Lecture 06: State

Vincenzo Ciancia

May 09, 2025

Section 1: Introduction to State

In this chapter, we extend our mini-language with the concept of **state**. Until now, our language has been purely functional, where expressions are evaluated to produce values without side effects. By adding state, we can:

1. Store and update values in memory
2. Observe changes to data over time
3. Model real-world systems that have changing state

Expressions vs. Commands

Our language extension introduces a new distinction:

- **Expressions**: Compute values (e.g., `x + 1`, `let x = 42 in x * 2`)
- **Commands**: Perform actions with side effects (e.g., `var x = 42`, `x <- 42`, `print x`)

This distinction is common in many programming languages:

```
# Expression (computes a value)
x + 1


# Command (performs an action)
var x = 42
x <- 42
```

Section 2: Commands and Command Sequences

To implement state, we introduce two new syntactic categories:

1. **Commands**: Individual actions that can modify state
2. **Command Sequences**: Ordered lists of commands to be executed in sequence

Commands in Our Language

We implement three basic command types:

1. **Variable Declaration**: Declares a new variable and initializes it

   ```
   var x = 42
   ```

2. **Assignment**: Updates a variable with a new value (only if already declared)

   ```
   x <- 10
   ```

3. **Print**: Outputs the value of an expression

   ```
   print x + 1
   ```

Command Sequences

Command sequences represent multiple commands separated by semicolons:

```
var x = 10;
var y = x + 5;
print y
```

This sequence declares `x`, then declares `y` as `x + 5` (which is 15), and finally prints the value of `y`.

Grammar Extensions

We extend our language grammar to support commands and sequences:

```
?program: command_seq


?command_seq: command
           | command ";" command_seq


?command: assign
        | print
        | vardecl


assign: IDENTIFIER "<-" expr
print: "print" expr
vardecl: "var" IDENTIFIER "=" expr
```

AST Representation

We represent commands and command sequences with these AST node types:

```
@dataclass
class Assign:
    name: str
    expr: Expression

@dataclass
class Print:
    expr: Expression

@dataclass
class VarDecl:
    name: str
    expr: Expression
```

```python
@dataclass
class CommandSequence:
    first: Command
    rest: Optional[CommandSequence] = None

type Command = Assign | Print | VarDecl
```

Section 3: The Store Model of State

To represent state, we use the "store model":

1. **Environment**: Maps variable names to locations (memory addresses)
2. **Store**: Maps locations to values

This two-level indirection allows multiple variables to refer to the same location or for a variable's value to change without changing its location.

The State as a Functional Dataclass

Our implementation uses a `State` dataclass to manage the store in a functional style:

- The store is **not** a dictionary, but a function from locations to values (or raises an error if not found), just like the environment is a function from names to locations. This ensures a fully functional approach.

```python
@dataclass
class State:
    store: Callable[[int], MVal]
    next_loc: int

def empty_store() -> Callable[[int], MVal]:
    def store_fn(loc: int) -> MVal:
        raise ValueError(f"Location {loc} not allocated")
    return store_fn

def empty_state() -> State:
    return State(store=empty_store(), next_loc=0)
```

```python
def allocate(state: State, value: MVal) -> tuple[Loc, State]:
    loc = Loc(state.next_loc)
    prev_store = state.store

    def new_store(l: int) -> MVal:
        if l == loc.address:
            return value
        return prev_store(l)

    return loc, State(store=new_store, next_loc=loc.address + 1)
```

```
def update(state: State, addr: int, value: MVal) -> State:
    prev_store = state.store

    def new_store(l: int) -> MVal:
        if l == addr:
            return value
        return prev_store(l)

    return State(store=new_store, next_loc=state.next_loc)

def access(state: State, addr: int) -> MVal:
    return state.store(addr)
```

Denotable Values (DVal)

Denotable values are those that can be bound to identifiers in an environment. Our language now has three distinct semantic domains:

- **EVal (Expressible Values)**: Values that expressions can evaluate to (integers in our language)
- **MVal (Memorizable Values)**: Values that can be stored in memory (equal to EVal in our implementation)
- **DVal (Denotable Values)**: Values that can be bound to identifiers in the environment

DVal is broader than MVal because not everything that can be bound to a name can be stored in memory. Specifically, DVal includes:

- **Numbers**: Simple integer values (EVal)
- **Operators**: Functions that take two integers and return an integer
- **Locations**: Memory addresses (wrapped in a Loc class)

The distinction between these domains is crucial for implementing state correctly:

- Expressions evaluate to EVal (integers)
- Memory cells store MVal (integers)
- Variable names can refer to DVal (integers, operators, or locations)

```python
@dataclass
class Loc:
    address: int


type DenOperator = Callable[[int, int], int]
type EVal = int  # Expressible value type (for expressions)
type MVal = EVal  # Main value type for store and evaluation (expressible)
type DVal = EVal | DenOperator | Loc  # Denotable values: can be associated with names
```

Rather than representing memory locations as plain integers, we wrap them in the `Loc` class for two important reasons:

1. **Pattern Matching**: It allows the pattern matcher to distinguish locations from integer values
2. **Type Safety**: It prevents accidentally using a location as an integer or vice versa

This distinction becomes essential in variable lookup. When we look up a variable, we need to determine if the value bound to it is:

- A direct integer value (for operators like $+$, -, etc.)
- A location that requires a further lookup in the store

Without the `Loc` wrapper class, the pattern matcher wouldn't be able to distinguish between an integer value and an integer location in memory.

Section 4: Evaluating Expressions with State

When evaluating expressions in a stateful language, we need to pass both the environment and the state:

```
def evaluate_expr(expr: Expression, env: Environment, state: State) -> EVal:
    # ...
```

Variable lookup now has two steps:

1. Use the environment to find the location or directly bound value
2. If it's a location, use the state to look up the value at that location

```
case Var(name):
    try:
        dval = lookup(env, name)
        match dval:
            case int():
                return dval
            case Loc(address=addr):
                return access(state, addr)
            case _:
                raise ValueError(f"Variable '{name}' does not refer to a value")
    except ValueError as e:
        raise ValueError(f"Variable error: {e}")
```

Section 5: Executing Commands

Commands modify the state or produce output. The `execute_command` function returns both the updated environment and state as a tuple:

```
def execute_command(
    cmd: Command, env: Environment, state: State
) -> tuple[Environment, State]:
    # ...
```

The Variable Declaration Command

The variable declaration command (`var x = expr`) creates a new variable and initializes it:

```
case VarDecl(name, expr):

    value = evaluate_expr(expr, env, state)

    loc, state = allocate(state, value)

    new_env = bind(env, name, loc)

    return new_env, state
```

The Assignment Command

The assignment command (<-) only updates an existing variable:

```
case Assign(name, expr):
    try:
        dval = lookup(env, name)
        match dval:
            case Loc(address=addr) as loc:
                value = evaluate_expr(expr, env, state)
                state1 = update(state, addr, value)
                return env, state1
            case _:
                raise ValueError(
                    f"Assignment target '{name}' is not a variable"
                )
    except ValueError:
        raise ValueError(f"Assignment to undeclared variable '{name}'")
```

The Print Command

The print command evaluates the expression and prints its value:

```
case Print(expr):
    # MORALLY THIS IS THE IDENTITY FUNCTION
    value = evaluate_expr(expr, env, state)
    print(value)
    return env, state
```

Command Sequences

Executing a command sequence involves:

1. Executing the first command
2. Executing the rest of the sequence with the updated environment and state

```
def execute_command_seq(
    seq: CommandSequence, env: Environment, state: State
) -> tuple[Environment, State]:
    # Execute the first command
    env1, state1 = execute_command(seq.first, env, state)

    # If there are more commands, execute them with the updated environment and state
    if seq.rest:
        return execute_command_seq(seq.rest, env1, state1)

    return env1, state1
```

Section 6: Examples of State in Action

Let's look at some examples that demonstrate the power of state:

Example 1: Basic Declaration, Assignment, and Printing

```
var x = 42; print x
```

This simple example shows creating a variable and reading its value. The output is `42`.

Example 2: Updating Variables

```
var x = 10; print x; x <- 20; print x
```

This example demonstrates updating a variable's value. The output is:

```
10
20
```

Example 3: Multiple Declarations and Operations

```
var x = 10; var y = 20; var z = x + y; print z; x <- 30; print x + y
```

The output of this program is:

```
30
50
```

Notice that changing `x` doesn't automatically update `z`, even though `z` was defined in terms of `x` and `y`.

Example 4: Let Expressions in Commands

```
var x = let y = 5 in y * 2; print x
```

This example shows how we can use let expressions within commands. The output is `10`.

Section 7: Differences from Previous Chapters

Adding state represents a significant shift in our language:

Purely Functional (Ch. 5)	Stateful (Ch. 6)
Values are immutable	Values can change over time
No side effects	Commands have side effects
Referential transparency	Expressions may evaluate differently
Environment maps names to values	Environment maps names to locations
Easier to reason about	More expressive

Section 8: Conclusion and Next Steps

Adding state to our mini-language significantly increases its expressiveness, making it capable of modeling real-world problems that involve change over time. In the next chapter, we'll build upon this foundation by adding control flow structures like loops and conditionals.

With state, environment, and control flow, our language will have all the essential ingredients of a complete programming language.

Exercises

1) Interactive REPL

Modify the REPL implementation to operate one command at a time instead of parsing and executing entire programs.

2) Aliasing

Add the command

```
alias x = y
```

where `x` and `y` are variables. The semantics is that after executing the command, both names x and y point to the same location, so assigning to one of them assigns also to the other one.

Make an example where this becomes apparent

Now consider the program

```
var x = 0;
alias y = x;
x <- x+1;
```

5) If-Then-Else Statements

Add if-then-else statements to the language:

```
if x > 0 then
    y <- 1
else
    y <- -1
```

5) Command Sequences in Branches

Extend if-then-else to support command sequences in the branches and consider the implications for variable scoping.