CSC2/455 Software Analysis and Improvement

Abstract Interpretation - II

Sreepathi Pai
April 7, 2021

URCS
Outline

Introduction

A Tiny Language and Its Semantics

To be continued ...
Introduction

A Tiny Language and Its Semantics

To be continued ...
We learnt about program analysis tools beyond iterative dataflow analysis.

Abstract Interpretation

- Maps concrete states of programs to abstract states.
- Abstract states belong to an abstract domain: signs, intervals, convex polyhedra, ...
- Define transfer functions to convert pre-condition (input) states to post-condition (output) states.
- Union for alternate paths.
- Widen for loops.

This lecture:

- Concrete Semantics for a small language.
A note on the presentation

- This lecture defines a number of formal concepts and is notation-heavy.
- I also provide an equivalent formal notation in (Python) code to hopefully make it easier.
Outline

Introduction

A Tiny Language and Its Semantics

To be continued ...
A Tiny Language: Grammar

\[ n \in \mathbb{V} \]
\[ x \in \mathbb{X} \]
\[ \circ ::= + | - | * | \ldots \]
\[ \otimes ::= < | \leq | > | == | \ldots \]

- \( n \) is a set of concrete values, here we shall treat \( \mathbb{V} = \mathbb{Z} \)
  - All values are integers
- \( x \) is the name of a variable. The set \( \mathbb{X} \) contains all variable names.
- \( \circ \) represents arithmetic binary operators
- \( \otimes \) represents boolean binary operators
A Tiny Language: Expressions

\[ E ::= n \mid x \mid E \odot E \]
\[ B ::= x \otimes n \]

- An arithmetic expression \( E \) is:
  - a number, or
  - a variable name,
  - or a binary expression

- A boolean expression \( B \) is:
  - a variable,
  - a boolean operator
  - a constant \( n \)
from typing import Union
from typing_extensions import Literal

BinaryOps = Literal['+', '-', '*', '/']
ComparisonOps = Literal['<', '>', '==', '<=', '>=', '!=']

Scalar = int # restrict Scalars to ints in this implementation

class Node(object):
    pass

class Var(Node):
    def __init__(self, name: str):
        self.name = name

    def __str__(self):
        return self.name

Expr = Union[Scalar, Var, 'BinOp']

- This is Python 3 augmented with types
  - Union stands for a union type
Nothing special here, each component of the grammar is stored in the respective AST nodes.

I'm eliding implementations of __str__, indicated by '...'
Commands in the language

\[
C ::= \\
skip \\
| C; C \\
| x := E \\
| input(x) \\
| if(B){C} else {C} \\
| while(B){C}
\]

\[
P ::= C
\]
class Cmd(Node):
    pass

class Skip(Cmd):
    def __init__(self):
        pass

class Seq(Cmd):
    def __init__(self, cmd0: Cmd, cmd1: Cmd):
        self.cmd0 = cmd0
        self.cmd1 = cmd1

class Assign(Cmd):
    def __init__(self, left: Var, right: Expr):
        self.left = left
        self.right = right

class Input(Cmd):
    def __init__(self, var: Var):
        self.var = var

    def __str__(self):
        return f"input({self.var})"

class IfThenElse(Cmd):
    def __init__(self, cond: BoolExpr, then_: Cmd, else_: Cmd):
        self.cond = cond
        self.then_ = then_
        self.else_ = else_

class While(Cmd):
    def __init__(self, cond: BoolExpr, body: Cmd):
        self.cond = cond
        self.body = body

class Program(Node):
    def __init__(self, cmd: Cmd):
        self.program = cmd
if(x > 7) {
    y := (x - 7)
} else {
    y := (7 - x)
}

can be represented using the AST as:

```plaintext
x = Var('x')
y = Var('y')

t = Program(IfThenElse(BoolExpr('>', x, 7),
                        Assign(y, BinOp('-', x, 7)),
                        Assign(y, BinOp('-', 7, x))
                    )
```
To execute programs represented as ASTs, we need the following:

- **Storage/Memory**: to track values of variables
- **Semantics**: to express what each command does, usually mathematical
  - Denotational semantics ("input/output" semantics)
  - Operational semantics
  - Axiomatic semantics
  - and many others...
\[ M = X \rightarrow V \]

- A store (from storage) is a map/function from variables to values.
- We’ll represent it as (assuming \( X = \{x, y\} \)):
  \[ m = \{x \rightarrow 3, y \rightarrow 4\} \]
- Store (or memory) \( m \) maps \( x \) to 3 and \( y \) to 4.
- So, \( m(x) = 3 \), and \( m(y) = 4 \)
from typing import Dict, List

# using str instead of Var, with Var.name as the key.
# This is accidental.
Memory = Dict[str, int]

x = Var('x')
y = Var('y')

m = {x.name: 3, y.name: 4}

print(m[x.name])
print(m[y.name])
The semantics of an expression \( E \) depend on the memory store \( m \).

We use \( \llbracket E \rrbracket(m) \) to denote its semantics.

We’ll define \( \llbracket E \rrbracket(m) \) over its grammar as:

\[
\begin{align*}
\llbracket n \rrbracket(m) &= n \\
\llbracket x \rrbracket(m) &= m(x) \\
\llbracket E_0 \circ E_1 \rrbracket(m) &= f_\circ(\llbracket E_0 \rrbracket(m), \llbracket E_1 \rrbracket(m))
\end{align*}
\]

Here \( f_\circ \) is the function that implements \( \circ \), for example:

- \( f_+(a, b) = a + b \)
def f_binop(op: BinaryOps, left: Scalar, right: Scalar) -> Scalar:
    if op == '+':
        return left + right
    elif op == '-':
        return left - right
    elif op == '*':
        return left * right
    elif op == '/':
        return left // right
    else:
        raise NotImplementedError(f"Unknown operator: {op}")

def evaluate_Expr(E: Expr, m: Memory) -> Scalar:
    if isinstance(E, Scalar):
        return E
    elif isinstance(E, Var):
        return m[E.name]
    elif isinstance(E, BinOp):
        return f_binop(E.op,
                       evaluate_Expr(E.left, m),
                       evaluate_Expr(E.right, m))
Let $\mathbb{B}$ be the set \{true, false\}

The semantics of a boolean expression is then $[B] : M \rightarrow \mathbb{B}$

$$[x \otimes n](m) = f_{\otimes}(m(x), n)$$

which can be expressed in Python as:

```python
def f_cmpop(op: ComparisonOps, left: Scalar, right: Scalar) -> bool:
    if op == '<':
        return left < right
    elif op == '>':
        return left > right
    ...

def evaluate_BoolExpr(B: BoolExpr, m: Memory) -> bool:
    return f_cmpop(B.op, m[B.left.name], B.right)
```
Semantics of other commands

- Both $[E]$ and $[B]$ are building blocks for the semantics of other commands.

- While they were defined on a single memory store $m$, we’re going to define the semantics for commands on a set of memory states $M$.
  - So, $m \in M$, and $M \in \mathcal{P}(\mathbb{M})$.
  - Where $\mathcal{P}(\mathbb{M})$ denotes the powerset of memory stores.

- This way, our semantics for commands $[\cdot]_{\mathcal{P}}$ will convert a set of input states to a set of output states.
The notation $m[x \mapsto n]$ is a memory update, it creates a new store identical to $m$ except that $x$ is updated to $n$
```python
def evaluate_Cmd(C: Cmd, M: List[Memory]) -> List[Memory]:
    def update_memories(var, value_lambda):
        out = []
        for m in M:
            m_out = dict(m)
            m_out[var] = value_lambda(m)
            out.append(m_out)

        return out

    if isinstance(C, Skip):
        return M
    elif isinstance(C, Program):
        return evaluate_Cmd(C.program, M)
    elif isinstance(C, Assign):
        return update_memories(C.left.name,
            lambda m: evaluate_Expr(C.right, m))
    elif isinstance(C, Input):
        n = random.randint(0, 100) # could be anything, actually
        return update_memories(C.var.name, lambda _: n)
    elif isinstance(C, Seq):
        return evaluate_Cmd(C.cmd1, evaluate_Cmd(C.cmd0, M))
    ...

    I’ve chosen $M$ to be a list of memories (recall Memory is a `Dict[str, int]`)
```
Example of using `evaluate_Cmd`:

```python
x = Var('x')
y = Var('y')

m1 = {x.name: 3, y.name: 4}
m2 = {x.name: 5, y.name: 6}

M_in = [m1, m2]

M_out = evaluate_Cmd(Assign(x, 7), M_in)

# M_out = [{'x': 7, 'y': 4}, {'x': 7, 'y': 3}]
```
\[ \text{if}(B)\{C_0\} \text{ else } \{C_1\}\] \(F(M) = ? \)

- \(C_0\) (the code executing when \(B\) is true) must only operate on \(m \in M\) where \(\llbracket B \rrbracket(m)\) evaluates to true.
- \(C_1\) (the code executing when \(B\) is false) must only operate on \(m \in M\) where \(\llbracket B \rrbracket(m)\) evaluates to false.
- Define a filter function \(F_B(M)\) such that
  \[
  F_B(M) = \{ m \in M \mid \llbracket B \rrbracket(m) = \text{true} \}
  \]
- Note: \(F_{\neg B}\) will give us the memories where \(B\) is false.
if(B){C₀} else {C₁} = \mathcal{D}(\mathcal{P}_B(M)) \cup \mathcal{D}(\mathcal{P}_{\neg B}(M))

- Find stores where $B$ is true, evaluate $C₀$ over them
- Find stores where $B$ is false, evaluate $C₁$ over them
- Combine the two results using $\cup$
Command Semantics for If in Python

def filter_memory(B: BoolExpr, M: List[Memory], res = True) -> List[Memory]:
    out = [m for m in M if evaluate_BoolExpr(B, m) == res]
    return list(out)

def evaluate_Cmd(C: Cmd, M: List[Memory]) -> List[Memory]:
    ...

    elif isinstance(C, IfThenElse):
        then_memory = evaluate_Cmd(C.then_, filter_memory(C.cond, M))
        else_memory = evaluate_Cmd(C.else_, filter_memory(C.cond, M, res = False))

        return union_memories(then_memory, else_memory)

    ...

def union_memories(M0: List[Memory], M1: List[Memory]) -> List[Memory]:
    # this implementation is, of course, ridiculous

    # convert everything to sets
    M0_set = set([frozenset(m.items()) for m in M0])
    M1_set = set([frozenset(m.items()) for m in M1])

    M_set = M0_set.union(M1_set)

    # convert back to lists of dicts
    return list([dict(m) for m in M_set])
\[ [\text{while}(B)\{C\}]_P(M) \]

- $B$ must be true in $m \in M$ to execute $C$ once
  - $(\llbracket C \rrbracket_P \circ \mathcal{F}_B)(M)$
- Executing $C$ twice is similar:
  - $(\llbracket C \rrbracket_P \circ \mathcal{F}_B)((\llbracket C \rrbracket_P \circ \mathcal{F}_B)(M))$
- Let $F$ be $\llbracket C \rrbracket_P \circ \mathcal{F}_B$, then execution $i$ times is represented as
  - $F^i(M)$, i.e. $F(F(F(M)))$ for $i = 3$
- If the loop executes $i$ times and exits, the memory stores are:
  - $M_i = \mathcal{F}_{\neg B}(F^i(M))$, because $B$ must be false when we exit the loop
Semantics for While - #2

- Let $M_i = \mathcal{F}_{\neg B}(F^i(M))$ represent executions of the loop body exactly $i$ times, $i \geq 0$
- Then we can define the semantics of those $i$ executions as:

$$
\bigcup_{i \geq 0} M_i = \bigcup_{i \geq 0} \mathcal{F}_{\neg B}(F^i(M)) \\
= \mathcal{F}_{\neg B}(\bigcup_{i \geq 0} F^i(M))
$$

$$
[\text{while}(B)\{C\}]_{\mathcal{P}}(M) = \mathcal{F}_{\neg B}(\bigcup_{i \geq 0} ([C]_{\mathcal{P}} \circ \mathcal{F}_B)^i(M)))
$$

- The semantics of a non-terminating loop are undefined.
def evaluate_Cmd(C: Cmd, M: List[Memory]) -> List[Memory]:
    ...
    elif isinstance(C, While):
        # L0
        out = [m for m in M]  # copy all input states

        pre_iter_memories = filter_memory(C.cond, out)
        accum: List[Memory] = []
        while len(pre_iter_memories):
            after_iter_memories = evaluate_Cmd(C.body, pre_iter_memories)
            accum = union_memories(accum, after_iter_memories)

            # only keep memories where the condition is true
            pre_iter_memories = filter_memory(C.cond, after_iter_memories)

        # This computes L0 U (L1 U L2...) and retains only those
        # memory states where the loop has terminated.
        out = filter_memory(C.cond, union_memories(out, accum), res=False)
        return out
Example of While execution

```plaintext
while(x < 7) {
    y := (y + 1);
    x := (x + 1)
}

START  [{x: 4, y: 0}, {x: 5, y: 0}, {x: 8, y: 0}]
pre:    [{x: 4, y: 0}, {x: 5, y: 0}]
after:  [{x: 5, y: 1}, {x: 6, y: 1}]
accum:  [{x: 5, y: 1}, {x: 6, y: 1}]

pre:    [{x: 5, y: 1}, {x: 6, y: 1}]
after:  [{x: 6, y: 2}, {x: 7, y: 2}]
accum:  [{x: 5, y: 1}, {x: 6, y: 1}, {x: 7, y: 2},
          {x: 6, y: 2}]

pre:    [{x: 6, y: 2}]
after:  [{x: 7, y: 3}]
accum:  [{x: 7, y: 3}, {x: 6, y: 2}, {x: 5, y: 1},
          {x: 6, y: 1}, {x: 7, y: 2}]

END     [{x: 7, y: 3}, {x: 7, y: 2}, {x: 8, y: 0}]```
Wrapping up the semantics

- \( [C]_{\mathcal{P}}(\emptyset) = \emptyset \)
  - Starting from an empty set of states leads to an empty set of states

- Key ideas:
  - Grammar \( \rightarrow \) AST
  - AST \( \rightarrow \) Semantics
  - Semantics \( \rightarrow \) Interpreter
Outline

Introduction

A Tiny Language and Its Semantics

To be continued ...
• Abstraction, and building an abstract interpreter
• This lecture was based on material from Chapter 3 in Rival and Yi
• You can find the Python code on GitHub
  • This lecture covered tinyast.py and sem.py