CSC2/455 Software Analysis and Improvement
Abstract Interpretation - III

Sreepathi Pai
April 12, 2021
URCS
Outline

Recap

Value Abstractions

Computable Abstract Semantics

Postscript
Outline

Recap

Value Abstractions

Computable Abstract Semantics

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

Value Abstractions

Computable Abstract Semantics

Postscript
Abstraction Examples

- Consider the concrete memory state $M$:
  - $\{\{x \mapsto 7, y \mapsto 2\}, \{x \mapsto 8, y \mapsto 0\}\}$
  - 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, $\top = \forall = \mathbb{Z}$)
  - Intervals: $y = [0, 3]$
  - Multiple abstractions are possible, but some are less precise
Lattice for Signs Domain

- **Signs**, $\mathbb{A}_\mathcal{F} = \{\top, [\leq 0], [\geq 0], [= 0], \bot\}$
  - $\top = \mathbb{V}$ (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, $\mathbb{A}_I = \{\top, \bot\} \cup \{[n, m] \mid n, m \in \mathbb{Z}\}$
  - $\top = (-\infty, +\infty) = \mathbb{V} = \mathbb{Z}$
  - $[n, m] = \{x \mid n \leq x \leq m\}$
  - $[n, +\infty) = \{x \mid n \leq x\}$
  - $(-\infty, m) = \{x \mid x \leq m\}$
  - $\bot = \emptyset$

- 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 $\phi_V$
  - $\phi_V : \mathbb{C} \rightarrow \mathbb{A}$
- Similarly, given an abstract element $a \in \mathbb{A}$, we want the concrete element $c$ corresponding to it
  - e.g., $a = [\leq 0] \in \mathbb{A}_\varphi$
  - So, corresponding $c = \{..., -3, -2, -1, 0\}$
- Let this value concretization function be $\gamma_V$
  - $\gamma_V : \mathbb{A} \rightarrow \mathbb{C}$
- Key questions: how do we relate $\phi_V$ to $\gamma_V$
  - soundly,
  - precisely?
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}"
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) except
        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
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 $\mathcal{P}(M)$
  - Recall $M = X \rightarrow \mathcal{V}$
- A concrete domain is the pair $(C, \subseteq)$
  - $C = \mathcal{P}(M)$
  - If $x, y \in C$, 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 $x > 10$
  - $y$ is set of all state where $x$ is non-negative
  - Clearly $x \subseteq y$
Abstraction

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

- $\gamma^\gamma : \mathbb{A} \to \mathbb{C}$, the concretization function is defined as:
  - $\gamma^\gamma(a) \models a$,
  - $\gamma^\gamma(a)$ is the maximum concrete element of $\mathbb{C}$ that satisfies $a$
  - I.e., if $\gamma^\gamma(a) = c$, there no other $c'$ such that $c' \models a$ and $c \subseteq c'$

- Examples:
  - $\gamma^\gamma([\leq 0]) = \{ x \mid x \leq 0 \}$
  - $\gamma^\gamma([0, 3]) = \{0, 1, 2, 3\}$
  - $\gamma(\bot) = \emptyset$

- Concretization can be used instead of $\models$ to define the abstraction relation:
  - $\forall c \in \mathbb{C}, a \in \mathbb{A}$, $c \models a \iff c \subseteq \gamma^\gamma(a)$
  - e.g.: using signs, $c = \{3\}$, $a = [\geq 0]$, $\gamma^\gamma(a) = \{0, 1, 2, 3, 4, \ldots\}$
• $\alpha : C \rightarrow A$, the abstraction function is defined as:
  • $c \models \alpha(c)$
  • $\alpha(c)$ is the minimum element of $A$ 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:
  • $\alpha_{\mathcal{G}}(\{0\}) = [= 0]$
  • $\alpha_{\mathcal{G}}(\{0, 3\}) = [0, 3]$

• $\alpha$ may not exist
When $\alpha$ 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
• When $\alpha_\forall$ exists:

$$\forall c \in C, \ a \in A, \ \alpha_\forall(c) \sqsubseteq a \iff c \subseteq \gamma_\forall(a)$$

• The pair $\gamma_\forall$ and $\alpha_\forall$ form a Galois connection with the following properties:
  
  • $\gamma_\forall$ and $\alpha_\forall$ are monotone
  
  • $\forall c \in C, \ c \subseteq \gamma_\forall(\alpha(c))$
  
  • $\forall a \in A, \ \alpha_\forall(\gamma_\forall(a)) \sqsubseteq a$
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:

- \( M' \) is the abstraction of \( M \)
- \( M \subseteq \gamma_N(M') \)

The concretization function is defined as:

\[ \gamma_N : M' \mapsto \{ m \in M \mid \forall x \in X, m(x) \in \gamma_V(M'(x)) \} \]

The order relation \( \sqsubseteq_V \) is pointwise-extended:

- \( M'_0 \sqsubseteq V M'_1 \) if and only if \( \forall x \in X, M'_0(x) \sqsubseteq_V M'_1(x) \)
The bottom $\bot_N$ is defined as:

- $\forall x \in X, \bot_N(x) = \bot_V$

The abstraction function, if it exists, is defined as:

- $\alpha_N : M \mapsto (x \in X) \mapsto \alpha_V(\{m(x) | m \in M\})$
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

Recap

Value Abstractions

Computable Abstract Semantics

Postscript
Goal: Sound Static Analysis

\[
\begin{align*}
& \downarrow \gamma \\
& \downarrow \gamma \\
& m \xrightarrow{[p]_{\mathcal{P}}} m'
\end{align*}
\]
Goal: Sound Static Analysis

\[ a_{\text{pre}} \xrightarrow{[p] \mathcal{P}} a_{\text{post}} = [p] \mathcal{P} (a_{\text{pre}}) \]
Abstraction of empty set

Recall:

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

so we will define:

\[ [C]^{\sharp}_{\mathcal{P}}(\bot) = \bot \]

In code:

```python
def evaluate_Cmd_abs(C: Cmd, M_abs: AbstractMemory, abstraction) -> AbstractMemory:
    ...
    if M_abs == abstraction.BOT:
        return M_abs
    ...
```
\([\text{skip}]_\mathcal{D}^\#(M^\#) = M^\#\)

In code:

```python
def evaluate_Cmd_abs(C: Cmd, M_abs: AbstractMemory, abstraction) -> AbstractMemory:
    ...
    # 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)
    ...
```
\[ \llbracket C_0; C_1 \rrbracket^\#(M^\#) = \llbracket C_1 \rrbracket^\#(\llbracket C_0 \rrbracket^\#(M^\#)) \]

- This seems to be intuitive, but we need to show that:
  - The concrete postcondition of \( \llbracket C_0; C_1 \rrbracket^\# \) is over-approximated by \( \llbracket C_0; C_1 \rrbracket^\# \)
  - I.e. \( \llbracket C_0; C_1 \rrbracket^\# \subseteq \gamma(\llbracket C_0; C_1 \rrbracket^\#) \)

Theorem: Approximation of Compositions: Let \( F_0, F_1 : \mathcal{P}(M) \to \mathcal{P}(M) \) be two monotone functions that are overapproximated by \( F_0^\#, F_1^\# : \mathbb{A} \to \mathbb{A} \), i.e. \( F_0 \circ \gamma \subseteq \gamma \circ F_0^\# \) and \( F_1 \circ \gamma \subseteq \gamma \circ F_1^\# \). Then, \( F_0 \circ F_1 \) can be approximated by \( F_0^\# \circ F_1^\# \).
def evaluate_Cmd_abs(C: Cmd, M_abs: AbstractMemory, abstraction) -> AbstractMemory:
    ...
    elif isinstance(C, Seq):
        return evaluate_Cmd_abs(C.cmd1, evaluate_Cmd_abs(C.cmd0, M_abs, abstraction), abstraction)
    ...
    ...
\[ [E]^\# : \mathbb{A} \rightarrow \mathbb{A}_{\Psi} \]
\[ [n]^\#(M^\#) = \phi_{\Psi}(n) \]
\[ [x]^\#(M^\#) = M^\#(x) \]
\[ [E_0 \odot E_1]^\#(M^\#) = f^\#( [E_0]^\#(M^\#), [E_1]^\#(M^\#) ) \]

- \( \phi_{\Psi} \) can be replaced by \( \alpha_{\Psi} \) if it exists.
- Otherwise just return an abstract element such that \( \{n\} \subseteq \gamma(\phi_{\Psi}(n)) \)
\[ \forall n^\#, n_1^\# \in A^\#, \{ f_{\circ}(n_0, n_1) | n_0 \in \gamma_{\#}(n_0^\#) \text{ and } n_1 \in \gamma_{\#}(n_1^\#) \} \subseteq \gamma_{\#}(f_{\circ}(n_0^\#, n_1^\#)) \]

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

Examples (using signs):

- \( f_+^\#([\geq 0], [\geq 0]) = [\geq 0] \)
- \( f_+^\#([\geq 0], [\leq 0]) = \top \)
Expressions: $f^\#$ in code (Signs)

```python
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}')
```

- $f^\#$ is per abstract domain (not per language as in the concrete semantics)
See \texttt{f\_binop} in \texttt{dom\_intervals.py}.

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

\[
[x := E]_\mathcal{D}(M) = \{ m[x \mapsto [E](m)] \mid m \in M \}
\]

The abstract semantics are:

\[
[x := E]_\mathcal{A}(M^\#) = M^\#[x \mapsto [E]^\#(M^\#)]
\]

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

\[
[input(x)]_\mathcal{A}(M^\#) = M^\#[x \mapsto \top_V]
\]

Recall that `input` can return any value from the user.
def evaluate_Cmd_abs(C: Cmd, M_abs: AbstractMemory, abstraction) -> AbstractMemory:

def update_abs_memories(var, value_lambda):
    out = dict(M_abs)
    out[var] = value_lambda(M_abs)
    return out

...

eelif isinstance(C, Assign):
    return update_abs_memories(C.left.name, lambda m: evaluate_Expr_abs(C.right, m, v_abs))

eelif 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 $\mathcal{F}_B$
  - Its effects are shown
- We need to join the abstract elements:
  - Use the lub (least upper bound), here $\sqcup$
- But we have lost precision for $x$!
must refine

```plaintext
# 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 $M^#$ is refined to $\bot$, by setting all variables to $\bot$
  - Recall that $\llbracket C \rrbracket^#_P (\bot) = \bot$ and that $a \sqcup^# \bot = a$
Soundness properties

For $\mathcal{F}_B^\#: $ For all $B$ and abstract states $M^\#

\mathcal{F}_B(\gamma(M^\#)) \subseteq \gamma(\mathcal{F}_B^\#(M^\#))$

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

$\gamma(M_0^\#) \cup \gamma(M_1^\#) \subseteq \gamma(M_0^\# \sqcup^\# M_1^\#)$
Abstract Semantics of If

\[
\begin{align*}
\llbracket \text{if}(B)\{C_0\} \text{ else } \{C_1\}\rrbracket_\mathcal{D}(M^\#) &= \llbracket C_0\rrbracket_\mathcal{D}(\mathcal{F}_B^\#(M^\#)) \sqcup \llbracket C_1\rrbracket_\mathcal{D}(\mathcal{F}_{\neg B}^\#(M^\#))
\end{align*}
\]

Code:

```python
def evaluate_Cmd_abs(C: Cmd, M_abs: AbstractMemory, abstraction) -> AbstractMemory:
    ...
    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
```
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
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}')
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 domain)
    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

\[
x := 0
\]
\[
\text{while}(x >= 0) \quad \{
\quad x := x + 1
\quad \}
\]

If we analyze this program abstractly using signs, using \(\sqcup\) 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.

- \(M^\#(x) = ([= 0] \sqcup^\# [>= 0] \sqcup^\# [>= 0]) = [>= 0]\)

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

- \(M^\#(x) = [0, 0] \sqcup^\# [1, 1] \sqcup^\# [2, 2] \sqcup^\# [3, 3]...\)
While: Example #2: Infinite Loop

```cpp
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

- \( M^\#(x) = ([= 0] \sqcup^\# [>= 0] \sqcup^\# [>= 0]) = [>= 0] \)

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

- \( M^\#(x) = [0, 0] \sqcup^\# [0, 1] \sqcup^\# [0, 2] \sqcup^\# ... \sqcup^\# [0, 50] \sqcup^\# [0, 50] = [0, 50] \)
Observations

- Signs is a lattice with a finite height
  - $\sqcup\#$ will eventually reach a fix point
- The Intervals lattice does not have a finite height
  - No such guarantees
Define an operator \( \nabla \) 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:

- \( a'_0 = a_0 \)
- \( a'_{n+1} = a'_n \nabla a_n \)
```python
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
    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

\[
\left[ \text{while}(B)\{C\}\right]_{\mathcal{P}}^{#}(M^#) = \mathcal{F}^#_{\neg B}(\text{abs\_iter}(\left[ C \right]_{\mathcal{P}}^{#} \circ \mathcal{F}^#_{B}, M^#))
\]

Code:

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

    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

    ...
```
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
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