Concepts of Programming Languages

Stateful languages

Prof. Dr. Guido Salvaneschi

Outline

- State, state in Scheme, state in our languages
- An interpretation style for modeling the effect of state
- Mutable data structures
- Variables
- Summary

Use of State is not Essential

```
C:

int i;
sum = 0;
for (i=0; i < 100; i++) {
  sum += f(i);
}
```

```
Haskell:

foldr (+) 0 (map f [0..99])
```

State May Cause Problems

- There are problems when dealing with state:
  - Have to reason differently about program before and after every mutation.
  - Easily causes bugs in multi-threaded environments.

```
b = 2
c = 3
a = b + c
print(a)  // 5
c = 5
print(a)  // 7
```

The same instruction
does something different

However ... Legitimate Uses of State

- In the real world, there are events that truly alter the state.
- Cars consume fuel, money is deposited or withdrawn from bank-accounts, ...

- Copying data structures is not a good fit in these cases.
- A mutable data structure might be the best solution.

Variables and Mutable Data Structures

- Many languages like C/C++, Java, or Scala have **two kinds of mutations**:
  - mutating the value bound to an identifier,
  - mutating a value in a container.

Variables and Mutable Data Structures

- **Mutating an identifier**:

```
int i;
i = nextPrime(12)
i = nextPrime(18)
```

  - `i` must literally be an identifier.
  - Identifiers with changeable values are called **variables**.

- **Mutating a value in a container**:

`o.f = e`	(object in Java)
`list.add(e)`	(collection in Java)

- Containers change their internal state but do not immediately reflect this change to the outside.

Terminology: State in Scheme

- Scheme **boxes** model **mutable containers** with one field.
- Scheme provides three operations on boxes:
  - **`box`** to create a new box,
  - **`set-box!`** to change the value of a box,
  - **`unbox`** to read the value of a box.

- Scheme also supports **variables** and the **`set!`** operation to change their value.

- Scheme supports sequencing of operations by means of **`begin`** blocks.

State in Scala

- So far we mostly used *immutable* data structures in Scala:
  - `case class Plus(x: Exp, y: Exp) extends Exp`
  - `val v = “abc” + “def”`
  - `def f(a: Int, b: Int) = a*a + 2*a*b + b*b`
  - `object o { ... }`

State in Scala

- Scala also has stateful constructs: variables and boxes

  -
    ```
    var v = 1
    while (v < 100) { v = v * 3 }
    ```

  -
    ```
    object o {
      var f: Int
    }
    o.f = 17
    ```
- a variable can be ***assigned*** different values over time
- a box can ***contain*** different values over time
- a box can be passed around to allow mutation by others:
  ```
  def foo(o: o.type) {
    o.f = 13;
  }
  ```

State in this Lecture

We will implement both variables and boxes with their operations to understand their differences and interactions with other language features.

We will also implement sequencing.

Our starting point: CFLAE

- Starting with CFLAE...

```
<CFLAE> ::= <num>
          | {+ <CFLAE> <CFLAE>}
          | {- <CFLAE> <CFLAE>}
          | {with {<id> <CFLAE>} <CFLAE>}
          | <id>
          | {fun {<id>} <CFLAE>}
          | {<CFLAE> <CFLAE>}
          | {* <CFLAE> <CFLAE>}
          | {if0 <CFLAE> <CFLAE> <CFLAE>}
```

Adding State Syntactically

- ... will add constructs for defining boxes, for setting/getting the value in a box, for changing the value of a variable, and for sequencing two expressions.

```
<BCFLAE> ::= <num>
           | {+ <SCFLAE> <SCFLAE>}
           | {- <SCFLAE> <SCFLAE>}
           | {let {<id> <SCFLAE>} <SCFLAE>}
           | <id>
           | {fun {<id>} <SCFLAE>}
           | {<SCFLAE> <SCFLAE>}
           | {* <SCFLAE> <SCFLAE>}
           | {if0 <SCFLAE> <SCFLAE> <CFLAE>}

           | {newbox <SCFLAE>}
           | {setbox <SCFLAE> <SCFLAE>}
           | {openbox <SCFLAE>}
           | {set <id> <SCFLAE>}
           | {seqn <SCFLAE> <SCFLAE>}
```

Adding State Semantically

- How do we implement these constructs?

```
<SCFLAE> ::=
             ...
             | {newbox <SCFLAE>}
             | {setbox <SCFLAE> <SCFLAE>}
             | {openbox <SCFLAE>}
             | {set <id> <SCFLAE>}
             | {seqn <SCFLAE> <SCFLAE>}
```

- Two choices:
  - Meta interpreter: Using Scala boxes and variables
  - Syntactic interpreter: By implementing a mutation-free interpreter.

- Which one do you prefer?

Adding State Semantically

Modeling state and mutation requires a new interpretation style ...

Outline

- State, state in Scheme, state in our languages
- An interpretation style for modeling the effect of state
- Mutable data structures
- Variables
- Summary

Reflecting on Semantics of Sequences

- A first try to implement **seqn** expressions following the pattern that we have used so far...

```
def interp(expr: SCFLAE, env: Env): Val = expr match {
  case Seqn(e1, e2) => {
    interp(e1, env)
    interp(e2, env)
  }
}
```

What do you think?
Will this work?

Reflecting on Semantics of Sequences

Quiz: What do the following code pieces evaluate to?

```
val box = Box(0)
box.value = 1 + box.value
box.value
```

```
var b = 0
b = 1 + b
b
```

```
Let('b, NewBox(0),
  Seqn(
    SetBox('b, Add(1, OpenBox('b))),
    OpenBox('b)))
```

```
Let('b, 0,
  Seqn(
    SetId('b, Add(1, 'b)),
    'b))
```

Reflecting on Semantics of Sequences

```
val box = Box(0)
box.value = 1 + box.value
box.value
```

```
var b = 0
b = 1 + b
b
```

```
Let('b, NewBox(0),
  Seqn(
    SetBox('b, Add(1, OpenBox('b))),
    OpenBox('b)))
```

```
Let('b, 0,
  Seqn(
    SetId('b, Add(1, 'b)),
    'b))
```

- The answer is 1.
- The mutation in the first operation in the sequence has an effect on the output of the second (which would otherwise evaluate to 0).

Reflecting on Semantics of Sequences

```
def interp(expr: SCFLAE, env: Env): Val = expr match {
  case Seqn(e1, e2) => {
    interp(e1, env)
    interp(e2, env)
  }
}
```

- This will not work, because there is no way the interpretation of the first sequent can affect the interpretation of the second.
- Somehow we need to **"chain" interpretation effects**.

How can we do that?

A First Idea for Chaining Effects

Use environments that flow through interpreter calls as the medium for chaining the mutation effects.

Using Environments for Chaining Effects

- The interpreter returns both the value of an expression and an updated environment reflecting mutation effects.
- **`interp`** returns a tuple **`Value × Environment`**

Using Environments for Chaining Effects

- The interpretation of **`seqn`** uses the environment resulting from evaluating the first sequent to interpret the second.
  - **`(interp e1 ...)`** returns a modified environment
  - **`(interp e2 ...)`** uses updated environment.

```scala
def interp(expr: SCFLAE, env: Env): (Val, Env) = expr match {
  //...
  case Seqn(e1, e2) => {
    val (res, env1) = interp(e1, env)
    interp(e2, env1)
  }
}
```

What do you think?
Will this work?

Using Environments for Chaining Effects

- Consider the following code. What would you expect it to evaluate to? What would it actually evaluate to?

```
{ val x = 10 }
x
```

```
Seqn(
    Let('x, 10, 'x),
    'x)
```

Using Environments for Chaining Effects

```
{ val x = 10 }
x
```

```
Seqn(
    Let('x, 10, 'x),
    'x)
```

The new “environment-passing style” of interpretation tells that **x** in the second expression in sequence will be evaluated in an environment that binds **x** to `10`.

This contradicts static scoping, which tells us that **x** in the second expression is unbound.

Using Environments for Chaining Effects

```
{ val x = 10 }
x
```

```
Seqn(
    Let('x, 10, 'x),
    'x)
```

We will have to remove bindings introduced by **`let ...`** complicates the implementation...

But, even if we are willing to work out the details, does this really solve the problem?

Using Environments for Chaining Effects

- Consider the following code.
- What would you expect it to evaluate to?
- What would it evaluate to using environment-passing style?

```
val a = Box(1)
def f(x: Int) = x + a.value
a.value = 2
println(f(5))
```

```
Let('a, NewBox(1),
  Let('f, Fun('x, Add('x, OpenBox('a))),
    Seqn(
      SetBox('a, 2),
      App('f, 5))))
```

Using Environments for Chaining Effects

- We would like the effect of the first sequent of the seqn block to be visible at function application.

```
val a = Box(1)
def f(x: Int) = x + a.value
a.value = 2
println(f(5))
```

- However, even if we pass the updated environment to the interpretation of the application, this will be ignored since f closes the body in the **environment at definition time**.

- We would have to use the environment at application time, which introduces dynamic scoping!

Using Environments for Chaining Effects

Static scoping rules for function application will defeat environment-passing style.

Bottom Line

Using environments for chaining effects will not work!

Any other idea?

The Solution: Introducing the Store

- We need **two repositories of information**:
  - The **environment** for statically scoped bindings.
  - The **store** for tracking dynamic changes.

The Solution: Introducing the Store

- We need **two repositories of information**:
  - The **environment** for statically scoped bindings.
  - The **store** for tracking dynamic changes.

  - **Environment** maps **identifiers to locations** in the store.
  - **Store** maps **locations to values**.

```
type Env = Map[Symbol, Location]
type Store = Map[Location, Val]
```

  - **Identifier lookup** becomes a **two-stage process**:

Store-Passing Interpretation Style

Will use a modified store to chain the interpretation of expressions. Interpreter returns a tuple **`Value × Store`**

```
def interp(expr: SCFLAE, env: Env, store: Store): (Val, Store)
```

This style of passing the current store in and the updated store out of every expression's evaluation is called **store-passing style**.

The Interpreter

```
def interp(expr: SCFLAE, env: Env, store: Store): (Val, Store)
```

Interpreting Terms that are Syntactically Values

- Terms that are already syntactically values do not affect the store (since they require no further evaluation).

- They return the store unaltered.

```
def interp(expr: SCFLAE, env: Env, store: Store): (Val, Store) = expr match
{
  // ...
  case Num(n) => (NumV(n), store)
  case Fun(arg, body) => (Closure(arg, body, env), store)
  // ...
  case Id(name) => (store(env(name)), store)
  // ...
}
```

Interpreting Terms that are Syntactically Values

- Terms that are already syntactically values do not affect the store (since they require no further evaluation).

- They return the store unaltered.

```
def interp(expr: SCFLAE, env: Env, store: Store): (Val, Store) = expr match
{
  // ...
  case Num(n) => (NumV(n), store)
  case Fun(arg, body) => (Closure(arg, body, env), store)
  // ...
  case Id(name) => (store(env(name)), store)
  // ...
}
```

Interpreting `if0` Conditionals

```
def interp(expr: SCFLAE, env: Env, store: Store): (Val, Store) = expr match {
  // ...
  case If0(test, truth, falsity) => {
    val (res, _) = interp(test, env, store)
    res match {
      case NumV(n) => {
        if (n == 0) interp(truth, env, store)
        else interp(falsity, env, store)
    }
    case _ => sys.error("can only test numbers, but got: " + res)
  }
}
```

What about this interpretation of `if0` expressions?

Recall ...

```
<SCFLAE> ::=
           ...
         | {if0 <SCFLAE> <SCFLAE> <SCFLAE>}
```

Interpreting `if0` Conditionals

```
def interp(expr: SCFLAE, env: Env, store: Store): (Val, Store) = expr match {
  // ...
  case If0(test, truth, falsity) => {
    val (res, _) = interp(test, env, store)
    res match {
      case NumV(n) => {
        if (n == 0) interp(truth, env, store)
        else interp(falsity, env, store)
    }
    case _ => sys.error("can only test numbers, but got: " + res)
  }
}
```

What about this interpretation of **`if0`** expressions?

```
Let('b, 0,
    If0(Seqn(SetId('b, 5), 5),
        1, 'b))
```

$\longrightarrow$

```
???
```

Interpreting `if0` Conditionals

- For **`if0`**, we need to use the modified store returned by the interpretation of the test in the interpretation of the conditional branches.
- This allows to propagate possible mutations in the test.

```
def interp(expr: SCFLAE, env: Env, store: Store): (Val, Store) = expr match {
  // ...
  case If0(test, truth, falsity) => {
    val (res, store1) = interp(test, env, store)
    res match {
      case NumV(n) => {
        if (n == 0) interp(truth, env, store1)
        else interp(falsity, env, store1)
    }
    case _ => sys.error("can only test numbers, but got: " + res)
  }
}
```

Left or Right?

- When it comes to arithmetic and function applications expressions, we have to decide on the order of evaluation.

```
Let('b, 4,
    Add('b,
      Seqn(
        SetId('b, 5),
        'b)))
```

With left-to-right evaluation the result is 9.

With right-to-left evaluation the result is 10.

We will use a left-to-right and function-before-argument order of execution.

Arithmetic

```
def interp(expr: SCFLAE, env: Env, store: Store): (Val, Store) = expr match
{
  // ...
  case Add(lhs, rhs) => {
    val (lhsv, store1) = interp(lhs, env, store)
    lhsv match {
      case NumV(n1) => {
        val (rhsv, store2) = interp(rhs, env, store1)
        rhsv match {
          case NumV(n2) => (NumV(n1 + n2), store2)
          case _ => sys.error(
            "can only add numbers, but got right %s".format(rhsv))
        }
      }
      case _ => sys.error(
        "can only add numbers, but got left '%s'".format(lhsv))
    }
  }
}
```

Function Application

```
def interp(expr: SCFLAE, env: Env, store: Store): (Val, Store) = expr match {
  ...
  case App(funExpr, argExpr) => { ... }
}
```

1. Interpret **`funExpr`** in **`env`** and **`store`** (function-before-arg order), yielding **`funValue`** (a closure) and **`funStore`**.
2. Interpret **`argExpr`** in **`env`** and **`funStore`**, yielding **`argValue`** and **`argStore`**.
3. Interpret **`funValue`**'s body in the following store and environment:
    1. Store: Reserve a new location in **`argStore`** (**`newLoc`**) and put **`argValue`** to it.
    2. Environment: Extend **`funValue`**'s environment with a binding for **`funValue`**'s argument to **`newLoc`**.

Function Application

```
def interp(expr: SCFLAE, env: Env, store: Store): (Val, Store) = expr match {
  ...
  case App(funExpr, argExpr) => {
    val (funV, funStore) = interp(funExpr, env, store)
    val (argV, argStore) = interp(argExpr, env, funStore)
    funV match {
      case Closure(fParam, fBody, fEnv) => {
        val newLoc = nextLocation
        interp(fBody, fEnv + (fParam -> newLoc), argStore + (newLoc -> argV))
      }
      case _ => sys.error("can only apply functions, but got: " + funV)
    }
  }
}
```

- **Argument bound to a new store location** in environment.
- The store to extend is the most recent one at application time.
- The environment taken from time of function definition.

Implementing Sequencing

With store-passing style, implementing **`seqn`** is straightforward:

```
case Seqn(e1, e2) => {
  val (v1, store1) = interp(e1, env, store)
  interp(e2, env, store1)
}
```

Outline

- State, state in Scheme, state in our languages
- An interpretation style for modeling the effect of state
- Mutable data structures
- Variables
- Summary

Hm... What are Boxes?

- Do we really understand the semantics of boxes values to start with?

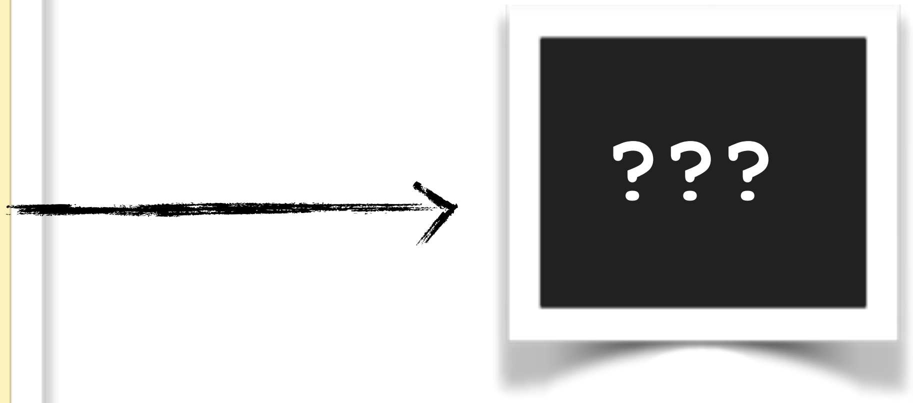
- Quiz: What does the following evaluate to?

```scala
class Box[A](var value: A)
val b1 = new Box(5)
val b2 = new Box(5)
val b3 = b1

b1 == b2

b1.value = 6

b1.value
b2.value
b3.value
```



Hm... What are Boxes?

- Do we really understand the semantics of Scheme boxes to start with?

- Quiz: What does the following evaluate to?

```scala
class Box[A](var value: A)
val b1 = new Box(5)
val b2 = new Box(5)
val b3 = b1

b1 == b2

b1.value = 6

b1.value
b2.value
b3.value
```

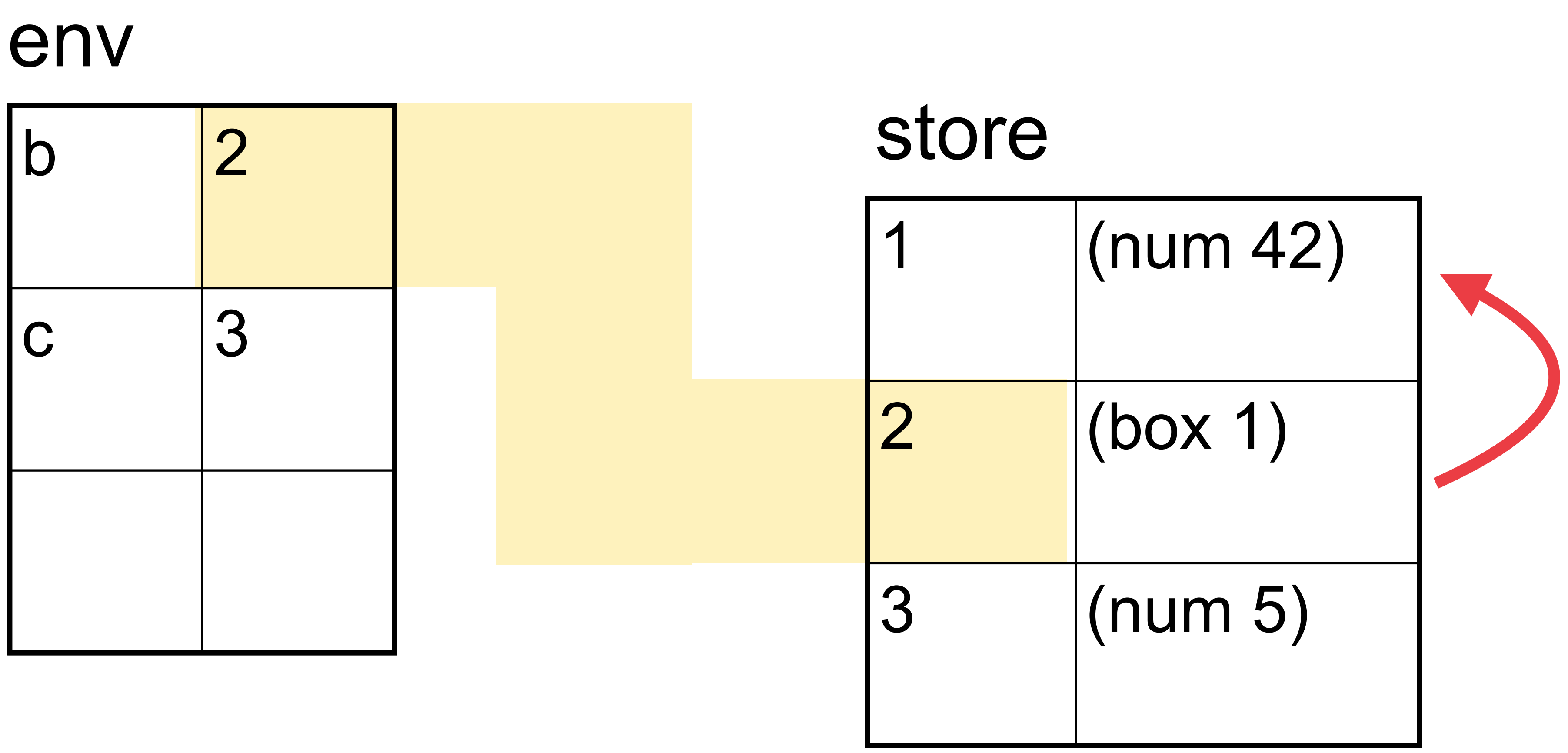
$\longrightarrow$

```
> false
  6
  5
  6
```

How can we explain this behavior?

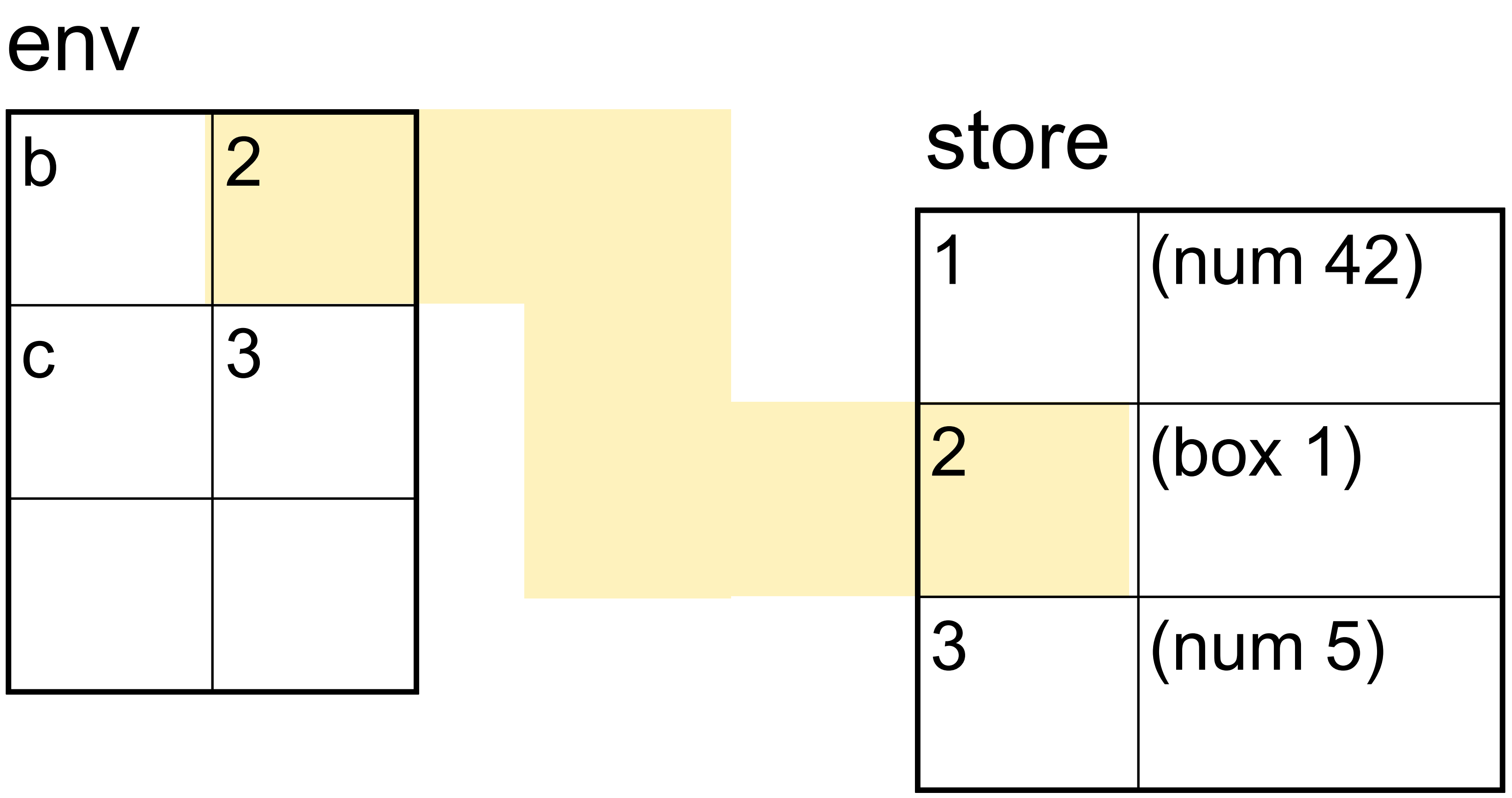
Boxed Values

- Boxes are a third kind of values.
- **A box value is just a location in the store.**
- At this location, the content of the box is stored.



Boxed Values: A Third Kind of Values

```scala
sealed abstract class Val
case class NumV(n: Int) extends Val
case class Closure(param: Symbol, body: SCFLAE, env: Env) extends Val
case class Box(location: Location) extends Val
```



env

b	2
c	3

store

1	(num 42)
2	(box 1)
3	(num 5)

Creating Mutable Boxes

- Interpret **`valueExpr`**
  - resulting in a **`value`** and potentially updated **`store`**.
- Get new location in the most recent store.
- Put value into new location and store this location in a box value.
- Return the box value and the updated store.

```scala
def interp(expr: SCFLAE, env: Env, store: Store): (Val, Store) = expr match {
  //...
  case NewBox(valueExpr) => {
    val (value, boxStore) = interp(valueExpr, env, store)
    val newLoc = nextLocation
    (Box(newLoc), boxStore + (newLoc -> value))
  }
}
```

Creating Mutable Boxes by Example

```
{let {b1 {newbox 5}}
  {let {b2 {newbox 5}}
       {... some-expr ...}
```

```
{{fun {b1} {{fun {b2} {... some-expr ...}} {newbox 5}}}

  {newbox 5}}
```

Creating Mutable Boxes by Example

```
interp {{fun {b1} {{fun {b2} ...}} {newbox 5}} [] []
interp {fun {b1} {...}} [] [] ~> (closure, [])
```

```
interp {newbox 5} [] [] ~> ((box 1),
```

1	(num 5

)

application semantics

```
interp {{fun {b2} ...} {newbox 5}}
```

env

b1	2

store

2	(box 1)
1	(num 5)

Creating Mutable Boxes by Example

`interp {fun {b2} exp }` env store `~> (closure,` store `)`

env

b1	2

store

2	(box 1)
1	(num 5)

store

2	(box 1)
1	(num 5)

`interp {newbox 5}` env store `~> ((box 3),` store `)`

env

b1	2

store

2	(box 1)
1	(num 5)

store

3	(num 5)
2	(box 1)
1	(num 5)

`interp exp` env store

application semantics

env

b2	4
b1	2

store

4	(box 3)
3	(num 5)
2	(box 1)
1	(num 5)

Opening a Box by Example

`interp (openbox b1)`

env

b2	4
b1	2

store

4	(box 3)
3	(num 5)
2	(box 1)
1	(num 5)

- The composed env and store lookup triggered by the interpretation of **`b1`** returns **`(box 1)`**.
- Return the value in the location enclosed in **`(box 1)`**, i.e., **`(num 5)`** and whatever the most recent store is.

Opening a Box Technically

- Evaluate the box expression to a box value and a potentially updated store, **`s1`**.
- Return the result of looking up **`s1`** at the location in the box value and **`s1`**.

```
def interp(expr: SCFLAE, env: Env, store: Store): (Val, Store) = expr match {
  // ...
  case OpenBox(boxExpr) => {
    val (boxV, s1) = interp(boxExpr, env, store)
    boxV match {
      case Box(loc) => (s1(loc), s1)
      case _ => sys.error("can only open boxes, but got: " + boxV)
    }
  }
}
```

Setting a Box by Example

`interp (setbox b1 6)` env store `~> (num 6)` store

env	
b2	4
b1	2

store	
4	(box 3)
3	(num 5)
2	(box 1)
1	(num 5)

store	
1	(num 6)
4	(box 3)
3	(num 5)
2	(box 1)
1	(num 5)

- The composed env and store lookup triggered by interpretation of **`b1`** returns **`(box 1)`** and leaves store unchanged.
- Interpretation of **`6`** returns **`(num 6)`** and leaves store unchanged.
- The result is **`(num 6)`** and the store extended with a binding of the location enclosed in **`(box 1)`**, to **`(num 6)`**.

Setting a Box technically

- Evaluate the box expression to a box value.
- Update the store with a new value for the location in the box.
- Return the new value stored and the updated store.

```
def interp(expr: SCFLAE, env: Env, store: Store): (Val, Store) = expr match {
  //...
  case SetBox(boxExpr, valueExpr) => {
    val (boxV, s1) = interp(boxExpr, env, store)
    val (value, s2) = interp(valueExpr, env, s1)
    boxV match {
      case Box(loc) => (value, s2 + (loc -> value))
      case _ => sys.error("can only set to boxes, but got: " + boxV)
    }
  }
}
```

Outline

- State, state in Scheme, state in our languages
- An interpretation style for modeling the effect of state
- Mutable data structures
- Variables
- Summary

Adding Variables

```
<SCFAE> ::= ...
          | {set <id> <SCFAE>}
```

Implementing Variables

- For evaluating **`set`**, the identifier is not fully evaluated.
- We just retrieve the identifier’s location from the environment.

```
def interp(expr: SCFLAE, env: Env, store: Store): (Val, Store) = expr match {
  case SetId(id, valExpr) => {
    val (value, s1) = interp(valExpr, env, store)
    val loc = env(id)
    (value, s1 + (loc -> value))
  }
}
```

L-Values and R-Values

- Variables in an assignment position (e.g., left of := or =), are only **partially resolved.**
  - These special values (the location of an identifier) are called **`l-values`**
  - Only appear on the left-hand side of assignments.

- Variables in expressions are **fully resolved** to a value (by store-lookup and env-lookup). They are called **`r-values`**.

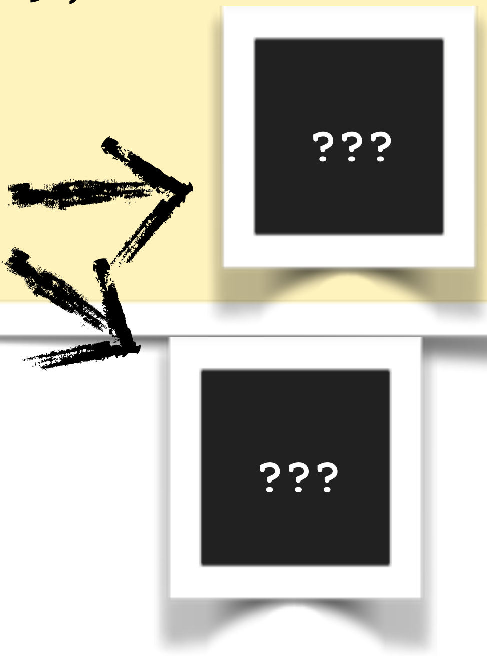
Quiz: Which variables are l-values, which are r-values?

```
x = 2;
y = x;
```

The Evaluation Pattern Illustrated

Let us investigate the evaluation of the following expression...

```
Let('switch, 0,
    Let('toggle,
      Fun('_,
        If0(
          'switch,
          Seqn(
            SetId('switch, 1),
            1),
          Seqn(
            SetId('switch, 0),
            0))),
      Add(
        App('toggle, 42),
        App('toggle, 42))))
```

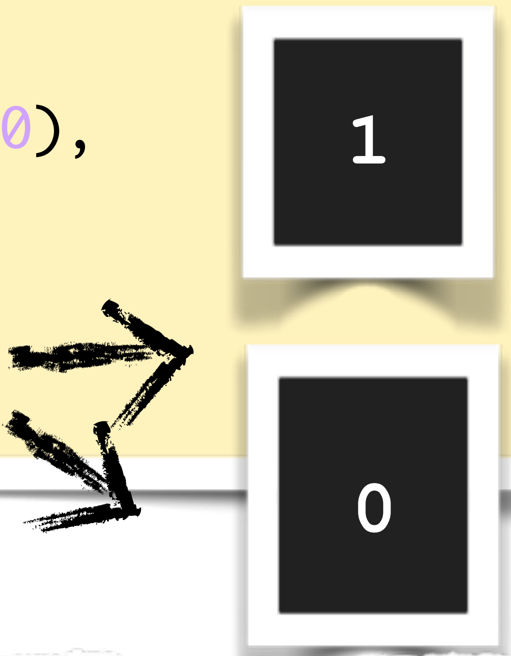


What is the result of evaluating the two operands of +

The Evaluation Pattern Illustrated

Let us investigate the evaluation of the following expression...

```
Let('switch, 0,
    Let('toggle,
      Fun('_,
        If0(
          'switch,
          Seqn(
            SetId('switch, 1),
            1),
          Seqn(
            SetId('switch, 0),
            0))),
      Add(
        App('toggle, 42),
        App('toggle, 42))))
```



The same expression reduced to different values at two different places in the program (at two different times during the execution). Why?

Variables and Function Application

- Are function application and variables independent language features?

- What does the following program evaluate to?

```
Let('v, 0,
    Let('f, Fun('x, SetId('x, 5)),
      Seqn(
        App('f, 'v),
        'v)))
```

- What does the program evaluate to with our interpreter?

Variables and Function Application

Here the answer is 0

```
Let('v, 0,
    Let('f, Fun('x, SetId('x, 5)),
      Seqn(
        App('f, 'v),
        'v)))
```

```
def interp(expr: SCFLAE, env: Env, store: Store): (Val, Store) = expr match {
   ...
  case App(funExpr, argExpr) => {
    val (funV, funStore) = interp(funExpr, env, store)
    val (argV, argStore) = interp(argExpr, env, funStore)
    funV match {
      case Closure(fParam, fBody, fEnv) => {
        val newLoc = nextLocation
        interp(fBody, fEnv + (fParam -> newLoc), argStore + (newLoc -> argV))
      }
      case _ => sys.error("can only apply functions, but got: " + funV)
    }
  }
}
```

Call-by-Value Eager Evaluation

- **`argValue`** is put into a **new store location** and the formal parameter is bound to that new location in the environment.
- **Changes to the formal parameter** do not affect the actual parameter. They **are not visible outside the function!**

Standard form of eager evaluation called "call-by-value"

```
def interp(                                                              match {
  ...
  case App(funExpr, argExpr) => {
    val (funV, funStore) = interp(funExpr, env, store)
    val (argV, argStore) = interp(argExpr, env, funStore)
    funV match {
      case Closure(fParam, fBody, fEnv) => {
        val newLoc = nextLocation
        interp(fBody, fEnv + (fParam -> newLoc), argStore + (newLoc -> argV))
      }
      case _ => sys.error("can only apply functions, but got: " + funV)
    }
  }
}
```

Call-by-Reference Eager Evaluation

- **call-by-reference**: A reference to (the location of) the actual argument (not its value) is passed when a function is called.

- Updates to the reference (location) within the called function become visible to the calling context.

Quiz: What does the following program evaluates to with call-by-reference semantics?

```
Let('v, 0,
    Let('f, Fun('x, SetId('x, 5)),
      Seqn(
        App('f, 'v),
        'v)))
```

Call-by-Reference Functions

- We add another kind of function to our language to introduce call-by-reference functions

```
<SCFLAE> ::= ...
           | {refun {<id>} <SCFLAE>}
```

- Such functions are syntactically the same as call-by-value functions except for the different introducing keyword.

Implementing Call-by-Reference

- Like a **`fun`**, a **`refun`** also evaluates to a closure.

```
def interp(expr: SCFLAE, env: Env, store: Store): (Val, Store) = expr match {
  // ...
  case ReFun(arg, body) => (RefClosure(arg, body, env), store)
}
```

- **`RefClosure`** values have the same fields as **`Closure`** values.

- Different tags are used to distinguish between the two kind of functions at application time.

Application with Call-by-Reference

```
def interp(expr: SCFLAE, env: Env, store: Store): (Val, Store) = expr match {
  // ...
  case App(funExpr, argExpr) => {
    val (funV, funStore) = interp(funExpr, env, store)
    funV match {
      // case Closure(fParam, fBody, fEnv) => ...
      case RefClosure(fParam, fBody, fEnv) => argExpr match {
        case Id(argName) => {
          interp(fBody, fEnv + (fParam -> env(argName)), funStore)
        }
        case _ => sys.error("can only call identifiers by reference")
      }
              error("can only apply functions, but got: " + funV)
    }
  }
```

assumes **`argExpr`** to be a syntactic identifier.

store is not updated

location of actual argument bound to formal parameter.

Reflecting on Call-by-Reference

- **Advantage**: less allocation to the store.

- **Disadvantage**: Creates aliases and makes reasoning about programs harder.

C allows to mark certain parameters of a multi-parameter procedure as & (other languages as ref).

Call-by-Value or Call-by-Reference

To master complexity, we need **abstraction boundaries** between units of modularity (e.g., procedures), that **limit hidden interactions** and **restrict interferences** to facilitate reasoning about their behavior in isolation.

There is agreement in modern languages that parameters should be passed by values!

Scope versus Extent

- Distinction between environment and store reveals different patterns of flow.
- Closures capture the environment at function definition, but not the store.
- The store is a global record of changes made during execution.

Scope versus Extent

- The two flows of values through the interpreter reflect the difference between names (identifiers) and values.

- Values remain in the store, even if the name that introduced them disappears from the environment.
- Not inherently a problem, since another name may have been associated with the value in the flow of computation.

- **Identifiers have lexical scope**.
- **Values** have (potentially indefinite) **dynamic extent**.

When are Values Removed from Store?

- Right now, never.

- **Garbage collection** lets languages **dissociate scope from the reclamation of space** consumed by values.
  - Can remove unreachable values.
  - Can be quite performant

Does C/C++ distinguish between lexical scope and dynamic extent?

The Problem with C

- The stack is destroyed after a frame returns. But the pointer is not set to NULL again.

```c
{
   char *dp = NULL;
   /* ... */
   {
       char c;
       dp = &c;
   }
     /* c falls out of scope */
     /* dp is now a dangling pointer */
}
```

Some languages confuse lexical scope and dynamic extent. As a result, when an identifier ceases to be in scope, they remove the value corresponding to the identifier. That value may be the result of the computation, however, and some other identifier may still have a reference to it. This premature removal of the value will inevitably lead to a system crash. Depending on how the implementation “removes” the value, however, the system may crash later instead of sooner, leading to extremely vexing bugs. This is a common problem in languages like C and C++.

Outline

- State, state in Scheme, state in our languages
- An interpretation style for modeling the effect of state
- Mutable data structures
- Variables
- Summary

Store-passing Interpretation Style

- Supporting state requires a new interpretation style to enable propagation of effects: **Store-passing interpretation**

- Store as a global medium for keeping track of dynamic value updates.

- **Static scope of variables** versus **dynamic extent of values**.
  - Environments as repositories for static scope.
  - Stores as repositories for dynamic extent.

Variables versus Mutable Data Structures

- **Location** corresponding to a **variable** is associated with a **“real” value**.

- **Location** corresponding to an identifier referring to a **mutable data structure** is associated with **a special “box value”** - denoting the **location where the mutable data is actually stored**.

Variables versus Mutable Data Structures

- `set`
  - Requires the first sub-expression to be an identifier.
  - Changes the binding of the identifier in the store.
- `set-box`
  - Allows any expression in the first sub-expression.
  - Must reduce to a “box value”.
  - The location referred to by the box value gets associated with a new “real” value.

Call-by-Value versus Call-by-Reference

- **Variables and function application are not independent language features.**

- When variables are supported, two (eager) evaluation semantics are possible: **call-by-value and call-by-reference**.

- Call-by-reference is problematic and is not supported in many languages.