Store Passing and Boxes

CS 350

Dr. Joseph Eremondi

Last updated: July 31, 2024

Overview

Store Passing Interpreters

- Objectives

Store Passing Interpreters

- Objectives
  - Understand boxes as an abstraction for mutable state

Store Passing Interpreters

- Objectives
  - Understand boxes as an abstraction for mutable state
  - Implement an interpreter for a language with mutable state

Store Passing Interpreters

- Objectives
  - Understand boxes as an abstraction for mutable state
  - Implement an interpreter for a language with mutable state
    - Where the implementation language does *not* have mutable state

Store Passing Interpreters

- Objectives
  - Understand boxes as an abstraction for mutable state
  - Implement an interpreter for a language with mutable state
    - Where the implementation language does *not* have mutable state
  - See how Racket can extend its own syntax

Store Passing Interpreters

- Objectives
  - Understand boxes as an abstraction for mutable state
  - Implement an interpreter for a language with mutable state
    - Where the implementation language does *not* have mutable state
  - See how Racket can extend its own syntax
- Key Concepts

Store Passing Interpreters

- Objectives
  - Understand boxes as an abstraction for mutable state
  - Implement an interpreter for a language with mutable state
    - Where the implementation language does *not* have mutable state
  - See how Racket can extend its own syntax
- Key Concepts
  - Mutable state

Store Passing Interpreters

- Objectives
  - Understand boxes as an abstraction for mutable state
  - Implement an interpreter for a language with mutable state
    - Where the implementation language does *not* have mutable state
  - See how Racket can extend its own syntax
- Key Concepts
  - Mutable state
  - Store data structure

Store Passing Interpreters

- Objectives
  - Understand boxes as an abstraction for mutable state
  - Implement an interpreter for a language with mutable state
    - Where the implementation language does *not* have mutable state
  - See how Racket can extend its own syntax
- Key Concepts
  - Mutable state
  - Store data structure
  - Store passing

Store Passing Interpreters

- Objectives
  - Understand boxes as an abstraction for mutable state
  - Implement an interpreter for a language with mutable state
    - Where the implementation language does *not* have mutable state
  - See how Racket can extend its own syntax
- Key Concepts
  - Mutable state
  - Store data structure
  - Store passing
  - Boxes

Store Passing Interpreters

- Objectives
  - Understand boxes as an abstraction for mutable state
  - Implement an interpreter for a language with mutable state
    - Where the implementation language does *not* have mutable state
  - See how Racket can extend its own syntax
- Key Concepts
  - Mutable state
  - Store data structure
  - Store passing
  - Boxes
  - Macros

Curly-Box: A Language with Mutation

Boxes

Expressions vs. Statements

- **Expression:** a program or program fragment that is meant to be evaluated

Expressions vs. Statements

- **Expression:** a program or program fragment that is meant to be evaluated
  - Produces a value

Expressions vs. Statements

- **Expression:** a program or program fragment that is meant to be evaluated
  - Produces a value
  - e.g. Function arguments in C++/Python are expressions

Expressions vs. Statements

- **Expression:** a program or program fragment that is meant to be evaluated
  - Produces a value
  - e.g. Function arguments in C++/Python are expressions
  - Might have side-effects

Expressions vs. Statements

- **Expression:** a program or program fragment that is meant to be evaluated
  - Produces a value
  - e.g. Function arguments in C++/Python are expressions
  - Might have side-effects
- **Statement**: a program fragment that is meant to be executed

Expressions vs. Statements

- **Expression:** a program or program fragment that is meant to be evaluated
  - Produces a value
  - e.g. Function arguments in C++/Python are expressions
  - Might have side-effects
- **Statement**: a program fragment that is meant to be executed
  - `if`, loop-bodies, function bodies are all statements in Python/C++

Expressions vs. Statements

- **Expression:** a program or program fragment that is meant to be evaluated
  - Produces a value
  - e.g. Function arguments in C++/Python are expressions
  - Might have side-effects
- **Statement**: a program fragment that is meant to be executed
  - `if`, loop-bodies, function bodies are all statements in Python/C++
  - e.g. Had to explicitly `return` or use assignment to produce a value from a statement

Expressions vs. Statements

- **Expression:** a program or program fragment that is meant to be evaluated
  - Produces a value
  - e.g. Function arguments in C++/Python are expressions
  - Might have side-effects
- **Statement**: a program fragment that is meant to be executed
  - `if`, loop-bodies, function bodies are all statements in Python/C++
  - e.g. Had to explicitly `return` or use assignment to produce a value from a statement
- All of our Curly languages (and Racket) are *expression oriented languages*

Expressions vs. Statements

- **Expression:** a program or program fragment that is meant to be evaluated
  - Produces a value
  - e.g. Function arguments in C++/Python are expressions
  - Might have side-effects
- **Statement**: a program fragment that is meant to be executed
  - `if`, loop-bodies, function bodies are all statements in Python/C++
  - e.g. Had to explicitly `return` or use assignment to produce a value from a statement
- All of our Curly languages (and Racket) are *expression oriented languages*
  - No notion of statement

Expressions vs. Statements

- **Expression:** a program or program fragment that is meant to be evaluated
  - Produces a value
  - e.g. Function arguments in C++/Python are expressions
  - Might have side-effects
- **Statement**: a program fragment that is meant to be executed
  - `if`, loop-bodies, function bodies are all statements in Python/C++
  - e.g. Had to explicitly `return` or use assignment to produce a value from a statement
- All of our Curly languages (and Racket) are *expression oriented languages*
  - No notion of statement
  - Syntax consists of nested expressions

Expressions vs. Statements

- **Expression:** a program or program fragment that is meant to be evaluated
  - Produces a value
  - e.g. Function arguments in C++/Python are expressions
  - Might have side-effects
- **Statement**: a program fragment that is meant to be executed
  - `if`, loop-bodies, function bodies are all statements in Python/C++
  - e.g. Had to explicitly `return` or use assignment to produce a value from a statement
- All of our Curly languages (and Racket) are *expression oriented languages*
  - No notion of statement
  - Syntax consists of nested expressions
    - If0 branches: expressions

Expressions vs. Statements

- **Expression:** a program or program fragment that is meant to be evaluated
  - Produces a value
  - e.g. Function arguments in C++/Python are expressions
  - Might have side-effects
- **Statement**: a program fragment that is meant to be executed
  - `if`, loop-bodies, function bodies are all statements in Python/C++
  - e.g. Had to explicitly `return` or use assignment to produce a value from a statement
- All of our Curly languages (and Racket) are *expression oriented languages*
  - No notion of statement
  - Syntax consists of nested expressions
    - If0 branches: expressions
    - Function bodies: expressions

Sequencing

- Expression-oriented version of a statement

Sequencing

- Expression-oriented version of a statement
- `{begin <expr> <expr>}`

Sequencing

- Expression-oriented version of a statement
- `{begin <expr> <expr>}`
    - Called `seq` in the textbook

Sequencing

- Expression-oriented version of a statement
- `{begin <expr> <expr>}`
  - Called `seq` in the textbook
- Evaluates the first expression *then discards the result*

Sequencing

- Expression-oriented version of a statement
- `{begin <expr> <expr>}`
  - Called `seq` in the textbook
- Evaluates the first expression *then discards the result*
  - Only useful if the first expression has side effects

Sequencing

- Expression-oriented version of a statement
- `{begin <expr> <expr>}`
  - Called `seq` in the textbook
- Evaluates the first expression *then discards the result*
  - Only useful if the first expression has side effects
- Then evaluates the second expression

Sequencing

- Expression-oriented version of a statement
- `{begin <expr> <expr>}`
  - Called `seq` in the textbook
- Evaluates the first expression *then discards the result*
  - Only useful if the first expression has side effects
- Then evaluates the second expression
- The value of the second expression is the value of the whole `begin` expression

Boxes

- Kind of like C++ References

Boxes

- Kind of like C++ References
- Pointers, but with no notion of pointer arithmetic

Boxes

- Kind of like C++ References
- Pointers, but with no notion of pointer arithmetic
- A box denotes a location in the store (memory, etc.)

Boxes

- Kind of like C++ References
- Pointers, but with no notion of pointer arithmetic
- A box denotes a location in the store (memory, etc.)
- Copies of a box refer to the same part of memory

Boxes

- Kind of like C++ References
- Pointers, but with no notion of pointer arithmetic
- A box denotes a location in the store (memory, etc.)
- Copies of a box refer to the same part of memory
    - So changes to that part are seen by *all* copies of that box

Boxes in Curly-Box

- {box <expr>}

Boxes in Curly-Box

- `{box <expr>}`
  - Allocates a new box in the store with the

Boxes in Curly-Box

- `{box <expr>}`
  - Allocates a new box in the store with the
  - Like C++ `void* x = new int; *x = expr;`

Boxes in Curly-Box

- `{box <expr>}`
  - Allocates a new box in the store with the
  - Like C++ `void* x = new int; *x = expr;`
- `{unbox <expr>}`

Boxes in Curly-Box

- `{box <expr>}`
  - Allocates a new box in the store with the
  - Like C++ `void* x = new int; *x = expr;`
- `{unbox <expr>}`
  - Evaluates the expression to a box

Boxes in Curly-Box

- `{box <expr>}`
  - Allocates a new box in the store with the
  - Like C++ `void* x = new int; *x = expr;`
- `{unbox <expr>}`
  - Evaluates the expression to a box
    - dynamic type error if not a box

Boxes in Curly-Box

- `{box <expr>}`
  - Allocates a new box in the store with the
  - Like C++ `void* x = new int; *x = expr;`
- `{unbox <expr>}`
  - Evaluates the expression to a box
    - dynamic type error if not a box
  - Produces the value for that box's location in memory

Boxes in Curly-Box

- `{box <expr>}`
  - Allocates a new box in the store with the
  - Like C++ `void* x = new int; *x = expr;`
- `{unbox <expr>}`
  - Evaluates the expression to a box
    - dynamic type error if not a box
  - Produces the value for that box's location in memory
  - Like C++ `*x;`

Boxes in Curly-Box

- `{box <expr>}`
  - Allocates a new box in the store with the
  - Like C++ `void* x = new int; *x = expr;`
- `{unbox <expr>}`
  - Evaluates the expression to a box
    - dynamic type error if not a box
  - Produces the value for that box's location in memory
  - Like C++ `*x;`
- `{set-box!  <expr> <expr>}`

Boxes in Curly-Box

- `{box <expr>}`
  - Allocates a new box in the store with the
  - Like C++ `void* x = new int; *x = expr;`
- `{unbox <expr>}`
  - Evaluates the expression to a box
    - dynamic type error if not a box
  - Produces the value for that box's location in memory
  - Like C++ `*x;`
- `{set-box!  <expr> <expr>}`
  - The `!` indicates that the form has a side effect

Boxes in Curly-Box

- `{box <expr>}`
  - Allocates a new box in the store with the
  - Like C++ `void* x = new int; *x = expr;`
- `{unbox <expr>}`
  - Evaluates the expression to a box
    - dynamic type error if not a box
  - Produces the value for that box's location in memory
  - Like C++ `*x;`
- `{set-box! <expr> <expr>}`
  - The `!` indicates that the form has a side effect
    - Just a convention, `!` is a normal character in Racket/Curly

Boxes in Curly-Box

- `{box <expr>}`
  - Allocates a new box in the store with the
  - Like C++ `void* x = new int; *x = expr;`
- `{unbox <expr>}`
  - Evaluates the expression to a box
    - dynamic type error if not a box
  - Produces the value for that box's location in memory
  - Like C++ `*x;`
- `{set-box! <expr> <expr>}`
  - The `!` indicates that the form has a side effect
    - Just a convention, `!` is a normal character in Racket/Curly
  - Evaluates the first expression to a box

Boxes in Curly-Box

- `{box <expr>}`
  - Allocates a new box in the store with the
  - Like C++ `void* x = new int; *x = expr;`
- `{unbox <expr>}`
  - Evaluates the expression to a box
    - dynamic type error if not a box
  - Produces the value for that box's location in memory
  - Like C++ `*x;`
- `{set-box! <expr> <expr>}`
  - The `!` indicates that the form has a side effect
    - Just a convention, `!` is a normal character in Racket/Curly
  - Evaluates the first expression to a box
  - Evaluates the second expression

Boxes in Curly-Box

- `{box <expr>}`
  - Allocates a new box in the store with the
  - Like C++ `void* x = new int; *x = expr;`
- `{unbox <expr>}`
  - Evaluates the expression to a box
    - dynamic type error if not a box
  - Produces the value for that box's location in memory
  - Like C++ `*x;`
- `{set-box! <expr> <expr>}`
  - The `!` indicates that the form has a side effect
    - Just a convention, `!` is a normal character in Racket/Curly
  - Evaluates the first expression to a box
  - Evaluates the second expression
  - Stores that value in the box's location in memory

Boxes in Curly-Box

- `{box <expr>}`
  - Allocates a new box in the store with the
  - Like C++ `void* x = new int; *x = expr;`
- `{unbox <expr>}`
  - Evaluates the expression to a box
    - dynamic type error if not a box
  - Produces the value for that box's location in memory
  - Like C++ `*x;`
- `{set-box! <expr> <expr>}`
  - The `!` indicates that the form has a side effect
    - Just a convention, `!` is a normal character in Racket/Curly
  - Evaluates the first expression to a box
  - Evaluates the second expression
  - Stores that value in the box's location in memory
  - Produces that value as the result

Boxes in Curly-Box

- `{box <expr>}`
  - Allocates a new box in the store with the
  - Like C++ `void* x = new int; *x = expr;`
- `{unbox <expr>}`
  - Evaluates the expression to a box
    - dynamic type error if not a box
  - Produces the value for that box's location in memory
  - Like C++ `*x;`
- `{set-box! <expr> <expr>}`
  - The `!` indicates that the form has a side effect
    - Just a convention, `!` is a normal character in Racket/Curly
  - Evaluates the first expression to a box
  - Evaluates the second expression
  - Stores that value in the box's location in memory
  - Produces that value as the result
    - Other design choices are possible for the result

Boxes in Curly-Box

- `{box <expr>}`
  - Allocates a new box in the store with the
  - Like C++ `void* x = new int; *x = expr;`
- `{unbox <expr>}`
  - Evaluates the expression to a box
    - dynamic type error if not a box
  - Produces the value for that box's location in memory
  - Like C++ `*x;`
- `{set-box! <expr> <expr>}`
  - The `!` indicates that the form has a side effect
    - Just a convention, `!` is a normal character in Racket/Curly
  - Evaluates the first expression to a box
  - Evaluates the second expression
  - Stores that value in the box's location in memory
  - Produces that value as the result
    - Other design choices are possible for the result
  - Like C++ `*x = expr;`

Example

Example

```
{letvar x {box 3}
  {letvar y x
  {begin {set-box! y 10}
    {unbox x}}}}
```

- Allocates a new box, with the value 3

Example

```
{letvar x {box 3}
  {letvar y x
  {begin {set-box! y 10}
    {unbox x}}}}
```

- Allocates a new box, with the value 3
- The variable x is given the value of that box

Example

```
{letvar x {box 3}
  {letvar y x
  {begin {set-box! y 10}
    {unbox x}}}}
```

- Allocates a new box, with the value 3
- The variable `x` is given the value of that box
- The variable `y` is given the value of `x`

Example

```
{letvar x {box 3}
  {letvar y x
  {begin {set-box! y 10}
    {unbox x}}}}
```

- Allocates a new box, with the value 3
- The variable `x` is given the value of that box
- The variable `y` is given the value of `x`
    - e.g. the value of the box

Example

```
{letvar x {box 3}
  {letvar y x
  {begin {set-box! y 10}
    {unbox x}}}}
```

- Allocates a new box, with the value 3
- The variable `x` is given the value of that box
- The variable `y` is given the value of `x`
  - e.g. the value of the box
- We write 10 to *the location that `y` points to*

Example

```
{letvar x {box 3}
  {letvar y x
  {begin {set-box! y 10}
    {unbox x}}}}
```

- Allocates a new box, with the value 3
- The variable `x` is given the value of that box
- The variable `y` is given the value of `x`
  - e.g. the value of the box
- We write 10 to *the location that `y` points to*
  - This is the same location that x points to

Example

```
{letvar x {box 3}
  {letvar y x
  {begin {set-box! y 10}
    {unbox x}}}}
```

- Allocates a new box, with the value 3
- The variable `x` is given the value of that box
- The variable `y` is given the value of `x`
  - e.g. the value of the box
- We write 10 to *the location that `y` points to*
  - This is the same location that `x` points to
- We produce the value from wherever `x` points

Example

```
{letvar x {box 3}
  {letvar y x
  {begin {set-box! y 10}
    {unbox x}}}}
```

- Allocates a new box, with the value 3
- The variable `x` is given the value of that box
- The variable `y` is given the value of `x`
  - e.g. the value of the box
- We write 10 to *the location that `y` points to*
  - This is the same location that `x` points to
- We produce the value from wherever `x` points
  - 10, since we updated its value

Interpreting Boxes

Semantics

- We *could* just use Racket mutation to implement boxes

Semantics

- We *could* just use Racket mutation to implement boxes
  - But we're not going to

Semantics

- We *could* just use Racket mutation to implement boxes
  - But we're not going to
- Recall: when we define a functional language, we get equations about the functions we define

Semantics

- We *could* just use Racket mutation to implement boxes
  - But we're not going to
- Recall: when we define a functional language, we get equations about the functions we define
  - e.g. `(interp (Plus x y))` is *mathematically equal* to `(+ (interp x) (interp y))`

Semantics

- We *could* just use Racket mutation to implement boxes
  - But we're not going to
- Recall: when we define a functional language, we get equations about the functions we define
  - e.g. `(interp (Plus x y))` is *mathematically equal* to `(+ (interp x) (interp y))`
- Our interpreters aren't just implementations

Semantics

- We *could* just use Racket mutation to implement boxes
  - But we're not going to
- Recall: when we define a functional language, we get equations about the functions we define
  - e.g. `(interp (Plus x y))` is *mathematically equal* to `(+ (interp x) (interp y))`
- Our interpreters aren't just implementations
  - They're *specifications* for the language

Semantics

- We *could* just use Racket mutation to implement boxes
  - But we're not going to
- Recall: when we define a functional language, we get equations about the functions we define
  - e.g. `(interp (Plus x y))` is *mathematically equal* to `(+ (interp x) (interp y))`
- Our interpreters aren't just implementations
  - They're *specifications* for the language
    - Semantics

Semantics

- We *could* just use Racket mutation to implement boxes
  - But we're not going to
- Recall: when we define a functional language, we get equations about the functions we define
  - e.g. `(interp (Plus x y))` is *mathematically equal* to `(+ (interp x) (interp y))`
- Our interpreters aren't just implementations
  - They're *specifications* for the language
    - Semantics
  - Equations we can use to reason about program behaviour

Semantics

- We *could* just use Racket mutation to implement boxes
  - But we're not going to
- Recall: when we define a functional language, we get equations about the functions we define
  - e.g. `(interp (Plus x y))` is *mathematically equal* to `(+ (interp x) (interp y))`
- Our interpreters aren't just implementations
  - They're *specifications* for the language
    - Semantics
  - Equations we can use to reason about program behaviour
- By using a purely functional approach, we define a semantics for Curly, even if it has mutation

Semantics

- We *could* just use Racket mutation to implement boxes
  - But we're not going to
- Recall: when we define a functional language, we get equations about the functions we define
  - e.g. `(interp (Plus x y))` is *mathematically equal* to `(+ (interp x) (interp y))`
- Our interpreters aren't just implementations
  - They're *specifications* for the language
    - Semantics
  - Equations we can use to reason about program behaviour
- By using a purely functional approach, we define a semantics for Curly, even if it has mutation
- The semantics is informal, but still a useful tool for *understanding* stateful programs and languages

The Store

- Data structure modelling memory

The Store

- Data structure modelling memory

The Store

- Data structure modelling memory

```
(define-type-alias Location Number)

(define-type Storage
  (cell [location : Location]
        [val : Value]))

(define-type-alias Store (Listof Storage))
(define mt-store empty)
```

- Store is a list of location-value pairs

The Store

- Data structure modelling memory

```
(define-type-alias Location Number)

(define-type Storage
  (cell [location : Location]
        [val : Value]))

(define-type-alias Store (Listof Storage))
(define mt-store empty)
```

- Store is a list of location-value pairs
  - Super inefficient version of a key-value store

The Store

- Data structure modelling memory

```
(define-type-alias Location Number)

(define-type Storage
  (cell [location : Location]
        [val : Value]))

(define-type-alias Store (Listof Storage))
(define mt-store empty)
```

- Store is a list of location-value pairs
  - Super inefficient version of a key-value store
    - e.g. Python dictionary with integer keys

The Store

- Data structure modelling memory

```
(define-type-alias Location Number)

(define-type Storage
  (cell [location : Location]
        [val : Value]))

(define-type-alias Store (Listof Storage))
(define mt-store empty)
```

- Store is a list of location-value pairs
  - Super inefficient version of a key-value store
    - e.g. Python dictionary with integer keys
  - Interface is what matters

Store Operations: override-store

Store Operations: override-store

```
;; (Storage Store -> Store)
(define override-store cons)
```

- Creates *a new store* that has the given value at the given location

Store Operations: override-store

```
;; (Storage Store -> Store)
(define override-store cons)
```

- Creates *a new store* that has the given value at the given location
  - If something was in that location, the new value overwrites it

Store Operations: override-store

```
;; (Storage Store -> Store)
(define override-store cons)
```

- Creates *a new store* that has the given value at the given location
  - If something was in that location, the new value overwrites it
  - In our implementation, this happens by adding the new value to the start of the list, so a search will always find it first

Store Operations: override-store

```
;; (Storage Store -> Store)
(define override-store cons)
```

- Creates *a new store* that has the given value at the given location
  - If something was in that location, the new value overwrites it
  - In our implementation, this happens by adding the new value to the start of the list, so a search will always find it first
- Original store is unchanged

Store Operations: override-store

```
;; (Storage Store -> Store)
(define override-store cons)
```

- Creates *a new store* that has the given value at the given location
  - If something was in that location, the new value overwrites it
  - In our implementation, this happens by adding the new value to the start of the list, so a search will always find it first
- Original store is unchanged
  - Purely functional *implementation language*

Store Operations: new-loc

Store Operations: new-loc

```
(define (new-loc [sto : Store]) : Location
  (+ 1 (max-address sto)))

(define (max-address [sto : Store]) : Location
  (type-case (Listof Storage) sto
   [empty 0]
   [(cons c rst-sto) (max (cell-location c)
                          (max-address rst-sto))]))
```

- Get a new `Location` that does not yet have a value in the store

Store Operations: new-loc

```
(define (new-loc [sto : Store]) : Location
  (+ 1 (max-address sto)))

(define (max-address [sto : Store]) : Location
  (type-case (Listof Storage) sto
   [empty 0]
   [(cons c rst-sto) (max (cell-location c)
                          (max-address rst-sto))]))
```

- Get a new `Location` that does not yet have a value in the store
  - `max-address` is a helper that enables this

Store Operations: new-loc

```
(define (new-loc [sto : Store]) : Location
  (+ 1 (max-address sto)))

(define (max-address [sto : Store]) : Location
  (type-case (Listof Storage) sto
   [empty 0]
   [(cons c rst-sto) (max (cell-location c)
                          (max-address rst-sto))]))
```

- Get a new `Location` that does not yet have a value in the store
  - `max-address` is a helper that enables this
- Used with `override-store` to extend a store to a new one with an additional location

Store Operations: fetch

Store Operations: fetch

```
(define (fetch [l : Location] [sto : Store]) : Value
  (type-case (Listof Storage) sto
   [empty (error 'interp "unallocated location")]
   [(cons c rst-sto) (if (equal? l (cell-location c))
                         (cell-val c)
                         (fetch l rst-sto))]))
```

- Get the value at the given location

Store Operations: fetch

```
(define (fetch [l : Location] [sto : Store]) : Value
  (type-case (Listof Storage) sto
   [empty (error 'interp "unallocated location")]
   [(cons c rst-sto) (if (equal? l (cell-location c))
                         (cell-val c)
                         (fetch l rst-sto))]))
```

- Get the value at the given location
  - Finds the first one in the list e.g. from the most recent `store-override`

A Store Passing Interpreter

- Recall how with generative recursion, we added an extra parameter to our function to simulate something being updated

A Store Passing Interpreter

- Recall how with generative recursion, we added an extra parameter to our function to simulate something being updated
- To interpret a language with mutable state, we'll do something similar

A Store Passing Interpreter

- Recall how with generative recursion, we added an extra parameter to our function to simulate something being updated
- To interpret a language with mutable state, we'll do something similar
  - Add a `Store` parameter to `interp`

A Store Passing Interpreter

- Recall how with generative recursion, we added an extra parameter to our function to simulate something being updated
- To interpret a language with mutable state, we'll do something similar
  - Add a `Store` parameter to `interp`
  - Make `interp` produce both a `Value` and an updated `Store`

A Store Passing Interpreter

- Recall how with generative recursion, we added an extra parameter to our function to simulate something being updated
- To interpret a language with mutable state, we'll do something similar
  - Add a `Store` parameter to `interp`
  - Make `interp` produce both a `Value` and an updated `Store`

A Store Passing Interpreter

- Recall how with generative recursion, we added an extra parameter to our function to simulate something being updated
- To interpret a language with mutable state, we'll do something similar
  - Add a `Store` parameter to `interp`
  - Make `interp` produce both a `Value` and an updated `Store`

```
(define-type Result
  (v*s [v : Value] [s : Store]))
(define (interp [env : Env]
                [e : Expr]
                [sto : Store]) : Result
  ....)
```

- Custom pair-type for value-store pairs

A Store Passing Interpreter

- Recall how with generative recursion, we added an extra parameter to our function to simulate something being updated
- To interpret a language with mutable state, we'll do something similar
  - Add a `Store` parameter to `interp`
  - Make `interp` produce both a `Value` and an updated `Store`

```
(define-type Result
  (v*s [v : Value] [s : Store]))
(define (interp [env : Env]
                [e : Expr]
                [sto : Store]) : Result
  ....)
```

- Custom pair-type for value-store pairs
- `interp` takes an expression, and environment, and a store

Updating The Interpreter: Plus

Updating The Interpreter: Plus

```
(define (interp [env : Env]
                [e : Expr]
                [sto : Store]) : Result
  (type-case Expr e
   [(Plus l r)
     (type-case Result (interp l env sto)
       [(v*s v-l sto-l)
        (type-case Result (interp r env sto-l)
          [(v*s v-r sto-r)
           (v*s (liftVal2 + v-l v-r) sto-r)])])])]
```

- Interp the left, getting its value and store

Updating The Interpreter: Plus

```
(define (interp [env : Env]
                [e : Expr]
                [sto : Store]) : Result
  (type-case Expr e
   [(Plus l r)
     (type-case Result (interp l env sto)
       [(v*s v-l sto-l)
        (type-case Result (interp r env sto-l)
          [(v*s v-r sto-r)
           (v*s (liftVal2 + v-l v-r) sto-r)])])])]
```

- Interp the left, getting its value and store
- Interp the right *using the new store from the left*

Updating The Interpreter: Plus

```
(define (interp [env : Env]
                [e : Expr]
                [sto : Store]) : Result
  (type-case Expr e
   [(Plus l r)
     (type-case Result (interp l env sto)
       [(v*s v-l sto-l)
        (type-case Result (interp r env sto-l)
          [(v*s v-r sto-r)
           (v*s (liftVal2 + v-l v-r) sto-r)])])])]
```

- Interp the left, getting its value and store
- Interp the right *using the new store from the left*
- Do the addition, *then return the resulting store from the right*

Updating The Interpreter: Plus

```
(define (interp [env : Env]
                [e : Expr]
                [sto : Store]) : Result
  (type-case Expr e
   [(Plus l r)
     (type-case Result (interp l env sto)
       [(v*s v-l sto-l)
        (type-case Result (interp r env sto-l)
          [(v*s v-r sto-r)
           (v*s (liftVal2 + v-l v-r) sto-r)])])]
```

- Interp the left, getting its value and store
- Interp the right *using the new store from the left*
- Do the addition, *then return the resulting store from the right*
  - Design choice: which operand we eval first makes a difference

Aside: Racket Macros

- Racket lets you define your own syntactic sugar

Aside: Racket Macros

- Racket lets you define your own syntactic sugar
  - Add or rewrite Racket syntax

Aside: Racket Macros

- Racket lets you define your own syntactic sugar
  - Add or rewrite Racket syntax
- These are called macros

Aside: Racket Macros

- Racket lets you define your own syntactic sugar
  - Add or rewrite Racket syntax
- These are called macros
- C++ has *textual macros*

Aside: Racket Macros

- Racket lets you define your own syntactic sugar
  - Add or rewrite Racket syntax
- These are called macros
- C++ has *textual macros*
  - Just inserts a string into the text of your source file

Aside: Racket Macros

- Racket lets you define your own syntactic sugar
  - Add or rewrite Racket syntax
- These are called macros
- C++ has *textual macros*
  - Just inserts a string into the text of your source file
- Racket has *syntactic macros*

Aside: Racket Macros

- Racket lets you define your own syntactic sugar
  - Add or rewrite Racket syntax
- These are called macros
- C++ has *textual macros*
  - Just inserts a string into the text of your source file
- Racket has *syntactic macros*
  - Actually lets you write functions that run at compile-time and inspect the syntax given

Aside: Racket Macros

- Racket lets you define your own syntactic sugar
  - Add or rewrite Racket syntax
- These are called macros
- C++ has *textual macros*
  - Just inserts a string into the text of your source file
- Racket has *syntactic macros*
  - Actually lets you write functions that run at compile-time and inspect the syntax given
- `define-type` and `type-case` are both Racket macros

Aside: Racket Macros

- Racket lets you define your own syntactic sugar
  - Add or rewrite Racket syntax
- These are called macros
- C++ has *textual macros*
  - Just inserts a string into the text of your source file
- Racket has *syntactic macros*
  - Actually lets you write functions that run at compile-time and inspect the syntax given
- `define-type` and `type-case` are both Racket macros
- The entire `plait` type system is implemented with Racket macros

A Macro for Interpreting with Stores

A Macro for Interpreting with Stores

```
(define-syntax-rule
  (with [(v-id sto-id) call]
    body)
  (type-case Result call
    [(v*s v-id sto-id) body]))
```

- I won't ask you to write a macro on an exam

A Macro for Interpreting with Stores

```
(define-syntax-rule
  (with [(v-id sto-id) call]
    body)
  (type-case Result call
    [(v*s v-id sto-id) body]))
```

- I won't ask you to write a macro on an exam
- I might check whether you know that

A Macro for Interpreting with Stores

```
(define-syntax-rule
  (with [(v-id sto-id) call]
    body)
  (type-case Result call
    [(v*s v-id sto-id) body]))
```

- I won't ask you to write a macro on an exam
- I might check whether you know that
  - Macros extend the syntax of a language using the language itself

A Macro for Interpreting with Stores

```
(define-syntax-rule
  (with [(v-id sto-id) call]
    body)
  (type-case Result call
    [(v*s v-id sto-id) body]))
```

- I won't ask you to write a macro on an exam
- I might check whether you know that
  - Macros extend the syntax of a language using the language itself
  - Macros are code that is run at compile-time

Using Our Macro

Using Our Macro

```
(define (interp [env : Env]
                [e : Expr]
                [sto : Store]) : Result
  (type-case Expr e
    [(Plus l r)
     (with [(v-l sto-l) (interp l env sto)]
       (with [(v-r sto-r) (interp r env sto-l)]
         (v*s (liftVal2 + v-l v-r) sto-r)))]
```

- Much more succinct: call interp, and immediately give a name to the store and value we get as a result

Using Our Macro

```
(define (interp [env : Env]
                [e : Expr]
                [sto : Store]) : Result
  (type-case Expr e
    [(Plus l r)
     (with [(v-l sto-l) (interp l env sto)]
       (with [(v-r sto-r) (interp r env sto-l)]
         (v*s (liftVal2 + v-l v-r) sto-r)))]
```

- Much more succinct: call interp, and immediately give a name to the store and value we get as a result
- Updating all the other cases is similar

Using Our Macro

```
(define (interp [env : Env]
                [e : Expr]
                [sto : Store]) : Result
  (type-case Expr e
    [(Plus l r)
     (with [(v-l sto-l) (interp l env sto)]
       (with [(v-r sto-r) (interp r env sto-l)]
         (v*s (liftVal2 + v-l v-r) sto-r)))]
```

- Much more succinct: call interp, and immediately give a name to the store and value we get as a result
- Updating all the other cases is similar
  - See in-class Racket if time permits

Boxes as Values

- We represent boxes using locations in the store

Boxes as Values

- We represent boxes using locations in the store
  - What value is in the box depends on the current store

Boxes as Values

- We represent boxes using locations in the store
  - What value is in the box depends on the current store

Boxes as Values

- We represent boxes using locations in the store
  - What value is in the box depends on the current store

```
(define-type Value
  (ClosureV [arg : Symbol]
            [body : Expr]
            [env : Env])
  (NumV [num : Number])
  (BoxV [loc : Location]))
```

Interpreting Box

Interpreting Box

```
[(Box a)
   (with [(v sto-v) (interp a env sto)]
     (let ([l (new-loc sto-v)])
       (v*s (BoxV l)
            (override-store (cell l v)
                            sto-v))))]
```

- Interpret the value we're putting in the box

Interpreting Box

```
[(Box a)
   (with [(v sto-v) (interp a env sto)]
     (let ([l (new-loc sto-v)])
       (v*s (BoxV l)
            (override-store (cell l v)
                            sto-v))))]
```

- Interpret the value we're putting in the box
  - And get the store from this interpretation

Interpreting Box

```
[(Box a)
   (with [(v sto-v) (interp a env sto)]
     (let ([l (new-loc sto-v)])
       (v*s (BoxV l)
            (override-store (cell l v)
                            sto-v))))]
```

- Interpret the value we're putting in the box
  - And get the store from this interpretation
- Create a new store that has the value at a new address

Interpreting Box

```
[(Box a)
   (with [(v sto-v) (interp a env sto)]
     (let ([l (new-loc sto-v)])
       (v*s (BoxV l)
            (override-store (cell l v)
                            sto-v))))]
```

- Interpret the value we're putting in the box
  - And get the store from this interpretation
- Create a new store that has the value at a new address
- Return the box with the new location

Interpreting Box

```
[(Box a)
   (with [(v sto-v) (interp a env sto)]
     (let ([l (new-loc sto-v)])
       (v*s (BoxV l)
            (override-store (cell l v)
                            sto-v))))]
```

- Interpret the value we're putting in the box
  - And get the store from this interpretation
- Create a new store that has the value at a new address
- Return the box with the new location
  - Return store is the store with the new location and value

Interpreting Unbox

Interpreting Unbox

```
[(Unbox a)
     (with [(v sto-v) (interp a env sto)]
       (type-case Value v
         [(BoxV l) (v*s (fetch l sto-v)
                        sto-v)]
         [else (error 'interp "not a box")]))]
```

- Interpret the expression to a value

Interpreting Unbox

```
[(Unbox a)
     (with [(v sto-v) (interp a env sto)]
       (type-case Value v
         [(BoxV l) (v*s (fetch l sto-v)
                        sto-v)]
         [else (error 'interp "not a box")]))]
```

- Interpret the expression to a value
  - Get the value and the store from this, since it might have had side effects

Interpreting Unbox

```
[(Unbox a)
     (with [(v sto-v) (interp a env sto)]
       (type-case Value v
         [(BoxV l) (v*s (fetch l sto-v)
                        sto-v)]
         [else (error 'interp "not a box")]))]
```

- Interpret the expression to a value
  - Get the value and the store from this, since it might have had side effects
- Check if the value is a Box

Interpreting Unbox

```
[(Unbox a)
     (with [(v sto-v) (interp a env sto)]
       (type-case Value v
         [(BoxV l) (v*s (fetch l sto-v)
                        sto-v)]
         [else (error 'interp "not a box")]))]
```

- Interpret the expression to a value
  - Get the value and the store from this, since it might have had side effects
- Check if the value is a Box
  - Type error otherwise

Interpreting Unbox

```
[(Unbox a)
     (with [(v sto-v) (interp a env sto)]
       (type-case Value v
         [(BoxV l) (v*s (fetch l sto-v)
                        sto-v)]
         [else (error 'interp "not a box")]))]
```

- Interpret the expression to a value
  - Get the value and the store from this, since it might have had side effects
- Check if the value is a Box
  - Type error otherwise
- Get its location, and produce whatever value was at that location

Interpreting Unbox

```
[(Unbox a)
     (with [(v sto-v) (interp a env sto)]
       (type-case Value v
         [(BoxV l) (v*s (fetch l sto-v)
                        sto-v)]
         [else (error 'interp "not a box")]))]
```

- Interpret the expression to a value
  - Get the value and the store from this, since it might have had side effects
- Check if the value is a Box
  - Type error otherwise
- Get its location, and produce whatever value was at that location
  - With the store from evaluating the box

Interpreting Set-box!

Interpreting Set-box!

```
[(Setbox bx val)
     (with [(v-b sto-b) (interp bx env sto)]
       (with [(v-v sto-v) (interp val env sto-b)]
         (type-case Value v-b
           [(BoxV l)
            (v*s v-v
                 (override-store (cell l v-v)
                                 sto-v))]
           [else (error 'interp "not a box")])))]
```

- Interpret the box to a value

Interpreting Set-box!

```
[(Setbox bx val)
     (with [(v-b sto-b) (interp bx env sto)]
       (with [(v-v sto-v) (interp val env sto-b)]
         (type-case Value v-b
           [(BoxV l)
            (v*s v-v
                 (override-store (cell l v-v)
                                 sto-v))]
           [else (error 'interp "not a box")])))]
```

- Interpret the box to a value
  - Get the store from this execution

Interpreting Set-box!

```
[(Setbox bx val)
     (with [(v-b sto-b) (interp bx env sto)]
       (with [(v-v sto-v) (interp val env sto-b)]
         (type-case Value v-b
           [(BoxV l)
            (v*s v-v
                 (override-store (cell l v-v)
                                 sto-v))]
           [else (error 'interp "not a box")])))]
```

- Interpret the box to a value
  - Get the store from this execution
- Interpret the expression to be stored to a value

Interpreting Set-box!

```
[(Setbox bx val)
     (with [(v-b sto-b) (interp bx env sto)]
       (with [(v-v sto-v) (interp val env sto-b)]
         (type-case Value v-b
           [(BoxV l)
            (v*s v-v
                 (override-store (cell l v-v)
                                 sto-v))]
           [else (error 'interp "not a box")])))]
```

- Interpret the box to a value
  - Get the store from this execution
- Interpret the expression to be stored to a value
  - In the store we previously computed, getting a new store

Interpreting Set-box!

```
[(Setbox bx val)
     (with [(v-b sto-b) (interp bx env sto)]
       (with [(v-v sto-v) (interp val env sto-b)]
         (type-case Value v-b
           [(BoxV l)
            (v*s v-v
                 (override-store (cell l v-v)
                                 sto-v))]
           [else (error 'interp "not a box")])))]
```

- Interpret the box to a value
  - Get the store from this execution
- Interpret the expression to be stored to a value
  - In the store we previously computed, getting a new store
- Check that the box value is actually a box

Interpreting Set-box!

```
[(Setbox bx val)
     (with [(v-b sto-b) (interp bx env sto)]
       (with [(v-v sto-v) (interp val env sto-b)]
         (type-case Value v-b
           [(BoxV l)
            (v*s v-v
                 (override-store (cell l v-v)
                                 sto-v))]
           [else (error 'interp "not a box")])))]
```

- Interpret the box to a value
  - Get the store from this execution
- Interpret the expression to be stored to a value
  - In the store we previously computed, getting a new store
- Check that the box value is actually a box
- Return the other value, *in the new store with the box location overwritten*

Interpreting Begin

Interpreting Begin

```
[(Begin l r)
     (with [(v-l sto-l) (interp l env sto)]
       (interp r env sto-l))]
```

- Evaluate the first expression *but do nothing with the value*

Interpreting Begin

```
[(Begin l r)
     (with [(v-l sto-l) (interp l env sto)]
       (interp r env sto-l))]
```

- Evaluate the first expression *but do nothing with the value*
  - Just get the store that's the result of evaluating it

Interpreting Begin

```
[(Begin l r)
     (with [(v-l sto-l) (interp l env sto)]
       (interp r env sto-l))]
```

- Evaluate the first expression *but do nothing with the value*
  - Just get the store that's the result of evaluating it
- Evaluate the second expression *in the store the first returned*

Stores vs Environment

- Environments enable *static scope*

Stores vs Environment

- Environments enable *static scope*
  - Follow the stack like structure of function calls

Stores vs Environment

- Environments enable *static scope*
  - Follow the stack like structure of function calls
  - When we return from a function call, we keep evaluating in the old environment

Stores vs Environment

- Environments enable *static scope*
  - Follow the stack like structure of function calls
  - When we return from a function call, we keep evaluating in the old environment
- Stores are inherently dynamic

Stores vs Environment

- Environments enable *static scope*
  - Follow the stack like structure of function calls
  - When we return from a function call, we keep evaluating in the old environment
- Stores are inherently dynamic
  - We *want* them to always take the value from the point `unbox` is executed

Stores vs Environment

- Environments enable *static scope*
  - Follow the stack like structure of function calls
  - When we return from a function call, we keep evaluating in the old environment
- Stores are inherently dynamic
  - We *want* them to always take the value from the point `unbox` is executed
  - Linearly threaded through the program

Stores vs Environment

- Environments enable *static scope*
  - Follow the stack like structure of function calls
  - When we return from a function call, we keep evaluating in the old environment
- Stores are inherently dynamic
  - We *want* them to always take the value from the point `unbox` is executed
  - Linearly threaded through the program
  - Once we've updated a store (by generating a new one), *we never use the old store again*

Stores vs Environment

- Environments enable *static scope*
  - Follow the stack like structure of function calls
  - When we return from a function call, we keep evaluating in the old environment
- Stores are inherently dynamic
  - We *want* them to always take the value from the point `unbox` is executed
  - Linearly threaded through the program
  - Once we've updated a store (by generating a new one), *we never use the old store again*
    - … until we implement more advanced features e.g. undo, backtracking