Lecture 06: State

Vincenzo Ciancia

May 02, 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:

TODO: also use an assignemnt in the example

```
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: vardecl
        | assign
        | print


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

AST Representation

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

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

@dataclass
class Assign:
    name: str
    expr: Expression

@dataclass
class Print:
    expr: Expression
```

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


type Command = VarDecl | Assign | Print
```

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(frozen=True)
class State:
    store: Callable[[int], Any]
    next_loc: int


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


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

```python
def alloc(state: State, value: Any) -> tuple[int, State]:
    loc = state.next_loc
    prev_store = state.store
    def new_store(l: int) -> Any:
        if l == loc:
            return value
        return prev_store(l)
    return loc, State(store=new_store, next_loc=loc + 1)
```

```
def update(state: State, loc: int, value: Any) -> State:
    state.store(loc)  # will raise if not found
    prev_store = state.store
    def new_store(l: int) -> Any:
        if l == loc:
            return value
        return prev_store(l)
    return State(store=new_store, next_loc=state.next_loc)

def lookup(state: State, loc: int) -> Any:
    return state.store(loc)
```

Denotable Values (DVal)

Denotable values are those that can be bound to identifiers in an environment. In our implementation, they include:

DVal includes:

- **Numbers**: Simple integer values
- **Operators**: Functions that take two integers and return an integer
- **Locations**: Memory addresses (as int, but semantically distinct)
- **Booleans**: For forward compatibility with later chapters

```python
# Python 3.13 union syntax


type Num = int  # A type alias for integers

type DenOperator = Callable[[int, int], int]

type Location = int


type DVal = int | DenOperator | bool | Location  # Denotable values: everything that could be bound in Lecture 5, plus
```

Note: *DVal is broader than what can be stored in memory (MVal). Not everything that can be bound to a name can be stored 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) -> int:
    # ...
```

Variable lookup now has two steps:

1. Use the environment to find the location
2. Use the state to look up the value at that location

```
case Var(name):
    # Look up variable's location and then its value in the state
    try:
        loc = lookup_env(env, name)
        return lookup(state, loc)
    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:

```
env, state = execute_command(cmd, env, state)
```

- `env` is the updated environment (with new variable bindings, if any)
- `state` is the updated state (with new or updated values)

You can access the first and second components of the tuple as `env` and `state` respectively.

Note: *The `print` command is not part of the mathematical semantics; it is just an aid for using the interpreter.*

The Variable Declaration Command

The variable declaration command (`var x = expr`) creates a new variable and initializes it. If the variable already exists, it is an error.

The Assignment Command

The assignment command (`<-`) only updates an existing variable. If the variable does not exist, it is an error.

The Print Command

The print command evaluates the expression and prints its value. It does not affect the environment or 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]:
    env, state = execute_command(seq.first, env, state)
    if seq.rest:
        return execute_command_seq(seq.rest, env, state)
    return env, state
```

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;

var y = x + 5;

print y;

x <- 20;

print x;

print y
```

This example demonstrates that changing `x` doesn't automatically change `y`, even though `y` was defined in terms of `x`. The output is:

```
15

20

15
```

The value of `y` remains 15 because the expression `x + 5` was evaluated at the time of declaration.

Example 3: 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: Extending State

1. Alias Command

Exercise: Implement an `alias x = y` command that makes `x` and `y` point to the same location in the environment (i.e., after `alias x = y`, both names refer to the same memory cell).

Example 1:

```
var a = 10;
alias b = a;
b <- 20;
print a  # Output: 20
```

Example 2:

```
var x = 5;
alias y = x;
print y  # Output: 5
x <- 42;
print y  # Output: 42
```

2. Conditional Command

Exercise: Implement an `if condition then command1 else command2` construct, where `condition` is an expression and `command1`/`command2` are single commands (not sequences).

Prompt:

- What if either `command1` or `command2` is a `var` declaration? Can this be used to conditionally declare variables?
- What should happen if the same variable is declared in both branches? Should the environment be merged, or should this be an error?
- How would you design the semantics to handle these cases?

Reflect on these questions and try to implement a solution that is both safe and intuitive.