Lecture 06: State

Vincenzo Ciancia

May 01, 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., `x <- 42`, `print x`)

This distinction is common in many programming languages:

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

# Command (performs an action)
x <- x + 1
```

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 two basic command types:

1. **Assignment**: Binds or updates a variable with a new value

   ```
   x <- 42
   ```

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

   ```
   print x + 1
   ```

Command Sequences

Command sequences represent multiple commands separated by semicolons:

```
x <- 10;

y <- x + 5;

print y
```

This sequence assigns 10 to x, then assigns $x + 5$ (which is 15) to y, 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


assign: IDENTIFIER "<-" expr

print: "print" expr
```

AST Representation

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

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


@dataclass
class Print:
    expr: Expression


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


type Command = 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 Class

Our implementation uses a `State` class to manage the store:

```python
class State:
    def __init__(self):
        self.store: Dict[int, Any] = {}
        self.next_loc: int = 0

    def alloc(self, value: Any) -> int:
        """Allocate a new location in the store for the value."""
        loc = self.next_loc
        self.store[loc] = value
        self.next_loc += 1
        return loc

    def update(self, loc: int, value: Any) -> None:
        """Update the value at a location."""
        if loc not in self.store:
            raise ValueError(f"Location {loc} not allocated")
```

Environment and Binding

Our environment maps names to locations:

```python
# Environment now maps to locations (store addresses)
type Environment = Callable[[str], int]


def empty_environment() -> Environment:
    """Create an empty environment function"""
    def env(name: str) -> int:
        raise ValueError(f"Undefined identifier: {name}")
    return env


def bind(env: Environment, name: str, loc: int) -> Environment:
    """Create new environment with an added binding to a location"""
    def new_env(n: str) -> int:
        if n == name:
            return loc
        return env(n)
    return new_env
```

Section 4: Evaluating Expressions with State

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

```python
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

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

```python
def execute_command(cmd: Command, env: Environment, state: State) -> tuple[Environment, State]:
    """Execute a command, returning the updated environment and state"""
    match cmd:
        case Assign(name, expr):
            # Evaluate the expression
            value = evaluate_expr(expr, env, state)

            try:
                # If variable exists, update its value in the state
                loc = lookup_env(env, name)
                state.update(loc, value)
                return env, state
            except ValueError:
                # If variable doesn't exist, allocate a new location
```

The Assignment Command

The assignment command (`<-`) has two behaviors:

1. If the variable exists, update its value in the store
2. If the variable is new, allocate a new location and create a binding

This is similar to how variable assignment works in most programming languages.

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

```python
def execute_command_seq(seq: CommandSequence, env: Environment, state: State) -> tuple[Environment, State]:
    """Execute a command sequence, returning the final environment and state"""
    # Execute the first command
    env, state = 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, 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 Assignment and Printing

```
x <- 42;
```

```
print x
```

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

Example 2: Updating Variables

```
x <- 10;

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 assignment.

Example 3: Let Expressions in Commands

```
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, closer to real languages

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 more complex conditionals.

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