

New syntactic forms

$t ::= \dots$
 $\text{fix } t$

terms
fixed point of t

New evaluation rules

$$\frac{\text{fix } (\lambda x:T_1. t_2)}{\longrightarrow [x \mapsto (\text{fix } (\lambda x:T_1. t_2))]t_2} \quad (\text{E-FIXBETA})$$

$$\frac{t_1 \longrightarrow t'_1}{\text{fix } t_1 \longrightarrow \text{fix } t'_1} \quad (\text{E-FIX})$$



General Recursions

New typing rules

$$\boxed{\Gamma \vdash t : T}$$

$$\frac{\Gamma \vdash t_1 : T_1 \rightarrow T_1}{\Gamma \vdash \text{fix } t_1 : T_1} \quad (\text{T-FIX})$$

A convenient form

$$\text{letrec } x:T_1=t_1 \text{ in } t_2 \stackrel{\text{def}}{=} \text{let } x = \text{fix } (\lambda x:T_1.t_1) \text{ in } t_2$$



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 ?

- $5 + 3;$
- $r := 8;$
- $(r := \text{succ}(!r); !r)$
- $(r := \text{succ}(!r); (r := \text{succ}(!r); (r := \text{succ}(!r); !r))$

sequencing



Reference

Exercise 13.1.1 :

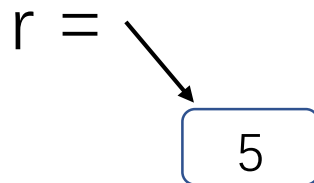
Draw a similar diagram showing the effects of evaluating the expressions

$a = \{\text{ref } 0, \text{ref } 0\}$ and

$b = (\lambda x:\text{Ref Nat. } \{x,x\}) (\text{ref } 0)$

Aliasing

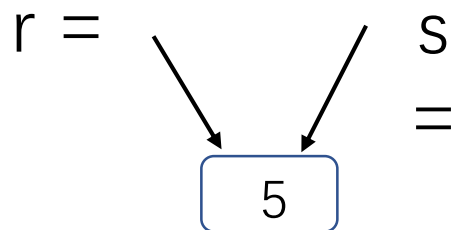
A value of type `ref T` is a *pointer* to a cell holding a value of type `T`



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
- 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

$c = \text{ref } 0$

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

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

$\text{incc } \text{unit}$

$\text{decc } \text{unit}$

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

```
let newcounter = o
  λ.Unit.
    let c = ref 0 in
      let incc = λx: Unit. (c := succ(!c); !c) in
        let decc = λx: Unit. (c := pred(!c); !c)
        let o = {i = incc, d = decc} in
          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

<code>t ::=</code>	<i>terms</i>
<code>unit</code>	<i>unit constant</i>
<code>x</code>	<i>variable</i>
<code>λx:T.t</code>	<i>abstraction</i>
<code>t t</code>	<i>application</i>
<code>ref t</code>	<i>reference creation</i>
<code>!t</code>	<i>dereference</i>
<code>t:=t</code>	<i>assignment</i>

Typing rules

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

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

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

- **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*



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 *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
 - Location : an abstract index into the store

Locations

Syntax of *values*:

$v ::=$	
unit	<i>unit constant</i>
$\lambda x:T.t$	<i>abstraction value</i>
/	<i>store location</i>

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

Syntax of Terms

$t ::=$

`unit`

`x`

`$\lambda x:T.t$`

`t t`

`ref t`

`!t`

`t:=t`

`/`

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*, 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* ...

$$\frac{t_1 \mid \mu \longrightarrow t'_1 \mid \mu'}{\text{ref } t_1 \mid \mu \longrightarrow \text{ref } t'_1 \mid \mu'} \quad (\text{E-REF})$$

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

$$\frac{l \notin \text{dom}(\mu)}{\text{ref } v_1 \mid \mu \longrightarrow l \mid (\mu, [l \mapsto v_1])} \quad (\text{E-REFV})$$

Evaluation

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

$$\frac{t_1 \mid \mu \longrightarrow t'_1 \mid \mu'}{!t_1 \mid \mu \longrightarrow !t'_1 \mid \mu'} \quad (\text{E-DEREF})$$

... 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

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

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`:

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



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*

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

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

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 :

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

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

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

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

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

A reasonable *store typing* would be

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



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

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

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

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

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

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

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

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,

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

we can observe the type of 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(?): if $\Gamma \mid \Sigma \vdash t:T$ and $t \mid \mu \rightarrow t' \mid \mu'$,
then $\Gamma \mid \Sigma \vdash t':T$

Right??

Wrong! Why ?

Because Σ and μ here 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 $l \in dom(\mu)$

Theorem (?) : if

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

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

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

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* ...

$$\frac{l \notin \text{dom}(\mu)}{\text{ref } v_1 \mid \mu \rightarrow l \mid (\mu, l \mapsto v_1)} \quad (\text{E-REFV})$$

... *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 **can't be in the domain of** Σ , and it will not be the case that **t' is typable under** Σ



Preservation

Theorem: if

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

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

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

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 λ_{\rightarrow}

Progress

well-typed expressions are
either *values*
or can be *further evaluated*



Progress

Theorem:

Suppose t is a closed, well-typed term

(i.e., $\Gamma \mid \Sigma \vdash t : T$ 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

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



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 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 λ_{\rightarrow} (with `Unit`) syntactic forms for *creating*, *dereferencing*, and *assigning* reference cells, plus a new type constructor `Ref`.

<code>t ::=</code>	<i>terms</i>
<code>unit</code>	<i>unit constant</i>
<code>x</code>	<i>variable</i>
<code>$\lambda x:T.t$</code>	<i>abstraction</i>
<code>t t</code>	<i>application</i>
<code>ref t</code>	<i>reference creation</i>
<code>!t</code>	<i>dereference</i>
<code>t:=t</code>	<i>assignment</i>
<code>/</code>	<i>store location</i>



Evaluation

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

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

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

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

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

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

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

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

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



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 with in Ocaml

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

```
# 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:

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

$$\lambda n: \text{Nat}.$$

$$\text{if iszero } n \text{ then } 1 \text{ else times } n \text{ } (! \text{fact}_{ref})(\text{pred } n)$$

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:

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



Homework😊

- Read chapter 13
- Read and chew over the codes of *fullref*.
- HW: 13.5.2 and 13.5.8