

2 November

Fall 2005

Software Foundations

CIS 500

Recursion

---

## Recursion in $\lambda \leftarrow$

- ◆ In  $\lambda \leftarrow$ , all programs terminate. (Cf. Chapter 12.)
- ◆ Hence, untyped terms like **omega** and **fix** are not typable.
- ◆ But we can **extend** the system with a (typed) fixed-point operator...

```
ff =  $\lambda$ !e:Nat $\rightarrow$ Bool.  
       $\lambda$ x:Nat.  
      if iszero x then true  
      else if iszero (pred x) then false  
      else !e (pred (pred x));  
iseven = fix ff;  
iseven 7;
```

---

Example

*New syntactic forms*

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

*New evaluation rules*

$\text{fix } (\lambda x:T_1. t_2) \longrightarrow [x \mapsto (\text{fix } (\lambda x:T_1. t_2))]t_2$

(E-FIXBETA)

$$\frac{\text{fix } t_1 \longrightarrow \text{fix } t'_1}{t_1 \longrightarrow t'_1}$$

(E-FIX)

*terms*  
*fixed point of*  $t$

$t \longrightarrow t'$

*New typing rules*

$$\frac{\Gamma \vdash t_1 : T_1}{\Gamma \vdash \text{fix } t_1 : T_1}$$

(T-FIX)

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

## A more convenient form

```
def letrec x:T1=t1 in t2 = fix (λx:T1.t1) in t2
letrec iseven : Nat → Bool =
  λx:Nat.
    if iszero x then true
    else if iszero (pred x) then false
    else iseven (pred (pred x))
  in
  iseven 7;
```

## References



## 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 two features are kept 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
  ir
  r := 7
(r:=succ(ir); ir)
(r:=succ(ir); r:=succ(ir); r:=succ(ir); ir)
```

## Basic Examples

---

```
let r = ref 5  
  ir  
  r := 7
```

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

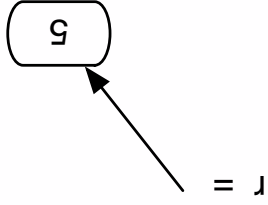
```
(r:=succ(ir); r:=succ(ir); r:=succ(ir); ir)
```

i.e.,

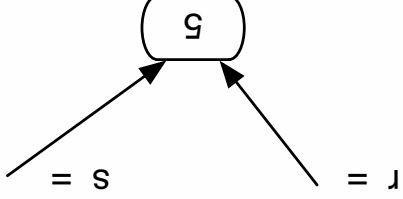
```
((r:=succ(ir); r:=succ(ir)); r:=succ(ir)); ir)
```

## 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, the cell pointed to is not copied.



So we can change **r** by assigning to **s**:  
`(s:=6; !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...

The function

```
λr:Ref Nat. λs:Ref Nat. (r:=2; s:=3; !r)
```

always returns **2** unless **r** and **s** are aliases for the same cell.

...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
inc =  $\lambda x:\text{Unit}. (c := \text{succ } (ic); ic)$ 
dec =  $\lambda x:\text{Unit}. (c := \text{pred } (ic); ic)$ 
inc unit
dec unit
o = {i = inc, d = dec}
```



```
let newcounter =
```

```
  λ_:Unit.
```

```
    let c = ref 0 in
```

```
      let inc = λx:Unit. (c := succ (ic); ic) in
```

```
      let dec = λx:Unit. (c := pred (ic); ic) in
```

```
      let o = {i = inc, d = dec} in
```

```
    o
```

# Syntax

$t ::=$	$t$
	unit
	$x$
	$\lambda x:T.t$
	$t \ t$
	ref $t$
	it
	$t := t$
terms	
unit constant	
variable	
abstraction	
application	
reference creation	
dereference	
assignment	

... plus other familiar types, in examples.

## Typing Rules

(T-REF)

$$\frac{\Gamma \vdash t_1 : T_1}{\Gamma \vdash \text{ref } t_1 : \text{Ref } T_1}$$

(T-DEREF)

$$\frac{\Gamma \vdash !t_1 : T_1}{\Gamma \vdash t_1 : \text{Ref } T_1}$$

(T-ASSIGN)

$$\frac{\Gamma \vdash t_1 : \text{Ref } T_1 \quad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1 := t_2 : \text{Unit}}$$

## Another example

```
BoolArray = Ref (Nat → Bool);

newarray = λ_:Unit. ref (λn:Nat.false);
          : Unit → BoolArray

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

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

## Evaluation

---

What is the **value** of the expression `ref 0`?

---

## 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.

---

## 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.

## 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?

---

## 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.

---

## 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

---

## 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:

$v ::=$

- $\text{unit}$
- $\lambda x:T.t$
- $1$

*values*

- unit constant*
- abstraction value*
- store location*

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

# Syntax of Terms

$t ::=$	$t$
	unit
	$x$
	$\lambda x:T.t$
	$t\ t$
	ref $t$
	$!t$
	$t := t$
	<b>1</b>
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. 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.

$$t \mid \sigma \longleftarrow t' \mid \sigma'$$

We use the metavariable  $\sigma$  to range over stores.



## Evaluation

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

$$\text{(E-ASSIGN1)} \quad \frac{t_1 \mid \mu \rightarrow t'_1 \mid \mu' \quad t_1 := t_2 \mid \mu \rightarrow t'_1 := t_2 \mid \mu'}{t_1 := t_2 \mid \mu \rightarrow t'_1 := t_2 \mid \mu'}$$

$$\text{(E-ASSIGN2)} \quad \frac{t_2 \mid \mu \rightarrow t'_2 \mid \mu' \quad v_1 := t_2 \mid \mu \rightarrow v_1 := t'_2 \mid \mu'}{v_1 := t_2 \mid \mu \rightarrow v_1 := t'_2 \mid \mu'}$$

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

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

A term of the form  $\text{ref } t_1$  first evaluates inside  $t_1$  until it becomes a value...

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

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

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

A term  $it_1$  first evaluates in  $t_1$  until it becomes a value...

$$\frac{t_1 \mid u \longleftarrow t'_1 \mid u'}{it_1 \mid u \longleftarrow it'_1 \mid u'}$$

(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:

$$\frac{it_1 \mid u \longleftarrow v \mid u}{n(l) = v}$$

(E-DEREFLOC)

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

$$(E\text{-APP1}) \quad \frac{t_1 \mid u \rightarrow t'_1 \mid u'}{t_1 \ t_2 \mid u \rightarrow t'_1 \ t'_2 \mid u'}$$

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

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

---

## Aside: garbage collection

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

We can't do any!

---

Aside: pointer arithmetic

Store Typings

---

## Typing Locations

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



## Typing Locations

---

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

A: It depends on the store!

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

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

---

## Typing Locations — first try

Roughly:

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

## Typing Locations — first try

Roughly:

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

More precisely:

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

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

$$(\mathbf{n} = \mathbf{l}_1 \mapsto \lambda x:\text{Nat}. 999,$$

$$\mathbf{l}_2 \mapsto \lambda x:\text{Nat}. \mathbf{l}_1 ( \mathbf{l}_1 x ),$$

$$\mathbf{l}_3 \mapsto \lambda x:\text{Nat}. \mathbf{l}_2 ( \mathbf{l}_2 x ),$$

$$\mathbf{l}_4 \mapsto \lambda x:\text{Nat}. \mathbf{l}_3 ( \mathbf{l}_3 x ),$$

$$\mathbf{l}_5 \mapsto \lambda x:\text{Nat}. \mathbf{l}_4 ( \mathbf{l}_4 x ),$$

then how big is the typing derivation for  $\mathbf{l}_5$ ?

---

## Problem!

But wait... it gets worse. Suppose

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

Now 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.

E.g., for

$$\mu = (l_1 \mapsto \lambda x:\text{Nat}. 999, \\ l_2 \mapsto \lambda x:\text{Nat}. i l_1 (i l_1 x), \\ l_3 \mapsto \lambda x:\text{Nat}. i l_2 (i l_2 x), \\ l_4 \mapsto \lambda x:\text{Nat}. i l_3 (i l_3 x), \\ l_5 \mapsto \lambda x:\text{Nat}. i l_4 (i l_4 x)),$$

A reasonable store typing would be

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

Now, suppose we are given a store typing  $\Sigma$  describing the store in which we intend to evaluate some term  $t$ . Then we can use  $\Sigma$  to look up the types of locations in  $t$  instead of calculating them from the values in  $\mathfrak{n}$ .

$$\frac{\Gamma \mid \Sigma \vdash 1 : \text{Ref } T_1}{\Sigma(1) = T_1} \text{ (T-Loc)}$$

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



## Final typing rules

(T-LOC)

$$\frac{\Sigma(1) = T_1}{\Gamma \mid \Sigma \vdash 1 : \text{Ref } T_1}$$

(T-REF)

$$\frac{\Gamma \mid \Sigma \vdash t_1 : T_1}{\Gamma \mid \Sigma \vdash \text{ref } t_1 : \text{Ref } T_1}$$

(T-DEREF)

$$\frac{\Gamma \mid \Sigma \vdash !t_1 : T_1}{\Gamma \mid \Sigma \vdash t_1 : \text{Ref } T_1}$$

(T-ASSIGN)

$$\frac{\Gamma \mid \Sigma \vdash t_1 : \text{Ref } T_1 \quad \Gamma \mid \Sigma \vdash t_2 : T_1}{\Gamma \mid \Sigma \vdash t_1 := t_2 : \text{Unit}}$$

Q: Where do these store typings come from?

Q: Where do these store typings come from?

A: 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,

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

we can observe the type of  $v_1$  and extend the “current store typing” appropriately.

[on board]

---

Safety