Lecture 16: Mutable State

Steven Holtzen
s.holtzen@northeastern.edu

CS4400/5400 Fall 2024

1 Programming with Mutable State

- We've seen a few examples of programming with mutable state in Racket using `box`, `unbox`, and `set-box!`:

```
> (define my-box (box 10))
> (unbox my-box)
10
> (set-box! my-box 20)
> (unbox my-box)
20
```

- OCaml also supports references, using different syntax:

```
# let r = ref 10;;   (* boxes are declared using the ref syntax *)
val r : int ref = {contents = 10}
# !r;;               (* unbox is written !r *)
- : int = 10
# r := 25;;          (* set-box! is written with := *)
- : unit = ()
# !r;;
- : int = 25
```

- So far in class, all the languages we have implemented have had **immutable values**: once a value is bound to an identifier, it can never change.
- These programs exhibit **mutable values**: it is possible for a value to change after it is defined.
- In this class we'll design a semantics and type system for languages with mutable state.

2 Some Consequences of Mutation

- Mutation is an **effect**. Effects are things a program does besides returning values. Examples of effects include: mutable updates; printing things to your display; writing things to a file; and other ways of interacting with the environment.

- So far in class all of our programs have been **pure**, meaning that they are free from effects. This means that our programs have been entirely characterized by their input and output behavior; this property is often called **referential transparency**.

- A pure function always has the same behavior when it is called with the same arguments.[1] In general, this is a nice property to have. But, mutation clearly permits writing programs that do not satisfy this property, for example, a simple counter function:

```
> (define my-counter (box 0))
> (define (fresh)
    (set-box! my-counter (+ (unbox my-counter) 1))
    (unbox my-counter))
 > (fresh)
 1
 > (fresh)
 2
 > (fresh)
 3
```

  The `fresh` function is not referentially transparent because it yields different outputs when provided with the same input.

- Purity is a powerful property because it enables many kinds of compiler optimizations and makes it easy for programmers to understand what a function does. For example, if we determine that a pure function is only ever called with a single argument, then the compiler can exploit this fact to generate more efficient code.

- It is also easier to unit test pure functions: when testing functions with effects, one must also consider the environment (or *context*) in which the program is being run.

[1] "The definition of insanity is doing the same thing over and over again and expecting different results"

3 MutLang: A Small Language with Mutable State

- Let's consider at first small untyped surface language for mutable state with the following surface syntax:

```
e ::= (let <ident> <e> <e>)
    | <ident>
    | (box <e>)
    | (unbox <ident>)
    | (set-box! <ident> <e>)
    | <number>
    | <unit>
    | (add <e> <e>)
```

Figure 1: Surface syntax of untyped MutLang.

- The goal is for MutLang to behave like a small subset of Racket that supports boxes.
- Semantics in English:
  - `(box e)` does the following: (1) it allocates a new cell on the **heap** at location ℓ (which maps addresses to values), (2) stores the result of running `e` in the heap at location ℓ, (3) returns the location ℓ.
  - `(unbox x)` does the following: (1) get location ℓ that the identifier `x` corresponds with, (2) returns the value in the heap at location ℓ.
  - `(set-box! x e)` does the following: (1) get location ℓ that identifier `x` corresponds with, (2) sets heap at location ℓ equal to the result of running `e`, (3) returns the number `0`.
- Let's run some MutLang programs to see how they work. We will again rely on our stepper arrow $\to$ to visualize simplifying the program step by step. Our step will have the form $\rho, e \to \rho', e'$ where ρ is an environment.
- In the following stepping judgments, we will use OCaml-like surface syntax and permit ourselves the use of syntactic sugar like `let` to make things easier to follow.

```
  {}, let x = box 10 in unbox x
-- evaluate box 10 -->
  {0x0 ↦ 10}, let x = 0x0 in unbox x
-- substitute x -->
  {0x0 ↦ 10}, unbox 0x0
-- evaluate unbox -->
  {0x0 ↦ 10}, 10
```

- Let's do an example involving `set-box!`:

```
  {}, let x = box 10 in (let z = set-box! x 20 in (unbox x))
-- evaluate box -->
  {0x0 ↦ 10}, let x = 0x0 in (let z = set-box! x 20 in (unbox x))
-- substitute x -->
  {0x0 ↦ 10}, let z = set-box! 0x0 20 in (unbox 0x0)
-- evaluate set-box! -->
  {0x0 ↦ 20}, let z = 0 in (unbox 0x0)
-- substitute z -->
  {0x0 ↦ 20}, unbox 0x0
-- evaluate unbox -->
  20
```

- *Notice*: it is not possible for us to generate a location without using box. This prevents a large class of errors, such as dereferencing unallocated memory cells or stack overflows.

4 A MutLang Interpreter

```
;;; result of running the interpreter
(struct interp-res (v heap) #:transparent)

;;; interp : heap -> enviroment -> lexpr -> value * heap
;;; runs a lambda term and produces a value and a new heap
(define (interp heap env e)
  (match e
    [(eident x)
     (interp-res (hash-ref env x) heap)]
    [(enum n) (interp-res (vnum n) heap)]
    [(elet id assgn body)
     (define interp-assgn (interp heap env assgn))
     (define extend-env (hash-set env id (interp-res-v interp-assgn)))
     (interp (interp-res-heap interp-assgn) extend-env body)]
    [(eadd e1 e2)
     (define interp-e1 (interp heap env e1))
     (define interp-e2 (interp (interp-res-heap interp-e1) env e2))
     (define res-v (vnum (+ (vnum-n (interp-res-v (interp-e1)))
                            (vnum-n (interp-res-v (interp-e2))))))
     (interp-res res-v (interp-res-heap interp-e2))]
    [(ebox e1)
     (define loc (fresh-loc))
     (define interp-e1 (interp heap env e1))
     (define new-heap (hash-set (interp-res-heap interp-e1)
                                loc
                                (interp-res-v interp-e1)))
     (interp-res (vloc loc) new-heap)]
    [(eunbox e)
     (define interp-e (interp heap env e))
     (define new-heap (interp-res-heap interp-e))
     (match (interp-res-v interp-e)
       [(vloc l)
        ; dereference the location and return it
        (interp-res (hash-ref new-heap l) new-heap)]
       [_ (error 'illegal-unbox)])]
    [(eset x e)
     ; set x = e in the heap and return the (0, new_heap)
     (define loc (vloc-l (hash-ref env x)))
     (define eval-e (interp heap env e))
     (interp-res (vunit) (hash-set (interp-res-heap eval-e) loc (interp-res-v eval-e)))
     ]))
```

5 Designing a Type System for MutLang

- What are the ways in which MutLang programs can go wrong? Here are some examples that go beyond the typical ones we've seen for the simply-typed lambda calculus:

  1. Unboxing a non-ref: `(unbox 10)`
  2. Treating an unboxed value as the wrong type: `let x = ref true in !x + 20`
  3. This one is very subtle: *changing the type* of a value on the heap. This is called a **strong update**. For example, in OCaml, the following program yields a runtime error:

     ```
     let x = ref 10 in
     x := true
     !x + 20
     ```

- Strong updates are tricky because sometimes they do not lead to runtime errors. For example, the following program performs a strong update that does not cause a runtime error:

  ```
  let x = ref 10 in
  x := true
  !x
  ```

- We are once again in a soundness–expressivity tradeoff. We will choose to *forbid strong updates* in our type system, because determining the type of a value at runtime can be very computationally expensive. This is the choice OCaml makes.

- So, we need a type system that prevents the above runtime errors. This necessitates the use of a *reference type* `Ref t`, which denotes a reference to a value of type `t`. Our types are then:

  ```
  t ::= Ref t
      | TNum
      | TBool
      | TFun t t
  ```

$$\frac{\Gamma \vdash \texttt{e} : t}{\Gamma \vdash \texttt{(box e)} : \texttt{Ref}\ t}\ \text{(T-Box)} \qquad \frac{\Gamma \vdash \texttt{e} : \texttt{Ref}\ t}{\Gamma \vdash \texttt{(unbox e)} : t}\ \text{(T-Unbox)}$$

$$\frac{\Gamma(x) = t \qquad \Gamma \vdash e : t}{\Gamma \vdash \texttt{(set-box! x e)} : \texttt{num}}\ \text{(T-Setbox)}$$

Figure 2: Typing rules for mutable references.

6 Consequences of Mutable State

- Quite surprisingly, the simply-typed lambda calculus augmented with mutable state is sufficiently expressive to write recursive programs!
- Consider the following OCaml program that is (1) well-typed, (2) does not make use of recursion features like letrec, and (3) runs forever:

```
let x = box (fun x -> x + 1) in
let myfun = fun y -> (!x) 10 in
x := myfun;
myfun 20
```

Figure 3: Landin's knot: a program that runs forever that involves mutable state.

- Let's run this program by hand to see why it runs forever:

```
{}, let x = box (fun x -> x + 1) in
    let myfun = fun y -> (!x) 10 in
    x := myfun;
    myfun 20
-- evaluate box -->
  {0x0 ↦ (fun x -> x + 1)},
    let x = 0x0 in
    let myfun = fun y -> (!x) 10 in
    x := myfun;
    myfun 20
-- substitute x -->
  {0x0 ↦ (fun x -> x + 1)},
    let myfun = fun y -> (!0x0) 10 in
    0x0 := myfun;
    myfun 20
-- substitute myfun -->
  {0x0 ↦ (fun x -> x + 1)},
    0x0 := fun y -> (!0x0) 10;
    (fun y -> (!0x0) 10) 20
-- update heap -->
  {0x0 ↦ (fun y -> (!0x0) 10)},
    (fun y -> (!0x0) 10) 20
-- substitute 20 -->
  {0x0 ↦ (fun y -> (!0x0) 10)},
  (!0x0) 10
-- lookup 0x0 -->
  {0x0 ↦ (fun y -> (!0x0) 10)},
  (fun y -> (!0x0) 10) 10
...
```

7 Type Conclusions

- Key concepts we've covered in this module:
  - The concept of types and type safety
  - How to design a typechecker to prevent runtime errors
  - The simply-typed lambda calculus (STLC) and its properties
  - Extensions of STLC: sum types, product types, mutable references
  - Beyond simple types: System F and polymorphism
  - Runtime safety and an abstract x86 machine
- Some interesting topics in types we did not have time to cover, almost too many to list: 1. Existential types and modules 2. Dependent types 3. Recursive types 4. Higher-kinded types 5. Subtyping 6. Gradual typing 7. Objects and object-oriented programming
- If you want to learn more:
  - Read Pierce [2002], which you should now be prepared to read.
  - Check out this webpage: `https://counterexamples.org/intro.html`
  - Explore Software Foundations, which is an introduction to using the Coq proof assistant: `https://softwarefoundations.cis.upenn.edu/`
- Types are increasingly influencing modern language design:
  - Web assembly (WASM) has types in its specification: `https://webassembly.github.io/spec/core/syntax/types.html`
  - Types are making their way into many languages that we use today and are growing in popularity: Typescript, Python, Rust

References

Benjamin C Pierce. *Types and programming languages*. MIT press, 2002.