

# Lecture 12: Memory and References

Polyvios Pratikakis

Computer Science Department, University of Crete

Type Systems and Programming Languages



# So far

- Pure lambda calculus
- Simply typed lambda calculus
- Additional types: sums, products, lists, tuples, variants, etc.
- *Pure* language features:
  - ▶ The machine state is a program expression
  - ▶ The semantics rewrite the program expression/machine state
  - ▶ Program evaluation reduces the program expression to a result
- Pure features form the backbone of most languages



# Impure features

- *Impure* languages
  - ▶ The machine state is not just the program expression
  - ▶ Program evaluation does not just produce a result,
    - ▶ ...it also changes the machine state
- Most languages also include impure features
  - ▶ Mutable state: memory locations, arrays, mutable record fields, etc.
  - ▶ I/O: network, display, etc.
  - ▶ Exceptions, signals, interrupts
  - ▶ Inter-process communication
  - ▶ ...
- Computation has “side-effects”: *computational effects*



# Memory effects

- Support for *assignment*, a way to alter memory contents
- Variable names remain immutable
  - ▶ In C, a variable name can mean two things
    - ★ At the left side of an assignment: a memory location
    - ★ At the right side of an assignment: the contents of a memory location
  - ▶ Keep variables immutable: a variable name always means the same
  - ▶ Use explicit syntax to read from or write to a memory location



# Memory operations

- Memory allocation (and initialization):

$\text{let } r = \text{ref } 5$

- Memory dereference (read)

$!r$

- Memory assignment (write)

$r := 42$



# Aliasing

- A reference points to a memory location
- We can copy the reference:

let  $s = r$

- That does not copy the memory location
  - ▶ Both  $s$  and  $r$  point to the same original location
  - ▶ If we assign  $s := 2$
  - ▶ Then  $!r$  will also be 2
  - ▶ We say references  $s$  and  $r$  are *aliases* for the same memory location
- Is the program  $(r := 1; r := !s)$  equivalent to the program  $(r := !s)$ ?



## Shared state

- A reference is like a communication channel
- Implicitly sends something from one part of the program to another, e.g.:

```
let c = ref 0
let incc = λx: Unit. (c := succ (!c); !c)
let decd = λx: Unit. (c := pred (!c); !c)
```

- Create sequential numbers from anywhere in the program by calling *incc*()
- The function *incc* is *stateful*: we don't need to give it the previous value, *incc* remembers it (and so is *decd*)
- Reference *c* works like an implicit argument to *incc* and *decd*, contains the last thing stored



## Shared state (cont'd)

- We can pack it all in a record

```
let counter =  
  let c = ref 0 in  
  {  
    incr =  $\lambda x : \text{Unit}. (c := \text{succ } (!c); !c)$ ,  
    decr =  $\lambda x : \text{Unit}. (c := \text{pred } (!c); !c)$   
  }
```

- We can now use *counter.incr()* and *counter.decr()*
- This is a simple *object*





# References, formally

- Syntax

$$e ::= \dots \mid \text{ref } e \mid !e \mid e := e$$
$$T ::= \dots \mid \text{Ref } T$$

- Typing

$$[\text{T-REF}] \frac{\Gamma \vdash e : T}{\Gamma \vdash \text{ref } e : \text{Ref } T}$$

$$[\text{T-DEREF}] \frac{\Gamma \vdash e : \text{Ref } T}{\Gamma \vdash !e : T}$$

$$[\text{T-ASSIGN}] \frac{\Gamma \vdash e_1 : \text{Ref } T \quad \Gamma \vdash e_2 : T}{\Gamma \vdash e_1 := e_2 : \text{Unit}}$$



## References, formally (cont'd)

- What is the result of ref 2 at run time?
  - ▶ Allocates a new memory location,
  - ▶ initializes it with 2, and
  - ▶ returns a pointer to that location
  - ▶ But what is the value of the pointer?
- We add another type of value (and expression) that only occurs at run-time:

$$v, e ::= \dots \mid l$$

- A pointer, or location,  $l$  is an element of an abstract set of all possible locations  $\mathcal{L}$
- We represent memory as a partial function from locations  $l$  to values



## References, formally (cont'd)

- Extend operational semantics with memory
- The machine state is not just an expression  $e$  like in pure calculus
- New machine state is  $\langle M \mid e \rangle$
- $M$  represents memory: a map from locations  $l$  to values (also called *store*)
- Operational semantics define transitions between the new machine states:
  - ▶ Small-step:  $\langle M \mid e \rangle \rightarrow \langle M' \mid e' \rangle$
  - ▶ Big-step:  $\langle M \mid e \rangle \Downarrow \langle M' \mid v \rangle$



- We need to extend all existing semantic rules with memory

$$\frac{}{\langle M \mid (\lambda x : T.e) \ v \rangle \rightarrow \langle M \mid e[v/x] \rangle}$$

$$\langle M \mid e_1 \rangle \rightarrow \langle M' \mid e'_1 \rangle$$

$$\frac{}{\langle M \mid e_1 \ e_2 \rangle \rightarrow \langle M' \mid e'_1 \ e_2 \rangle}$$

$$\langle M \mid e \rangle \rightarrow \langle M' \mid e' \rangle$$

$$\frac{}{\langle M \mid v \ e \rangle \rightarrow \langle M' \mid v \ e' \rangle}$$



# Semantics (cont'd)

- Allocation

$$\frac{\langle M \mid e \rangle \rightarrow \langle M' \mid e' \rangle}{\langle M \mid \text{ref } e \rangle \rightarrow \langle M' \mid \text{ref } e' \rangle}$$

$$\frac{l \notin \text{dom}(M)}{\langle M \mid \text{ref } v \rangle \rightarrow \langle (M, l \mapsto v) \mid l \rangle}$$

- Dereference

$$\frac{\langle M \mid e \rangle \rightarrow \langle M' \mid e' \rangle}{\langle M \mid !e \rangle \rightarrow \langle M' \mid !e' \rangle} \qquad \frac{M(l) = v}{\langle M \mid !l \rangle \rightarrow \langle M \mid v \rangle}$$



# Semantics (cont'd)

- Assignment

$$\frac{\langle M \mid e_1 \rangle \rightarrow \langle M' \mid e'_1 \rangle}{\langle M \mid e_1 := e_2 \rangle \rightarrow \langle M' \mid e'_1 := e_2 \rangle}$$

$$\frac{\langle M \mid e \rangle \rightarrow \langle M' \mid e' \rangle}{\langle M \mid v := e \rangle \rightarrow \langle M' \mid v := e' \rangle}$$

$$\frac{}{\langle M \mid l := v \rangle \rightarrow \langle M[l \mapsto v] \mid () \rangle}$$



# Store typing

- To prove type soundness, we need (as before) progress and preservation
- But, the run-time language includes locations  $l$
- What is the type of a location?
  - ▶ It depends on the value it points to in the store (incorrect):

$$\frac{\Gamma \vdash M(l) : T}{\Gamma \vdash l : Ref\ T}$$

- The store becomes part of the typing relation:  $\Gamma; M \vdash e : T$
- Typing locations (not yet correctly):

$$\frac{\Gamma; M \vdash M(l) : T}{\Gamma; M \vdash l : Ref\ T}$$



## Store typing (cont'd)

- What happens when the store has a cycle?
  - ▶ Typing doesn't terminate: bad!
- Instead, use *store typing*  $\Sigma$ , a map from locations to types
- Now, typing relation depends on  $\Sigma$ :  $\Gamma; \Sigma \vdash e : T$
- Typing locations (correctly):

$$[\text{T-LOC}] \frac{\Sigma(l) = T}{\Gamma; \Sigma \vdash l : \text{Ref } T}$$

- The other rules are simple to extend: just pass  $\Sigma$  up recursively
- To type original program, use empty  $\Sigma$ : no pointers allowed in the original program text





# Typing, finally

$$[\text{T-ABS}] \frac{\Gamma, x : T; \Sigma \vdash e : T'}{\Gamma; \Sigma \vdash (\lambda x : T. e) : T \rightarrow T'}$$

$$[\text{T-VAR}] \frac{x : T \in \Gamma}{\Gamma; \Sigma \vdash x : T}$$

$$[\text{T-APP}] \frac{\Gamma; \Sigma \vdash e_1 : T \rightarrow T' \quad \Gamma; \Sigma \vdash e_2 : T}{\Gamma; \Sigma \vdash e_1 e_2 : T'}$$

$$[\text{T-UNIT}] \frac{}{\Gamma; \Sigma \vdash () : \text{Unit}}$$

$$[\text{T-REF}] \frac{\Gamma; \Sigma \vdash e : T}{\Gamma; \Sigma \vdash \text{ref } e : \text{Ref } T}$$

$$[\text{T-DEREF}] \frac{\Gamma; \Sigma \vdash e : \text{Ref } T}{\Gamma; \Sigma \vdash !e : T}$$

$$[\text{T-ASSIGN}] \frac{\Gamma; \Sigma \vdash e_1 : \text{Ref } T \quad \Gamma; \Sigma \vdash e_2 : T}{\Gamma; \Sigma \vdash e_1 := e_2 : \text{Unit}}$$

$$[\text{T-LOC}] \frac{\Sigma(l) = T}{\Gamma; \Sigma \vdash l : \text{Ref } T}$$

...



# Store typing, finally

- To state and prove soundness (progress and preservation) we need to link  $M$  and  $\Sigma$ :
  - ▶ A store  $M$  is *well-typed* in context  $\Gamma$  under store typing  $\Sigma$ , written  $\Gamma; \Sigma \vdash M$ , if
    - ★  $dom(M) = dom(\Sigma)$  and
    - ★  $\Gamma; \Sigma \vdash M(l) : \Sigma(l)$  for all  $l \in dom(M)$



# Preservation theorem

- If a well-typed program takes a step, it is still well-typed:  
If

- ▶  $\Gamma; \Sigma \vdash e : T$ ,
- ▶  $\Gamma; \Sigma \vdash M$  and
- ▶  $\langle M \mid e \rangle \rightarrow \langle M' \mid e' \rangle$

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

- ▶  $\Gamma; \Sigma' \vdash e' : T$  and
  - ▶  $\Gamma; \Sigma' \vdash M'$
- We prove as before by induction on the evaluation derivation.
  - But first, we need a few auxiliary lemmas



## Preservation theorem (cont'd)

- Prove the substitution lemma:  
If  $\Gamma, x : T; \Sigma \vdash e : T'$  and  $\Gamma; \Sigma \vdash v : T$  then  $\Gamma; \Sigma \vdash e[v/x] : T'$ .
- Prove we can update values in the store (keeping the same type):  
If  $\Gamma; \Sigma \vdash M$ ,  $\Sigma(l) = T$  and  $\Gamma; \Sigma \vdash v : T$ , then  $\Gamma; \Sigma \vdash M[l \mapsto v]$
- Prove weakening for stores, we can always add stuff to the store:  
If  $\Gamma; \Sigma \vdash e : T$  and  $\Sigma' \supseteq \Sigma$ , then  $\Gamma; \Sigma' \vdash e : T$ .



# Progress theorem

- A closed, well-typed program is either a value, or it can take a step:  
If  $\emptyset, \Sigma \vdash e : T$ , then either  $e$  is a value, or for any store  $M$  for which  $\emptyset; \Sigma \vdash M$ , there are some  $e'$  and  $M'$  such that  $\langle M \mid e \rangle \rightarrow \langle M' \mid e' \rangle$ .
- Proof as before, by induction on typing derivations
- Need to extend the canonical forms lemma with the cases for *Unit* and *Ref T*

