Lecture 04: Domains

Vincenzo Ciancia

May 01, 2025

Section 1: Introduction to Semantic Domains

In programming language semantics, **semantic domains** are mathematical structures used to give meaning to syntactic constructs. They provide the foundation for defining the behavior of programs in a precise, mathematical way.

Key Semantic Domains in Programming Languages

- **Syntactic Domains**: Represent the structure of programs (tokens, parse trees, syntax trees).
- **Semantic Domains**: Represent the meaning of programs (values, environments, states).

The relationship between these domains is at the heart of language semantics:

```
Syntax -> Semantics
```

```
Program -> Meaning
```

Core Semantic Domains

In our mini-interpreter, we'll work with several fundamental semantic domains:

- **Expressible Values**: Values that can be produced by evaluating expressions
- **Denotable Values (DVal)**: Values that can be bound to identifiers in the environment
- **Memorizable Values (MVal)**: Values that can be stored in memory/state
- **Identifiers**: Names of constants, variables, functions, modules, etc.
- **Locations**: Memory addresses of variables
- **Environment**: Maps identifiers to denotable values, representing the binding context
- **State**: Maps memory locations to memorizable values, representing the program's memory

While these domains often overlap, they aren't necessarily identical. Understanding the differences is crucial for language design.

Side Effects and Pure Functions

A fundamental distinction in programming language semantics is between **pure** computations and those that produce **side effects**.

Pure Functions

A **pure function** is a computation that: 1. Always produces the same output for the same input 2. Has no observable effects beyond computing its result

In mathematical terms, a pure function is simply a mapping from inputs to outputs, like mathematical functions (e.g., sin(x), log(x)).

```
def add(x: int, y: int) -> int:
    return x + y  # Pure: same inputs always give same output
```

Side Effects

A **side effect** is any observable change to the system state that occurs during computation, beyond returning a value. Common side effects include:

1. **Memory updates**: Modifying variables or data structures

```
x = 10  # Changes the program's state
```

   Q: in this context, what is a function that does **not** return always the same value for the same inputs?

2. **Input/Output operations**:

```
print("Hello")  # Affects the external world (terminal)
```

3. **File operations**:

```
with open("data.txt", "w") as f:
    f.write("data")  # Changes the file system
```

4. **Network operations**:

```
requests.get("https://example.com")  # Interacts with external systems
```

Memory Updates as Side Effects

In our interpreter design, memory (state) updates are a primary form of side effect. When we update state:

```
def state_update(state: State, location: int, value: MVal) -> State:
    new_state = state.copy()
    new_state[location] = value
    return new_state
```

We're representing a change to the program's memory. In a real computer, this would modify memory cells directly. Our functional implementation returns a new state rather than modifying the existing one, but conceptually it represents the same side effect.

Side Effects in Programming Languages

Languages differ in how they handle side effects:

- **Purely functional languages** (like Haskell) isolate side effects using type systems and monads
- **Imperative languages** (like C, Python) embrace side effects as their primary mechanism for computation
- **Hybrid languages** (like Scala, OCaml) support both styles

Understanding side effects is crucial for language design because they impact: - Program correctness (pure functions are easier to reason about) - Parallelization (side effects complicate parallel execution) - Optimization (pure functions allow more aggressive optimizations)

In our interpreter, we'll model side effects using explicit state passing, maintaining the mathematical clarity of our semantics while accurately representing the behavior of stateful programs.

Section 2: Denotable vs. Memorizable Values

Denotable Values (DVal)

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

```
type Num = int  # A type alias for integers
type DenOperator = Callable[[int, int], int]
type DVal = int | DenOperator  # Denotable values
```

DVal includes: - **Numbers**: Simple integer values - **Operators**: Functions that take two integers and return an integer

Memorizable Values (MVal)

Memorizable values are those that can be stored in memory (the state). In our implementation:

```
type MVal = int  # Memorizable values
```

MVal only includes integers, not functions. This highlights an important distinction:

> *Not everything that can be bound to a name can be stored in memory.*

This distinction is crucial for understanding: - Why some languages don't support first-class functions - Why some types require special treatment in memory management - How languages with different type systems handle values differently

Section 3: Environment and State as Functions

In our treatment of semantic domains, we adopt a purely functional approach where environments are represented as functions, and states are represented as dataclasses containing store functions, rather than mutable data structures. This approach aligns with the mathematical view of semantic domains and provides a clean conceptual model for understanding program behavior.

Functional Programming: A Brief Digression

Before diving into our function-based implementation of environments and state, it's worth taking a brief detour to discuss functional programming concepts, as they form the foundation of our approach.

Functional programming is a paradigm where computations are treated as evaluations of mathematical functions, emphasizing immutable data and avoiding side effects. Python, while not a pure functional language, supports many functional programming techniques.

Functions as First-Class Citizens

In functional programming, functions are “first-class citizens” — they can be: - Assigned to variables - Passed as arguments to other functions - Returned from functions - Stored in data structures

For example:

```
# Function assigned to a variable
increment = lambda x: x + 1


# Function passed as an argument
def apply_twice(f, x):
    return f(f(x))


result = apply_twice(increment, 3)  # Returns 5
```

Higher-Order Functions: Map

A common pattern in functional programming is applying a function to each element in a collection. Python's `map` function does exactly this:

```
numbers = [1, 2, 3, 4, 5]


# Apply a function to each element
squared = list(map(lambda x: x**2, numbers))  # [1, 4, 9, 16, 25]


# Equivalent to a list comprehension
squared_alt = [x**2 for x in numbers]  # [1, 4, 9, 16, 25]
```

Function Composition

Functional programming emphasizes building complex behaviors by composing simpler functions:

```
def compose(f, g):
    return lambda x: f(g(x))

# Compose two functions
negate_and_square = compose(lambda x: -x, lambda x: x**2)
result = negate_and_square(5)  # -25
```

Pure Functions and Immutability

Pure functions always produce the same output for the same input and have no side effects. This property makes them predictable and easier to reason about:

```
# Pure function
def add(a, b):
    return a + b

# Impure function (has side effects)
def add_and_print(a, b):
    result = a + b
    print(f"The result is {result}")  # Side effect: printing
    return result
```

Relevance to Semantic Domains

These functional programming concepts directly inform our approach to implementing semantic domains:

1. We represent environments and stores as functions, not data structures
2. We use higher-order functions to create updated environments and stores
3. We maintain immutability through functional updates rather than mutations
4. We compose simple operations to build complex behaviors

With this foundation in mind, let's explore how we represent environments and states as functions.

Environment as a Function

An environment is mathematically a function that maps identifiers to denotable values:

```
type Environment = Callable[[str], DVal]
```

This means an environment is a function that: - Takes an identifier (string) as input - Returns a denotable value (DVal) - Raises an error if the identifier is not defined

While many practical implementations use dictionaries or hash tables for efficiency, conceptually an environment is simply a function:

```
Environment: Identifier → DVal
```

State as a Dataclass

Unlike environment, which is purely a function, a state encapsulates both a store function and allocation information:

```
@dataclass
class State:
    store: Callable[[Location], MVal]  # The store function
    next_loc: int                      # Next available location
```

This represents a state as: - A dataclass containing a store function and next location counter - The store function takes a location as input and returns the value at that location - The next_loc field tracks the next available memory location for allocation

Conceptually, the store component maintains the mapping:

```
Store: Location → MVal
```

While the next_loc component tracks allocation state.

Section 4: Functional Updates

In a purely functional approach, we don't mutate existing environments or states. Instead, we create new functions or dataclasses that encapsulate the updated behavior.

Environment Updates

Instead of modifying a dictionary, we define a new function that returns the new value for the updated identifier and delegates to the original environment for all other identifiers:

```
def bind(env: Environment, name: str, value: DVal) -> Environment:
    """Create new environment with an added binding"""
    def new_env(n: str) -> DVal:
        if n == name:
            return value
        return env(n)
    return new_env
```

This function returns a new environment that: - Returns `value` when asked for `name` - Delegates to the original environment for all other identifiers

This approach: - Preserves referential transparency - Enables easy implementation of lexical scoping - Facilitates reasoning about program behavior - Models the mathematical concept of function extension

State Updates

Similarly, state updates create new State objects with updated store functions:

```
def state_update(state: State, location: Location, value: MVal) -> State:
    """Create new state with an updated value at given location"""
    def new_store(loc: Location) -> MVal:
        if loc == location:
            return value
        return state.store(loc)
    return State(store=new_store, next_loc=state.next_loc)
```

This function creates a new State object that: - Contains a new store function that returns the new value when asked for the specified location - Delegates to the original state's store function for all other locations - Preserves the next_loc value from the original state

Empty Environment and State

The primitives for creating empty environments and states define initial values:

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


def empty_memory() -> State:
    """Create an empty memory state"""
    def store(location: Location) -> MVal:
        raise ValueError(f"Undefined memory location: {location}")
    return State(store=store, next_loc=0)
```

Note that empty_memory returns a State dataclass initialized with an empty store function and next_loc set to 0.

Memory Allocation

In a complete interpreter, we use the State dataclass to implement memory allocation elegantly:

```
def allocate(state: State, value: MVal) -> tuple[State, Location]:
    """Allocate a new memory location and store a value there.
    Returns the updated state and the new location."""
    location = state.next_loc
    new_state = state_update(state, location, value)
    # Return state with incremented next_loc and the allocated location
    return State(store=new_state.store, next_loc=location + 1), location
```

This function: 1. Gets the next available location from the state 2. Updates the store function to map this location to the provided value 3. Returns a new state with the incremented next_loc and the allocated location

By bundling the store function with the next_loc counter in our State dataclass, we maintain a purely functional approach while elegantly handling the allocation challenge. This design demonstrates how functional programming can manage state without side effects by making state changes explicit in the return values of functions.

Initial Environment Setup

In our functional implementation, the initial environment is built by starting with an empty environment and extending it with each operator:

```
def create_initial_env() -> Environment:
    """Create an environment populated with standard operators"""
    env = empty_environment()
    env = bind(env, "+", add)
    env = bind(env, "-", subtract)
    env = bind(env, "*", multiply)
    env = bind(env, "/", divide)
    env = bind(env, "%", modulo)
    return env
```

This builds up the environment incrementally, adding each binding through functional extension.

Section 5: Environment-Based Interpretation

Traditional interpreters often use pattern matching on operators directly in the evaluation function. An environment-based approach takes a more abstract view, treating operators as first-class values in the environment.

Traditional Approach (from Chapter 3)

```
def evaluate(ast: Expression) -> int:
    match ast:
        case Number(value):
            return value
        case BinaryExpression(op, left, right):
            left_value = evaluate(left)
            right_value = evaluate(right)
            match op:
                case "+":
                    return left_value + right_value
                case "-":
                    return left_value - right_value
                # ... other operations
```

Environment-Based Approach

```
def evaluate(ast: Expression, env: Environment) -> MVal:
    match ast:
        case Number(value):
            return value
        case BinaryExpression(op, left, right):
            try:
                # Get operator from environment
                operator = lookup(env, op)

                # Ensure it's a DenOperator
                if not isinstance(operator, Callable):
                    raise ValueError(f"{op} is not a function")

                # Evaluate operands and apply operator
                left_value = evaluate(left, env)
                right_value = evaluate(right, env)
```

Benefits of the Environment-Based Approach

- **Extensibility**: New operators can be added to the environment without modifying the evaluator
- **First-class operations**: Operators are values that can be passed, returned, and manipulated
- **Consistent treatment**: Operators and other identifiers are handled uniformly
- **Semantic clarity**: The environment explicitly represents the mapping from names to meanings

Section 6: Implementing an Environment-Based Interpreter

Our `domains.py` file implements a complete environment-based interpreter:

1. **Define semantic domains**: Denotable and memorizable values
2. **Implement operators**: Define functions for arithmetic operations
3. **Create the environment**: Populate with standard operators
4. **Evaluate expressions**: Using the environment to look up operators
5. **REPL**: Interactive environment for testing the interpreter

Section 7: Extending the Interpreter

This approach makes it easy to extend the language with new features:

Adding New Operators

To add a new operator, simply define its function and add it to the environment:

```
def power(x: int, y: int) -> int:
    return x ** y


# Extend environment
env = bind(create_initial_env(), "**", power)
```

```
### Adding Variables

To support variables, extend the AST with a variable node and update the evaluator:

```python
@dataclass
class Variable:
    name: str

# Update Expression type
type Expression = Number | BinaryExpression | Variable
```
```

Update evaluate function

```
def evaluate(ast: Expression, env: Environment) -> MVal:
    match ast:
        case Variable(name):
            value = lookup(env, name)
            # Additional check might be needed if variables can only be Numbers
            if not isinstance(value, int):
                raise ValueError(f"{name} is not a number")
            return value
        # ... existing cases
```

Additional Resources

- Denotational Semantics (Wikipedia)
- Programming Language Semantics (Stanford)
- Functional Programming and Lambda Calculus
- Environment and Store in Programming Languages

Section 9: Practical Implementation and Testing

Our environment-based interpreter includes practical components for execution and testing, demonstrating how theoretical concepts translate to code.

Parsing and AST Construction

In our implementation, we use the Lark parser to convert text expressions into parse trees:

```
def parse_ast(expression: str) -> Expression:
    """Parse a string expression into an AST"""
    parse_tree = parser.parse(expression)
    return transform_parse_tree(parse_tree)
```

The `transform_parse_tree` function then converts these parse trees into our AST representation, ready for evaluation.

Interactive REPL

A Read-Eval-Print Loop allows interactive testing of the interpreter:

```
def REPL():
    """Read-Evaluate-Print Loop with environment"""
    env = create_initial_env()

    print("Mini-interpreter with environment (type 'exit' to quit)")
    print("Available operators: +, -, *, /, %")

    while True:
        expression = input("Enter an expression (exit to quit): ")
        if expression == "exit":
            break

        try:
            ast = parse_ast(expression)
            result = evaluate(ast, env)
            print(result)
```

Automated Tests

Testing ensures the interpreter behaves as expected across various expressions:

```
def run_tests():
    """Run test expressions to verify the parser and evaluator"""
    test_expressions = [
        "1+2",
        "3*4",
        "5-3",
        "10/2",
        "10%3",
        "(1+2)*3",
        "1+(2*3)",
        "10/(2+3)",
        "10%(2+3)",
    ]

    env = create_initial_env()
```

Section 10: Conclusion and Next Steps

Summary of Key Concepts

In this chapter, we've explored:

1. **Semantic domains** as mathematical structures that give meaning to programs
2. **Environment-based interpretation** as a flexible approach to language implementation
3. **Functional representations** of environments and state
4. **Primitive operations** for manipulating environments and memory
5. **The distinction** between denotable and memorizable values

These concepts form the foundation for understanding more complex language features in subsequent chapters.

Looking Forward

The environment-based approach introduced here will be extended in future chapters to support:

1. **Variables and assignment**: Using environments to bind identifiers to memory locations
2. **Scoping mechanisms**: Creating nested environments for block-structured code
3. **Functions and closures**: Capturing environments for later execution
4. **Typing systems**: Adding constraints on what values expressions can produce

By building on these semantic foundations, we can construct a rich understanding of programming language design and implementation.

Exercises for the Reader

1. Extend the interpreter to support variables using the Variable AST node described in Section 7
2. Add support for multi-character operators (e.g., "**" for exponentiation)
3. Implement a memory system with the primitives described in Section 8
4. Add support for conditional expressions (if-then-else)
5. Implement a simple block structure with local variables

These exercises will deepen your understanding of language semantics and interpreter design while preparing you for more advanced topics in the next chapters.