Mutable Variables

CS 350

Dr. Joseph Eremondi

Last updated: August 1, 2024

Curly-Mutvar

Overview

- Learning goals

Overview

- Learning goals
  - To interpret a language where variables can change values (mutate)

Overview

- Learning goals
  - To interpret a language where variables can change values (mutate)
  - To understand the design choices around functions in such a language

Overview

- Learning goals
  - To interpret a language where variables can change values (mutate)
  - To understand the design choices around functions in such a language
- Key concepts

Overview

- Learning goals
  - To interpret a language where variables can change values (mutate)
  - To understand the design choices around functions in such a language
- Key concepts
  - Pass-by-reference vs. Pass-by-value

Identifiers vs Variables

- Until now, a variable denoted a *value*

Identifiers vs Variables

- Until now, a variable denoted a *value*
    - In a given environment

Identifiers vs Variables

- Until now, a variable denoted a *value*
  - In a given environment
- They didn't really ever vary

Identifiers vs Variables

- Until now, a variable denoted a *value*
    - In a given environment
- They didn't really ever vary
    - Except between different function calls

Identifiers vs Variables

- Until now, a variable denoted a *value*
  - In a given environment
- They didn't really ever vary
  - Except between different function calls
- To allow variables values to change, we can make one simple change:

Identifiers vs Variables

- Until now, a variable denoted a *value*
  - In a given environment
- They didn't really ever vary
  - Except between different function calls
- To allow variables values to change, we can make one simple change:
  - **Keep locations instead of values in the environment**

Identifiers vs Variables

- Until now, a variable denoted a *value*
    - In a given environment
- They didn't really ever vary
    - Except between different function calls
- To allow variables values to change, we can make one simple change:
    - **Keep locations instead of values in the environment**
    - Then each variable refers to a single store location, whose value can change

Bindings with mutable variables

- Each binding associates a symbol with a *location*

Bindings with mutable variables

- Each binding associates a symbol with a *location*

Bindings with mutable variables

- Each binding associates a symbol with a *location*

```
(define-type Binding
  (bind [name : Symbol]
        [loc : Location]))
```

- All other environment operations are the same

Bindings with mutable variables

- Each binding associates a symbol with a *location*

```
(define-type Binding
  (bind [name : Symbol]
        [loc : Location]))
```

- All other environment operations are the same
- Lookup now has type:

Bindings with mutable variables

- Each binding associates a symbol with a *location*

```
(define-type Binding
  (bind [name : Symbol]
        [loc : Location]))
```

- All other environment operations are the same
- Lookup now has type:

Bindings with mutable variables

- Each binding associates a symbol with a *location*

```
(define-type Binding
  (bind [name : Symbol]
        [loc : Location]))
```

- All other environment operations are the same
- Lookup now has type:

```
(define (lookup [n : Symbol] [env : Env]) : Location ....)
```

Interpreting Variables

Interpreting Variables

```
(define (interp [env : Env]
                [e : Expr]
                [sto : Store]) : Result
  (type-case Expr e
    [(Var x)
     (v*s (fetch (lookup x env) sto) sto)]
....))
```

- Previously, we just looked up a value with `lookup`

Interpreting Variables

```
(define (interp [env : Env]
                [e : Expr]
                [sto : Store]) : Result
  (type-case Expr e
    [(Var x)
     (v*s (fetch (lookup x env) sto) sto)]
....))
```

- Previously, we just looked up a value with `lookup`
- Now we lookup a *location*

Interpreting Variables

```
(define (interp [env : Env]
                [e : Expr]
                [sto : Store]) : Result
  (type-case Expr e
    [(Var x)
     (v*s (fetch (lookup x env) sto) sto)]
....))
```

- Previously, we just looked up a value with `lookup`
- Now we lookup a *location*
  - Have to use `fetch` to get its value from the store

Interpreting Variables

```
(define (interp [env : Env]
                [e : Expr]
                [sto : Store]) : Result
  (type-case Expr e
    [(Var x)
     (v*s (fetch (lookup x env) sto) sto)]
....))
```

- Previously, we just looked up a value with `lookup`
- Now we lookup a *location*
  - Have to use `fetch` to get its value from the store
- Produce that value, along with the unchanged store

Adding mutation: Curly-Mutvar

- Curly syntax: `{setvar!  SYMBOL <expr>}`

Adding mutation: Curly-Mutvar

- Curly syntax: `{setvar!  SYMBOL <expr>}`
    - `{setvar!  x e}` changes the value of in-scope variable `x` to be the value of `e`

Adding mutation: Curly-Mutvar

- Curly syntax: `{setvar!  SYMBOL <expr>}`
    - `{setvar!  x e}` changes the value of in-scope variable `x` to be the value of `e`
- Interpreting

Adding mutation: Curly-Mutvar

- Curly syntax: `{setvar!  SYMBOL <expr>}`
    - `{setvar!  x e}` changes the value of in-scope variable `x` to be the value of `e`
- Interpreting
    - Just like `SetBox` except we get the location from the environment, instead of by evaluating a box

Adding mutation: Curly-Mutvar

- Curly syntax: `{setvar!  SYMBOL <expr>}`
  - `{setvar!  x e}` changes the value of in-scope variable `x` to be the value of `e`
- Interpreting
  - Just like `SetBox` except we get the location from the environment, instead of by evaluating a box

Adding mutation: Curly-Mutvar

- Curly syntax: `{setvar!  SYMBOL <expr>}`
  - `{setvar!  x e}` changes the value of in-scope variable x to be the value of e
- Interpreting
  - Just like `SetBox` except we get the location from the environment, instead of by evaluating a box

```
(define (interp [env : Env]
                [e : Expr]
                [sto : Store]) : Result
(type-case Expr e
  [(Setvar! x e)
   (with ([e-val e-sto] (interp env e sto))
    (v*s e-val
      ;; Get the location from the environment
      (override-store (cell (lookup x env) e-val)
                      e-sto)))]
....))
```

Function Calls

- To call a function, we evaluate the body in the environment extended with the argument value

Function Calls

- To call a function, we evaluate the body in the environment extended with the argument value
  - Environment now takes *locations*

Function Calls

- To call a function, we evaluate the body in the environment extended with the argument value
  - Environment now takes *locations*
  - Need a location to associate with the new variable

Function Calls

- To call a function, we evaluate the body in the environment extended with the argument value
    - Environment now takes *locations*
    - Need a location to associate with the new variable
    - Need to make sure the argument value ends up at that location

Function Calls

- To call a function, we evaluate the body in the environment extended with the argument value
    - Environment now takes *locations*
    - Need a location to associate with the new variable
    - Need to make sure the argument value ends up at that location
- If the argument is e.g. a number, then we have to make a new location for it

Function Calls

- To call a function, we evaluate the body in the environment extended with the argument value
  - Environment now takes *locations*
  - Need a location to associate with the new variable
  - Need to make sure the argument value ends up at that location
- If the argument is e.g. a number, then we have to make a new location for it
- *What if the argument is already a variable?*

Function Calls

- To call a function, we evaluate the body in the environment extended with the argument value
    - Environment now takes *locations*
    - Need a location to associate with the new variable
    - Need to make sure the argument value ends up at that location
- If the argument is e.g. a number, then we have to make a new location for it
- *What if the argument is already a variable?*
    - Have a design decision

Pass-by-value

- If we implement function calls using pass-by-value, then:

Pass-by-value

- If we implement function calls using pass-by-value, then:
  - **Each function call generates a new location where its argument values are stored**

Pass-by-value

- If we implement function calls using pass-by-value, then:
  - **Each function call generates a new location where its argument values are stored**
  - If the arguments are variables, their values are looked up and copied to the new location

Pass-by-value interp

Pass-by-value interp

```
[(Call funExpr argExpr)
     (with ([fun-v fun-sto] (interp env funExpr sto))
        (with ([arg-v arg-sto] (interp env argExpr fun-sto))
          (let* (
            [funPair (checkAndGetClosure fun-v)] ;; Function might be
            [argVar (fst (fst funPair))]
            [funBody (snd (fst funPair))]
            [funEnv (snd funPair)]
            ;; Allocate a new location for the argument value
            [argLoc (new-location arg-sto)]
            ;; new store with the arg value at the new location
            ;; Use most recent store from arg
            [body-sto (override-store (cell argLoc arg-v) arg-sto)])
           ;; Evaluate the body in the extended *closure* env
           ;; with the new location bound to the parameter name,
           ;; using the new store with the argument value
          (interp (extendEnv (bind argVar argLoc) funEnv)
                  funBody
                  body-sto))))]
```

Pass-by-reference

- Pass-by-reference means that, when a function argument is a variable, the function's body is evaluated in an environment *where the argument variable is bound to the input variable's location*

Pass-by-reference

- Pass-by-reference means that, when a function argument is a variable, the function's body is evaluated in an environment *where the argument variable is bound to the input variable's location*
- Changes to one will be seen in the other

Interpreting

Interpreting

```
;; Everything the same as pass-by-value
;; except we check if the argument is a variable
;; and use its location instead of the new one if it us
(type-case Expr argExpr
  [(Var x)
     (interp (extendEnv (bind argVar (lookup x env)) funEnv)
                  funBody
                  arg-sto)))]
  ;; Otherwise do the same as call-by-value
  [else ....])
```

The Difference

- Any changes made to the parameter variable are lost in pass-by-value, but kept in pass-by-reference

The Difference

- Any changes made to the parameter variable are lost in pass-by-value, but kept in pass-by-reference
- Pass by value means that a function can only mutate locations that it is *explicitly given*

The Difference

- Any changes made to the parameter variable are lost in pass-by-value, but kept in pass-by-reference
- Pass by value means that a function can only mutate locations that it is *explicitly given*
  - i.e. as box parameters

The Difference

- Any changes made to the parameter variable are lost in pass-by-value, but kept in pass-by-reference
- Pass by value means that a function can only mutate locations that it is *explicitly given*
  - i.e. as box parameters
- Pass by reference allows you to abstract over patterns of mutation

The Difference

- Any changes made to the parameter variable are lost in pass-by-value, but kept in pass-by-reference
- Pass by value means that a function can only mutate locations that it is *explicitly given*
    - i.e. as box parameters
- Pass by reference allows you to abstract over patterns of mutation
    - e.g. Write a function that says “change these variables in this way” that can be used over and over again

The Difference

- Any changes made to the parameter variable are lost in pass-by-value, but kept in pass-by-reference
- Pass by value means that a function can only mutate locations that it is *explicitly given*
  - i.e. as box parameters
- Pass by reference allows you to abstract over patterns of mutation
  - e.g. Write a function that says "change these variables in this way" that can be used over and over again
  - e.g. swap two variable's values

Example

Example

```
{letvar y 2
  {letvar f {fun {x} {begin
                       {setvar! x {* x 3}}
                       x}}
        {+ {f y} y}}}
```

- In both pass-by-value and pass-by-reference, the call to `{f y}` produces 6

Example

```
{letvar y 2
  {letvar f {fun {x} {begin
                        {setvar! x {* x 3}}
                        x}}
        {+ {f y} y}}}
```

- In both pass-by-value and pass-by-reference, the call to `{f y}` produces 6
- In pass-by-value, `x` has a different location than `y`, that started off with `2` (the value of `y`)

Example

```
{letvar y 2
  {letvar f {fun {x} {begin
                       {setvar! x {* x 3}}
                       x}}
        {+ {f y} y}}}
```

- In both pass-by-value and pass-by-reference, the call to `{f y}` produces 6
- In pass-by-value, `x` has a different location than `y`, that started off with `2` (the value of `y`)
    - The final addition adds the value of `{f y}` to the value of `y`, which did not change

Example

```
{letvar y 2
  {letvar f {fun {x} {begin
                      {setvar! x {* x 3}}
                      x}}
        {+ {f y} y}}}
```

- In both pass-by-value and pass-by-reference, the call to `{f y}` produces 6
- In pass-by-value, `x` has a different location than `y`, that started off with `2` (the value of `y`)
  - The final addition adds the value of `{f y}` to the value of `y`, which did not change
  - `{+ 6 2}`, result of 8

Example

```
{letvar y 2
  {letvar f {fun {x} {begin
                      {setvar! x {* x 3}}
                      x}}
        {+ {f y} y}}}
```

- In both pass-by-value and pass-by-reference, the call to `{f y}` produces 6
- In pass-by-value, `x` has a different location than `y`, that started off with `2` (the value of `y`)
  - The final addition adds the value of `{f y}` to the value of `y`, which did not change
  - `{+ 6 2}`, result of 8
- In pass-by-reference, `x` refers to the same store location as `y`

Example

```
{letvar y 2
  {letvar f {fun {x} {begin
                       {setvar! x {* x 3}}
                       x}}
        {+ {f y} y}}}
```

- In both pass-by-value and pass-by-reference, the call to `{f y}` produces 6
- In pass-by-value, `x` has a different location than `y`, that started off with `2` (the value of `y`)
    - The final addition adds the value of `{f y}` to the value of `y`, which did not change
    - `{+ 6 2}`, result of 8
- In pass-by-reference, `x` refers to the same store location as `y`
    - Setting the value of `x` changes `y` because they both refer to the same location

Example

```
{letvar y 2
  {letvar f {fun {x} {begin
                     {setvar! x {* x 3}}
                     x}}
        {+ {f y} y}}}
```

- In both pass-by-value and pass-by-reference, the call to `{f y}` produces 6
- In pass-by-value, `x` has a different location than `y`, that started off with `2` (the value of `y`)
  - The final addition adds the value of `{f y}` to the value of `y`, which did not change
  - `{+ 6 2}`, result of 8
- In pass-by-reference, `x` refers to the same store location as `y`
  - Setting the value of `x` changes `y` because they both refer to the same location
  - Result is 12, since both the call and `y` have the value of 6

Which to Choose?

- We will use pass-by-value, since it's simpler

Which to Choose?

- We will use pass-by-value, since it's simpler
  - C++ is pass by value, but sometimes that value is a pointer/reference

Which to Choose?

- We will use pass-by-value, since it’s simpler
  - C++ is pass by value, but sometimes that value is a pointer/reference
- However, we can replicate pass-by-reference using Boxes

Which to Choose?

- We will use pass-by-value, since it's simpler
  - C++ is pass by value, but sometimes that value is a pointer/reference
- However, we can replicate pass-by-reference using Boxes
  - This is what e.g. Python does

Which to Choose?

- We will use pass-by-value, since it's simpler
    - C++ is pass by value, but sometimes that value is a pointer/reference
- However, we can replicate pass-by-reference using Boxes
    - This is what e.g. Python does
        - Most objects are implicitly boxed, so it seems like pass by reference

Which to Choose?

- We will use pass-by-value, since it's simpler
  - C++ is pass by value, but sometimes that value is a pointer/reference
- However, we can replicate pass-by-reference using Boxes
  - This is what e.g. Python does
    - Most objects are implicitly boxed, so it seems like pass by reference
    - Actually passing a value, but the value is (something like) a box

Which to Choose?

- We will use pass-by-value, since it's simpler
  - C++ is pass by value, but sometimes that value is a pointer/reference
- However, we can replicate pass-by-reference using Boxes
  - This is what e.g. Python does
    - Most objects are implicitly boxed, so it seems like pass by reference
    - Actually passing a value, but the value is (something like) a box
    - Immutable types (like tuples) are passed by value

Aliasing

- To enable passing by reference, we add an *aliasing expression* to Curly-Mutvar

Aliasing

- To enable passing by reference, we add an *aliasing expression* to Curly-Mutvar
  - Like “address-of” operator & in C++

Aliasing

- To enable passing by reference, we add an *aliasing expression* to Curly-Mutvar
  - Like "address-of" operator & in C++
- `{getloc SYMBOL}` produces a box whose location is the same location as whatever in-scope symbol it is given

Aliasing

- To enable passing by reference, we add an *aliasing expression* to Curly-Mutvar
    - Like “address-of” operator & in C++
- `{getloc SYMBOL}` produces a box whose location is the same location as whatever in-scope symbol it is given
    - e.g. `{getloc x}` would produce `{BoxV l}` where `l` is the location in the environment for `x`

Aliasing

- To enable passing by reference, we add an *aliasing expression* to Curly-Mutvar
  - Like “address-of” operator & in C++
- `{getloc SYMBOL}` produces a box whose location is the same location as whatever in-scope symbol it is given
  - e.g. `{getloc x}` would produce `{BoxV l}` where `l` is the location in the environment for `x`
- Interpreting:

Aliasing

- To enable passing by reference, we add an *aliasing expression* to Curly-Mutvar
  - Like “address-of” operator & in C++
- `{getloc SYMBOL}` produces a box whose location is the same location as whatever in-scope symbol it is given
  - e.g. `{getloc x}` would produce `{BoxV l}` where `l` is the location in the environment for `x`
- Interpreting:
  - Like new-box, except we look up the location in the environment instead of getting a new one

Aliasing

- To enable passing by reference, we add an *aliasing expression* to Curly-Mutvar
  - Like "address-of" operator & in C++
- `{getloc SYMBOL}` produces a box whose location is the same location as whatever in-scope symbol it is given
  - e.g. `{getloc x}` would produce `{BoxV l}` where `l` is the location in the environment for `x`
- Interpreting:
  - Like new-box, except we look up the location in the environment instead of getting a new one

Aliasing

- To enable passing by reference, we add an *aliasing expression* to Curly-Mutvar
  - Like "address-of" operator & in C++
- `{getloc SYMBOL}` produces a box whose location is the same location as whatever in-scope symbol it is given
  - e.g. `{getloc x}` would produce `{BoxV l}` where `l` is the location in the environment for x
- Interpreting:
  - Like new-box, except we look up the location in the environment instead of getting a new one

```
[(GetLoc x)
  (v*s (BoxV (lookup x env))
    ;; No changes to the store
    sto)]
```

- Now the box and the variable point to the *same location*

Aliasing

- To enable passing by reference, we add an *aliasing expression* to Curly-Mutvar
  - Like “address-of” operator & in C++
- `{getloc SYMBOL}` produces a box whose location is the same location as whatever in-scope symbol it is given
  - e.g. `{getloc x}` would produce `{BoxV l}` where `l` is the location in the environment for x
- Interpreting:
  - Like new-box, except we look up the location in the environment instead of getting a new one

```
[(GetLoc x)
  (v*s (BoxV (lookup x env))
    ;; No changes to the store
    sto)]
```

- Now the box and the variable point to the *same location*
- Changes to one are seen in the other

Example

Example

```
{letvar doublebox! {fun {x} {set-box! x {* 2 {unbox x}}}}
        letvar y 3
        {begin {doublebox {getloc y}}
               y}}
```

- Function takes in a box, gets its value, doubles it, and writes it to the same location

Example

```
{letvar doublebox! {fun {x} {set-box! x {* 2 {unbox x}}}}
        letvar y 3
        {begin {doublebox {getloc y}}
               y}}
```

- Function takes in a box, gets its value, doubles it, and writes it to the same location
- The `getloc` makes a new box whose location is the same as `y`

Example

```
{letvar doublebox! {fun {x} {set-box! x {* 2 {unbox x}}}}
        letvar y 3
        {begin {doublebox {getloc y}}
               y}}
```

- Function takes in a box, gets its value, doubles it, and writes it to the same location
- The `getloc` makes a new box whose location is the same as y
- When `doublebox!` runs it alters the value at the location of its box, which was the location of y

Example

```
{letvar doublebox! {fun {x} {set-box! x {* 2 {unbox x}}}}
        letvar y 3
        {begin {doublebox {getloc y}}
               y}}
```

- Function takes in a box, gets its value, doubles it, and writes it to the same location
- The `getloc` makes a new box whose location is the same as y
- When `doublebox!` runs it alters the value at the location of its box, which was the location of y
- The final result is 6, since the value of y was changed