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

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 :=

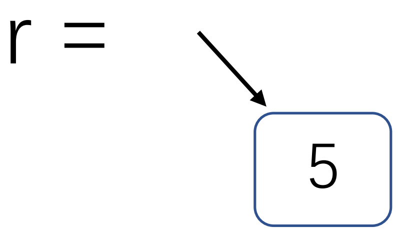
Is there any difference between the expressions of ?

- 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 :

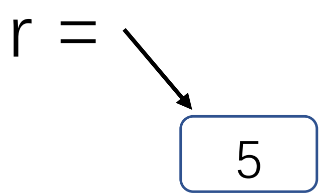
Draw a similar diagram showing the effects of evaluating the expressions

a = {ref 0, ref 0}

b = (λx:Ref Nat. {x, x}) (ref 0)

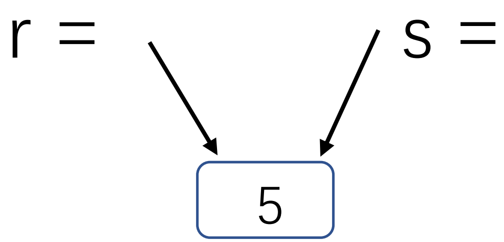
Aliasing

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



If this value is "*copied*" by assigning it to another variable: 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

$c = ref\ 0$

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

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

$\text{incc}\ unit$

$decc\ unit$

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

$let\ newcounter = o$

$\lambda_{.Unit}.$

$\text{let}\ c\ =\ ref\ 0\ \text{in}$

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

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

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

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

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

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

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

- **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 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* ...