CSC2/455 Software Analysis and Improvement
Abstract Interpretation - III

Sreepathi Pai

April 6, 2020

URCS

Outline

Recap

Value Abstractions

Computable Abstract Semantics

Postscript

Outline

Previous lecture

- Previous lecture
  - Concrete Semantics for a Small Language
- Today:
  - Value abstractions
  - Non-relational Abstractions
  - Abstract Semantics
  - Soundness, termination, etc.

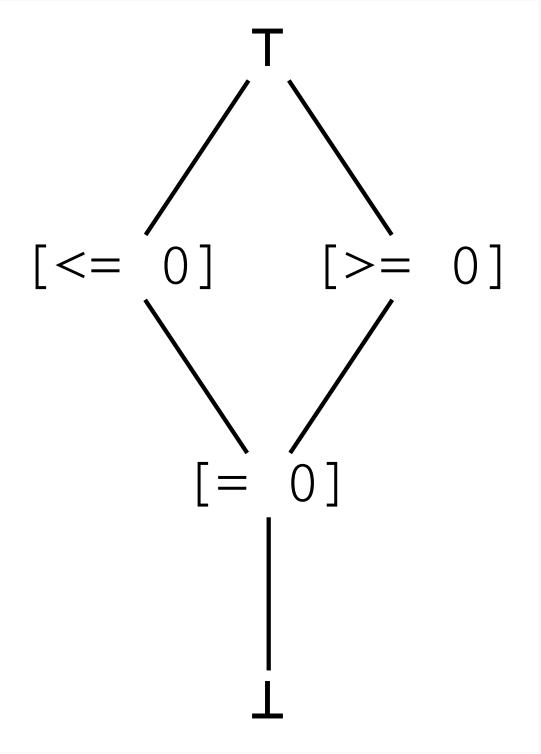
Outline

Abstraction Examples

- Consider the concrete memory state M:
  - $\{\{\mathrm{x} \mapsto 7, \mathrm{y} \mapsto 2\}, \{\mathrm{x} \mapsto 8, \mathrm{y} \mapsto 0\}\}$
  - How shall we abstract it?
- $x = \{7, 8\}$
  - Signs: $x = [\geq 0]$
  - Intervals: $x = [7, 8]$
- $y = \{0, 2\}$
  - Signs: $y = [\geq 0]$
  - Intervals: $y = [0, 2]$ (note: $[0, 2] = \{0, 1, 2\}$)
- Alternatively:
  - Signs: $x = \top$ (here, $\top = \mathbb{V} = \mathbb{Z}$)
  - Intervals: $y = [0, 3]$
  - Multiple abstractions are possible, but some are less precise

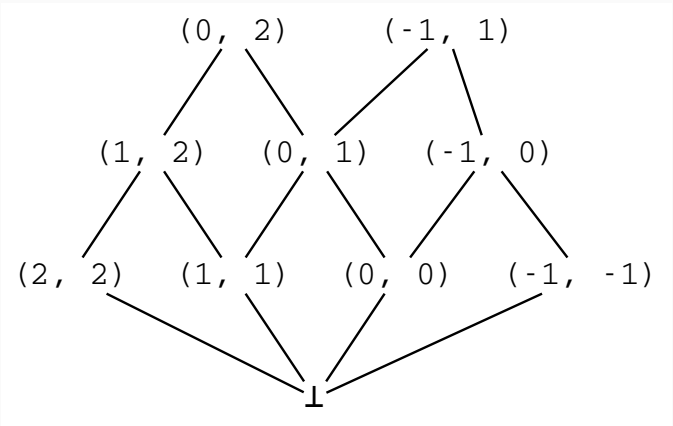
Lattice for Signs Domain

- Signs, $\mathbb{A}_{\mathscr{S}} = \{\top, [\leq 0], [\geq 0], [= 0], \bot\}$
  - $\top = \mathbb{V}$ (recall $\mathbb{V} = \mathbb{Z}$ for our language)
  - $[\leq 0] = \{x \mid x \leq 0\}$
  - $[\geq 0] = \{x \mid x \geq 0\}$
  - $[= 0] = \{0\}$
  - $\bot = \emptyset$
- Order relation $\sqsubseteq$
  - Items *lower* in the lattice are more precise
  - $a \sqsubseteq b$, read as a less than b
- Join $\sqcup$
  - Least upper bound, lub



Lattice for Intervals Domain

- Intervals, $\mathbb{A}_{\mathcal{I}} = \{\top, \bot\} \cup \{[n, m] \mid n, m \in \mathbb{Z}\}$
  - $\top = (-\infty, +\infty) = \mathbb{V} = \mathbb{Z}$
  - $[n, m] = \{x \mid n \leq x \leq m\}$
  - $[n, +\infty) = \{x \mid n \leq x\}$
  - $(-\infty, m) = \{x \mid x \leq m\}$
  - $\bot = \emptyset$
- Infinite lattice
- Order relation $\sqsubseteq$ and Join $\sqcup$ supported



Abstraction and Concretization Functions (Informal)

- Given an element c of the concrete domain $\mathbb{C}$, we want $a \in \mathbb{A}$
  - c is a set of values
  - e.g. $x = \{7, 8\}$
- Let the value abstraction function be $\phi_{\mathcal{V}}$
  - $\phi_{\mathcal{V}} : \mathbb{C} \to \mathbb{A}$
- Similarly, given an abstract element $a \in \mathbb{A}$, we want the concrete element c corresponding to it
  - e.g., $a = [<= 0] \in \mathbb{A}_{\mathcal{S}}$
  - So, corresponding $c = \{..., -3, -2, -1, 0\}$
- Let this value concretization function be $\gamma_{\mathcal{V}}$
  - $\gamma_{\mathcal{V}} : \mathbb{A} \to \mathbb{C}$
- Key questions: how do we relate $\phi_{\mathcal{V}}$ to $\gamma_{\mathcal{V}}$
  - soundly,
  - precisely?

Code for Signs

```
class SignsDomain(object):
    LTZ = "[<= 0]"
    GTZ = "[>= 0]"
    EQZ = "[= 0]"
    TOP = "TOP"
    BOT = "BOT"
    finite_height = True

    def phi(self, v: int):
        if v == 0:
            return self.EQZ
        elif v > 0:
            return self.GTZ
        elif v < 0:
            return self.LTZ
        else:
            raise ValueError(f"Unknown value for signs abstraction {v}")
```

⊑ in code

```
class SignsDomain(object):
    ...
    # it helps to think of abstract elements as sets, with lte
    # denoting set inclusion. So we're asking, is x included in y?
    def lte(self, x, y):
        # bot is always less than everything else
        # empty set {} is always included
        if x == self.BOT: return True

        # top is only lte
        # top is all possible values, so it is only included in itself
        if x == self.TOP:
            if y != self.TOP: return False
            return True

        # eqz is the set {0}, which is included in all sets (>=0, <=0) exce
        if x == self.EQZ:
            if y == self.BOT: return False
            return True

        if x == self.LTZ or x == self.GTZ:
            if y == x: return True
            if y == self.TOP: return True

            # these sets are not included in {0} or {} or {>=0} [resp. {<=0
            return False
```

⊔ in code

```
class SignsDomain(object):
    ...
    def lub(self, x, y):
        if self.lte(x, y): return y # y includes x
        if self.lte(y, x): return x # x includes y

        # if incomparable, then we return T
        return self.TOP
```

Concrete Domains

- Values in our concrete domain belong to $\wp(\mathbb{M})$
  - Recall $\mathbb{M} = \mathbb{X} \to \mathbb{V}$
- A concrete domain is the pair $(\mathbb{C}, \subseteq)$
  - $\mathbb{C} = \wp(\mathbb{M})$
  - If $x, y \in \mathbb{C}$, and $x \subseteq y$, then x implies y
  - x and y are behavioural properties expressed as sets
  - x is at least as "strong" as y
- Example:
  - x is set of all state where $\mathtt{x} > 10$
  - y is set of all state where $\mathtt{x}$ is non-negative
  - Clearly $x \subseteq y$

Abstraction

- An abstract domain is $(\mathbb{A}, \sqsubseteq)$
  - $\sqsubseteq$ orders members of $\mathbb{A}$
- An abstraction relation $(\models) \subseteq \mathbb{C} \times \mathbb{A}$, such that:
  - for all $c \in \mathbb{C}, a_0, a_1 \in \mathbb{A}$, if $c \models a_0$, and $a_0 \sqsubseteq a_1$, then $c \models a_1$
  - example: $c = \{0\}$, $a_0 = [= 0]$, $a_1 = [\geq 0]$ in the signs domain
  - for all $c_0, c_1 \in \mathbb{C}, a \in \mathbb{A}$, if $c_0 \subseteq c_1$ and $c_1 \models a$, then $c_0 \models a$
  - example: $c_0 = \{3, 5\}$, $c_1 = \{2, 3, 4, 5, 6\}$, $a = [2, 6]$
- The goal of abstraction is to map $c \in \mathbb{C}$ to the most precise $a \in \mathbb{A}$

Concretization Function

- $\gamma_{\mathscr{V}} : \mathbb{A} \to \mathbb{C}$, the concretization function is defined as:
  - $\gamma_{\mathscr{V}}(a) \models a$,
  - $\gamma_{\mathscr{V}}(a)$ is the maximum concrete element of $\mathbb{C}$ that satisfies a
  - I.e., if $\gamma_{\mathscr{V}}(a) = c$, there no other c' such that $c' \models a$ and $c \subseteq c'$
- Examples:
  - $\gamma_{\mathscr{S}}([\leq 0]) = \{x \mid x \leq 0\}$
  - $\gamma_{\mathscr{S}}([0, 3]) = \{0, 1, 2, 3\}$
  - $\gamma(\bot) = \emptyset$
- Concretization can be used instead of $\models$ to define the abstraction relation:
  - $\forall c \in \mathbb{C}, a \in \mathbb{A} \;\; c \models a \iff c \subseteq \gamma_{\mathscr{V}}(a)$
  - e.g.: using signs, $c = \{3\}$, $a = [\geq 0]$, $\gamma_{\mathscr{V}}(a) = \{0, 1, 2, 3, 4, ...\}$

(Best) Abstraction Function

- $\alpha : \mathbb{C} \to \mathbb{A}$, the abstraction function is defined as:
  - $c \models \alpha(c)$
  - $\alpha(c)$ is the minimum element of $\mathbb{A}$ that is satisfied by c
  - i.e., if $\alpha(c) = a$, there is no other a' such that $c \models a'$ and $a' \sqsubseteq a$
- Examples:
  - $\alpha_{\mathscr{S}}(\{0\}) = [= 0]$
  - $\alpha_{\mathscr{I}}(\{0, 3\}) = [0, 3]$
- α may not exist

When α may not exist

- When $[= 0]$ is removed from signs, it has no best abstraction function
  - $\{0\}$ can be described by either $[\leq 0]$ or $[\geq 0]$
  - $[\leq 0] \not\subseteq [\geq 0]$ and $[\geq 0] \not\subseteq [\leq 0]$
- Convex polyhedra
  - No finite set of linear inequalities can approximate a circle (in the 2-D domain) or its equivalents in higher domains
  - Each linear equality is a tangent to the circle

Galois Connections

- When $\alpha_{\sim\mathcal{V}}$ exists:

$$\forall c \in \mathbb{C}, a \in \mathbb{A}, \quad \alpha_{\sim\mathcal{V}}(c) \sqsubseteq a \iff c \subseteq \gamma_{\sim\mathcal{V}}(a)$$

- The pair $\gamma_{\sim\mathcal{V}}$ and $\alpha_{\sim\mathcal{V}}$ form a Galois connection with the following properties:
  - $\gamma_{\sim\mathcal{V}}$ and $\alpha_{\sim\mathcal{V}}$ are monotone
  - $\forall c \in \mathbb{C}, c \subseteq \gamma_{\sim\mathcal{V}}(\alpha(c))$
  - $\forall a \in \mathbb{A}, \alpha_{\sim\mathcal{V}}(\gamma_{\sim\mathcal{V}}(a)) \sqsubseteq a$

Non-relational Abstraction #1

- A non-relational abstraction does not capture relationships between variables
  - Each variable is abstracted independently
- We can extend the value abstraction functions we've defined so far to define a non-relational abstraction:
  - $M^\sharp$ is the abstraction of M
  - $M \subseteq \gamma_{\mathcal{N}}(M^\sharp)$
- The concretization function is defined as:
  - $\gamma_{\mathcal{N}} : M^\sharp \mapsto \{m \in \mathbb{M} \mid \forall \mathtt{x} \in \mathbb{X}, m(\mathtt{x}) \in \gamma_{\mathcal{V}}(M^\sharp(\mathtt{x}))\}$
- The order relation $\sqsubseteq_{\mathcal{V}}$ is pointwise-extended:
  - $M_0^\sharp \sqsubseteq^\sharp M_1^\sharp$ if and only if $\forall \mathtt{x} \in \mathbb{X}, M_0^\sharp(\mathtt{x}) \sqsubseteq_{\mathcal{V}} M_1^\sharp(\mathtt{x})$

Non-relational Abstraction #2

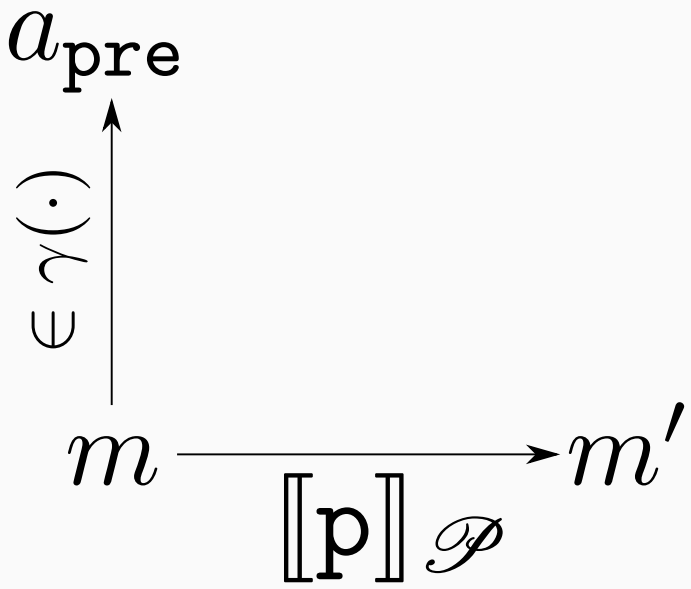
- The bottom $\bot_{\mathcal{N}}$ is defined as:
  - $\forall \mathtt{x} \in \mathbb{X}, \bot_{\mathcal{N}}(\mathtt{x}) = \bot_{\mathcal{V}}$
- The abstraction function, if it exists, is defined as:
  - $\alpha_{\mathcal{N}} : M \mapsto (\mathtt{x} \in \mathbb{X}) \mapsto \alpha_{\mathcal{V}}(\{m(x) | m \in M\})$

Code Implementation for a Non-relational Abstraction

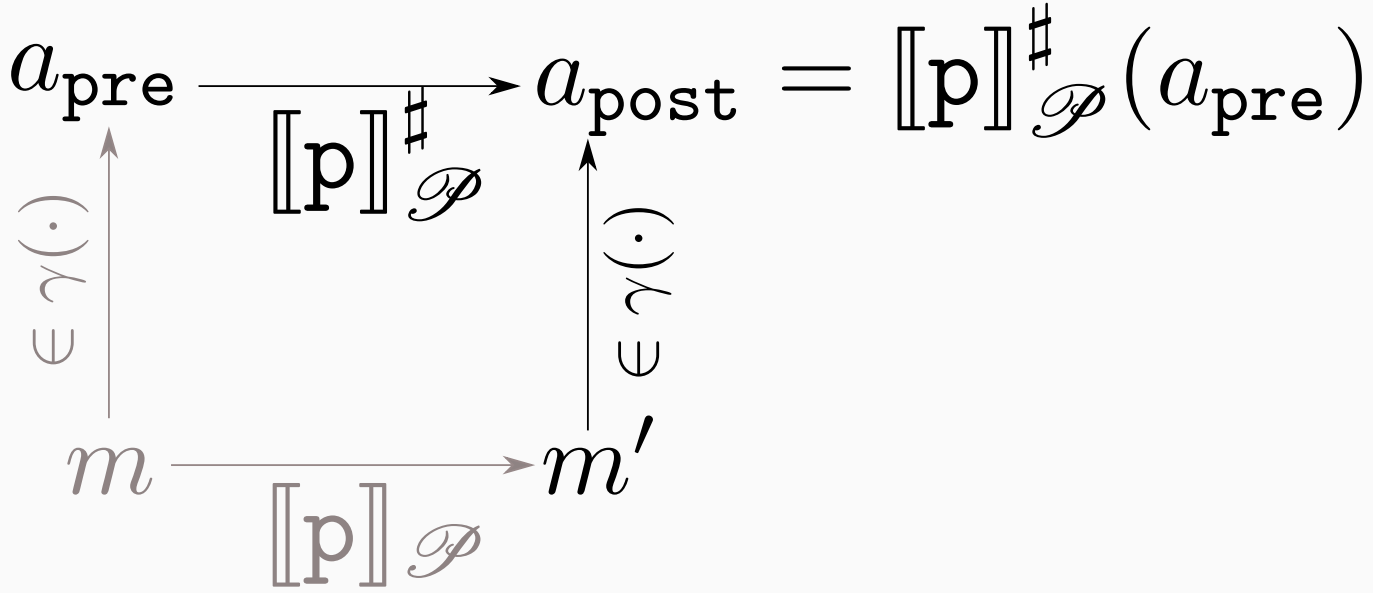
```
class NonRelationalAbstraction(object):
    def __init__(self, domain):
        self.dom = domain

    def phi(self, M):
        m_accum = {}

        for m in M:
            m_abs = {}
            for x in m:
                m_abs[x] = self.dom.phi(m[x])

            if len(m_accum) == 0:
                m_accum = m_abs
            else:
                m_accum = self.union(m_accum, m_abs)

        # also construct BOT
        self.BOT = {}
        for x in m_accum:
            self.BOT[x] = self.dom.BOT

        return m_accum

    def lte(self, M0_abs, M1_abs):
        for x in M0_abs:
            if not self.dom.lte(M0_abs[x], M1_abs[x]): return False

        return True
```

Outline

Goal: Sound Static Analysis



Goal: Sound Static Analysis



Abstraction of empty set

Recall:

$$[\![C]\!](\emptyset) = \emptyset$$

so we will define:

$$[\![C]\!]^{\sharp}_{\mathscr{P}}(\bot) = \bot$$

In code:

```
def evaluate_Cmd_abs(C: Cmd, M_abs: AbstractMemory, abstraction) -> Abstrac
    ...

    if M_abs == abstraction.BOT:
        return M_abs
    ...
```

Skip

$$[\![\texttt{skip}]\!]^{\sharp}_{\mathscr{D}}(M^{\sharp}) = M^{\sharp}$$

In code:

```
def evaluate_Cmd_abs(C: Cmd, M_abs: AbstractMemory, abstraction) -> Abstrac
    ...

    # the value abstraction
    v_abs = abstraction.dom

    if isinstance(C, Skip):
        return M_abs
    elif isinstance(C, Program):
        return evaluate_Cmd_abs(C.program, M_abs, abstraction)
    ...
```

Composition

$$[\![C_0; C_1]\!]^\sharp_{\mathscr{P}}(M^\sharp) = [\![C_1]\!]^\sharp_{\mathscr{P}}([\![C_0]\!]^\sharp_{\mathscr{P}}(M^\sharp))$$

- This seems to be intuitive, but we need to show that:
  - The concrete postcondition of $[\![C_0; C_1]\!]_{\mathscr{P}}$ is over-approximated by $[\![C_0; C_1]\!]^\sharp_{\mathscr{P}}$
  - I.e. $[\![C_0; C_1]\!]_{\mathscr{P}} \subseteq \gamma([\![C_0; C_1]\!]^\sharp_{\mathscr{P}})$

Theorem: Approximation of Compositions: Let $F_0, F_1 : \wp(\mathbb{M}) \to \wp(\mathbb{M})$ be two monotone functions that are overapproximated by $F_0^\sharp, F_1^\sharp : \mathbb{A} \to \mathbb{A}$, i.e. $F_0 \circ \gamma \subseteq \gamma \circ F_0^\sharp$ and $F_1 \circ \gamma \subseteq \gamma \circ F_1^\sharp$. Then, $F_0 \circ F_1$ can be approximated by $F_0^\sharp \circ F_1^\sharp$

Composition in Code

```
def evaluate_Cmd_abs(C: Cmd, M_abs: AbstractMemory, abstraction) -> Abstrac
    ...
    elif isinstance(C, Seq):
        return evaluate_Cmd_abs(C.cmd1,
                                evaluate_Cmd_abs(C.cmd0, M_abs,
                                                 abstraction),
                                abstraction)
    ...
```

Expressions

$$
\begin{aligned}
[\![E]\!]^\sharp &: \mathbb{A} \to \mathbb{A}_{\mathcal{V}} \\
[\![n]\!]^\sharp(M^\sharp) &= \phi_{\mathcal{V}}(n) \\
[\![\mathtt{x}]\!]^\sharp(M^\sharp) &= M^\sharp(\mathtt{x}) \\
[\![E_0 \odot E_1]\!]^\sharp(M^\sharp) &= f^\sharp_\odot([\![E_0]\!]^\sharp(M^\sharp), [\![E_1]\!]^\sharp(M^\sharp))
\end{aligned}
$$

- $\phi_{\mathcal{V}}$ can be replaced by $\alpha_{\mathcal{V}}$ if it exists
  - Otherwise just return an abstract element such that $\{n\} \subseteq \gamma(\phi_{\mathcal{V}}(n))$

Expressions: $f_{\odot}^{\sharp}$

$$\forall n_0^{\sharp}, n_1^{\sharp} \in \mathbb{A}_{\mathcal{V}}, \{f_{\odot}(n_0, n_1) | n_0 \in \gamma_{\mathcal{V}}(n_0^{\sharp}) \text{ and } n_1 \in \gamma_{\mathcal{V}}(n_1^{\sharp})\} \subseteq \gamma_{\mathcal{V}}(f_{\odot}^{\sharp}(n_0^{\sharp}, n_1^{\sharp}))$$

- The result of applying $f_{\odot}^{\sharp}(n_0^{\sharp}, n_1^{\sharp})$, when concretized
  - $\gamma_{\mathcal{V}}(f_{\odot}^{\sharp}(n_0^{\sharp}, n_1^{\sharp}))$
- must include the concrete set formed when we apply $f_{\odot}$ to ...
- ... the elements of the individual concretizations of $n_0^{\sharp}, n_1^{\sharp}$
  - $n_0 \in \gamma_{\mathcal{V}}(n_0^{\sharp})$
  - $n_1 \in \gamma_{\mathcal{V}}(n_1^{\sharp})$

Examples (using signs):

- $f_{+}^{\sharp}([\geq 0], [\geq 0]) = [\geq 0]$
- $f_{+}^{\sharp}([\geq 0], [\leq 0]) = \top$

Expressions: $f^\sharp_\odot$ in code (Signs)

```
def f_binop(self, op, left, right):
    if op == '+':
        return self.lub(left, right)
    elif op == '*':
        if left != right:
            return self.lub(left, right)
        elif left == self.LTZ:
            return self.GTZ  # - * - = +
        elif left == self.GTZ:
            return self.GTZ  # + * + = +
    elif op == '-':
        if left == right:
            if left != self.EQZ and left != self.BOT:
                return self.TOP

            return left # {0} - {0} => {0}, {} - {} => {}
        else:
            return left   # {+ve} - {-ve} => {+ve}, {-ve} - {+ve} => {-ve}

    else:
        raise NotImplementedError(f'Operator {op}')
```

- $f^\sharp_\odot$ is per abstract domain (not per language as in the concrete semantics)

Expressions: $f^{\sharp}_{\odot}$ in code (Intervals)

See `f_binop` in `dom_intervals.py`.

- The tricky aspects revolve around handling $-\infty$ and $+\infty$

Evaluating Expressions Abstractly in Code

```
def evaluate_Expr_abs(E: Expr, m: AbstractMemory, vabs):
    if isinstance(E, Scalar):
        return vabs.phi(E)
    elif isinstance(E, Var):
        return m[E.name]
    elif isinstance(E, BinOp):
        return vabs.f_binop(E.op,
                            evaluate_Expr_abs(E.left, m, vabs),
                            evaluate_Expr_abs(E.right, m, vabs))
```

Assignments and `input`

The concrete semantics are:

$$[\![\mathtt{x} := E]\!]_{\mathscr{P}}(M) = \{m[\mathtt{x} \mapsto [\![E]\!](m)] \mid m \in M\}$$

The abstract semantics are:

$$[\![\mathtt{x} := E]\!]^{\sharp}_{\mathscr{P}}(M^{\sharp}) = M^{\sharp}[\mathtt{x} \mapsto [\![E]\!]^{\sharp}(M^{\sharp})]$$

Similarly, since `input` also writes to a variable:

$$[\![\mathtt{input(x)}]\!]^{\sharp}_{\mathscr{P}}(M^{\sharp}) = M^{\sharp}[\mathtt{x} \mapsto \top_{\mathscr{V}}]$$

Recall that `input` can return any value from the user.

Assignments and `input` Code

```
def evaluate_Cmd_abs(C: Cmd, M_abs: AbstractMemory, abstraction) -> Abstrac
    def update_abs_memories(var, value_lambda):
        out = dict(M_abs)
        out[var] = value_lambda(M_abs)
        return out

    ...

    elif isinstance(C, Assign):
        return update_abs_memories(C.left.name,
                                   lambda m: evaluate_Expr_abs(C.right,
                                                               m, v_abs))
    elif isinstance(C, Input):
        return update_abs_memories(C.var.name, lambda _: v_abs.TOP)

    ...
```

Conditionals: Example

```
# M# = {x: T, y: T}
x := 7
# M# = {x: [7, 7], y: T}

if (x > 5)
  # M# = {x: [6, +inf), y: T}
  y = 1
  # M# = {x: [6, +inf), y: [1, 1]}
else
  # M# = {x: (-inf, 5], y: T}
  y = 10
  # M# = {x: (-inf, 5], y: [10, 10]}

# M# = {x: [-inf, +inf], y: [1, 10]}
```

- We need a abstract filtering function $\mathscr{F}_B^\sharp$
  - Its effects are shown
- We need to join the abstract elements:
  - Use the lub (least upper bound), here $\sqcup^\sharp$
- But we have lost precision for x!

$\mathscr{F}_B^\sharp$ must refine

```
# M# = {x: T, y: T}
x := 7
# M# = {x: [7, 7], y: T}

if (x > 5)
  # M# = {x: [7, 7], y: T}
  y = 1
  # M# = {x: [7, 7], y: [1, 1]}
else
  # M# = {x: BOT, y: BOT}
  y = 10
  # M# = {x: BOT, y: BOT}

# M# = {x: [7, 7], y: [1, 1]}
```

- For the true part, $[6, +\infty)$ is refined to $[7, 7]$
- For the false part, $(-\infty, 5]$ does not include $[7, 7]$
  - So the abstract state $M^\sharp$ is refined to $\bot$, by setting all variables to $\bot$
  - Recall that $[\![C]\!]^\sharp_{\mathscr{P}}(\bot) = \bot$ and that $a \sqcup^\sharp \bot = a$

Soundness properties

For $\mathscr{F}_B^\sharp$: For all B and abstract states $M^\sharp$

$$\mathscr{F}_B(\gamma(M^\sharp)) \subseteq \gamma(\mathscr{F}_B^\sharp(M^\sharp))$$

For $\sqcup^\sharp$ over $M_0^\sharp$ and $M_1^\sharp$:

$$\gamma(M_0^\sharp) \cup \gamma(M_1^\sharp) \subseteq \gamma(M_0^\sharp \sqcup^\sharp M_1^\sharp)$$

Abstract Semantics of If

$$[\![\texttt{if}(B)\{C_0\}\,\texttt{else}\,\{C_1\}]\!]^\sharp_{\mathscr{P}}(M^\sharp) = [\![C_0]\!]^\sharp_{\mathscr{P}}(\mathscr{F}^\sharp_B(M^\sharp)) \sqcup^\sharp [\![C_1]\!]^\sharp_{\mathscr{P}}(\mathscr{F}^\sharp_{\neg B}(M^\sharp))$$

Code:

```
def evaluate_Cmd_abs(C: Cmd, M_abs: AbstractMemory, abstraction) -> Abstrac
    ...
    elif isinstance(C, IfThenElse):
        then_memory, else_memory = filter_memory_abs(C.cond, M_abs, v_abs)

        then_memory = evaluate_Cmd_abs(C.then_, then_memory, abstraction)
        else_memory = evaluate_Cmd_abs(C.else_, else_memory, abstraction)

        ite_memory = abstraction.union(then_memory, else_memory)

        return ite_memory
```

Code for `filter_memory_abs`

```
def filter_memory_abs(B: BoolExpr, M_abs: AbstractMemory, vabs) ->
      Tuple[AbstractMemory, AbstractMemory]:
    true_abs, false_abs = evaluate_BoolExpr_abs(B, M_abs, vabs)
    var_abs = M_abs[B.left.name]

    true_abs = vabs.refine(var_abs, true_abs)

    if true_abs != vabs.BOT:
        # may enter true part
        M_abs_true = dict(M_abs)
        M_abs_true[B.left.name] = true_abs
    else:
        M_abs_true = dict([(m, vabs.BOT) for m in M_abs])

    false_abs =  vabs.refine(var_abs, false_abs)

    if false_abs != vabs.BOT:
        # may enter false part
        M_abs_false = dict(M_abs)
        M_abs_false[B.left.name] = false_abs
    else:
        M_abs_false = dict([(m, vabs.BOT) for m in M_abs])

    return M_abs_true, M_abs_false
```

Partial code for `f_cmpop` and `refine` in the Intervals domain

```
def refine(self, l, r):
    l = self._norm(l)
    r = self._norm(r)

    if l == self.BOT: return r
    if r == self.BOT: return l

    new_start = max(l[0], r[0])
    new_end = min(l[1], r[1])

    return self._norm((new_start, new_end))

def f_cmpop(self, op, left, c):
    left = self._norm(left)
    c = self._norm(c)

    # assume integers
    if op == '<':
        return (self.NINF, c[0] - 1), (c[0], self.PINF)
    elif op == '<=':
        return (self.NINF, c[0]), (c[0] + 1, self.PINF)
    elif op == '>':
        return (c[0] + 1, self.PINF), (self.NINF, c[0])
    elif op == '>=':
        return (c[0], self.PINF), (self.NINF, c[0] - 1)
    else:
        raise NotImplementedError(f'Operator {op}')
```

Partial code for `f_cmpop` and `refine` in the Signs domain

```
def refine(self, l, r):
    if self.lte(l, r): return l
    if self.lte(r, l): return r

    return self.TOP

def f_cmpop(self, op, left, c):
    # (abst of c, op) : (variable's true domain, variables false domai
    abs_results = {(self.EQZ, '<'): (self.LTZ, self.GTZ),
                   (self.EQZ, '<='): (self.LTZ, self.GTZ),
                   (self.EQZ, '>'): (self.GTZ, self.LTZ),
                   (self.EQZ, '>='): (self.GTZ, self.LTZ),
                   (self.EQZ, '!='): (self.TOP, self.EQZ),

                   (self.GTZ, '>'): (self.GTZ, self.TOP),
                   (self.GTZ, '<'): (self.TOP, self.GTZ),
                   (self.GTZ, '<='): (self.TOP, self.GTZ),
                   (self.GTZ, '>='): (self.GTZ, self.TOP),
                   }

    key = (c, op)
    if key not in abs_results:
        raise NotImplementedError(f"{key} not implemented")

    return abs_results[key]
```

While: Example #1: Infinite Loop

```
x := 0
while(x >= 0)  {
   x := x + 1
}
```

If we analyze this program abstractly using signs, using $\sqcup^\sharp$ to combine states across loop iterations, as we did in the concrete execution, the analysis will reach a fixpoint, which can be used to terminate the analysis.

- $M^\sharp(\mathtt{x}) = ([= 0] \sqcup^\sharp [>= 0] \sqcup^\sharp [>= 0]) = [>= 0]$

If we analyze this program abstractly using intervals, the analysis will not terminate.

- $M^\sharp(\mathtt{x}) = [0, 0] \sqcup^\sharp [1, 1] \sqcup^\sharp [2, 2] \sqcup^\sharp [3, 3]...$

While: Example #2: Infinite Loop

```
x := 0
while(x <= 100)  {
   if (x >= 50) {
       x := 10
   } else {
       x := x + 1
   }
}
```

If we analyze this program abstractly using signs, the analysis terminates as in the previous example

- $M^\sharp(\mathtt{x}) = ([= 0] \sqcup^\sharp [>= 0] \sqcup^\sharp [>= 0]) = [>= 0]$

If we analyze this program abstractly using intervals, the analysis also terminates, but after 50 analysis iterations.

- $M^\sharp(\mathtt{x}) = [0,0] \sqcup^\sharp [0,1] \sqcup^\sharp [0,2] \sqcup^\sharp \ldots \sqcup^\sharp [0,50] \sqcup^\sharp [0,50] = [0,50]$

Observations

- Signs is a lattice with a finite height
  - $\sqcup^\sharp$ will eventually reach a fix point
- The Intervals lattice does not have a finite height
  - No such guarantees

Widening Operator

- Define an operator ∇ so that the sequence will explicitly reach a stationary point
- Soundness condition

$$\gamma(a_0) \cup \gamma(a_1) \subseteq \gamma(a_0 \nabla a_1)$$

- For all $(a_n)_{n \in \mathbb{N}}$, the sequence $(a'_n)_{n \in \mathbb{N}}$ is ultimately stationary:
  - $a'_0 = a_0$
  - $a'_{n+1} = a'_n \nabla a_n$

Code for Widening Operator for Intervals

```
def widen(self, x, y):
    # assume x is previous and y is current

    # compute a_n
    u = self.lub(x, y)

    if u[0] == x[0]:
        # stationary left
        return (u[0], u[1] if u[1] == x[1] else self.PINF)
    elif u[1] == x[1]:
        # stationary right
        return (u[0] if u[0] == x[0] else self.NINF, u[1])
    else:
        return u
```

Abstract Semantics for While

$$[\![\texttt{while}(B)\{C\}]\!]^\sharp_{\mathscr{P}}(M^\sharp) = \mathscr{F}^\sharp_{\neg B}(\texttt{abs\_iter}([\![C]\!]^\sharp_{\mathscr{P}} \circ \mathscr{F}^\sharp_B, M^\sharp))$$

Code:

```
def evaluate_Cmd_abs(C: Cmd, M_abs: AbstractMemory, abstraction) ->
    ...

    elif isinstance(C, While):
        def F_abs(MM_abs):
            pre_memory, _ = filter_memory_abs(C.cond, MM_abs,
                                              v_abs)
            post_memory = evaluate_Cmd_abs(C.body, pre_memory,
                                           abstraction)
            return post_memory

        _, out = filter_memory_abs(C.cond,
                                   abs_iter(F_abs, M_abs,
                                            abstraction),
                                   v_abs)
        return out

    ...
```

abs_iter in code

```
def abs_iter(F_abs, M_abs, abstraction):
    R = M_abs
    while True:
        T = R
        if abstraction.dom.finite_height:
            R = abstraction.union(R, F_abs(R))
        else:
            R = abstraction.widen(R, F_abs(R))

        if R == T: break

    return T
```

Outline

References

- Code that accompanies this lecture can be found in GitHub repository:
  - Abstract Domains: dom_signs.py and dom_intervals.py
  - Non-Relational Abstraction: abstractions.py
  - Abstract Semantics: sem_abs.py
- Chapter 3 of Rival and Yi.
  - This covers compositional semantics
  - Also has examples of relational domains (convex polyhedra)
- Abstract interpretation can also be performed using transitional semantics
  - Chapter 4 of Rival and Yi