Introduction to program analysis

Material covered in chapter 2 of
Introduction to Static Analysis: an Abstract Interpretation Perspective

Purpose of this lecture

We aim at describing **the core concepts of static analysis by abstract interpretation, in an intuitive manner**:

1. concrete semantics
2. abstraction
3. abstract interpretation of basic program commands
4. abstract iteration and widening (to analyze loops)

This presentation is done using a small language where programs describe sequences of transformations

No background required!

Outline

Syntax

Intuition:

- imperative programs, with a graphical interpretation
- a state is a point in the two-dimensional plane (think of a pair of variables $\mathtt{x}, \mathtt{y}$)
- starting point in a given region
- basic operations are translations and rotations
- choices (conditions and loop iteration numbers) are non deterministic

Syntax

p	$::=$	$\mathtt{init}(\mathfrak{R})$	initialization, with a state in $\mathfrak{R}$
	$\mid$	$\mathtt{translation}(u, v)$	translation by vector (u, v)
	$\mid$	$\mathtt{rotation}(u, v, \theta)$	rotation with center (u, v) and angle θ
	$\mid$	$\mathtt{p}\ ;\ \mathtt{p}$	sequence of operations
	$\mid$	$\{\mathtt{p}\}\mathtt{or}\{\mathtt{p}\}$	choice
	$\mid$	$\mathtt{iter}\{\mathtt{p}\}$	iteration

States and executions

- A **program state** (or state) is a **point in the 2D field**
- A **program execution** is defined by a sequence of states

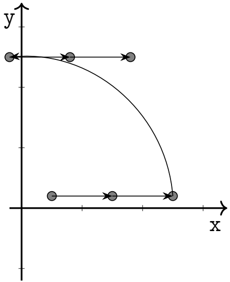
A basic program:

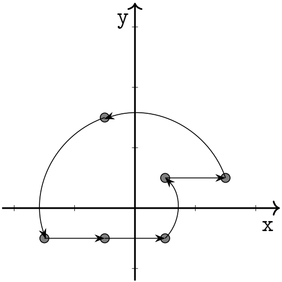
```
init([0,1] × [0,1]);
translation(1,0);
iter{
  {
    translation(1,0)
  }or{
    rotation(0,0,90°)
  }
}
```

States and executions

- A **program state** (or state) is a **point in the 2D field**
- A **program execution** is defined by a sequence of states

Example executions:





Note: left execution is terminating; right execution is non-terminating

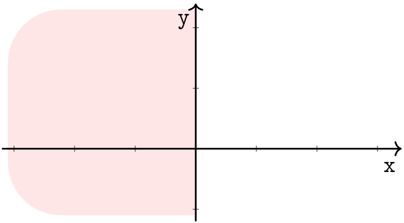
A verification problem

In this lecture, we fix a very simple target semantic property:

Property to verify

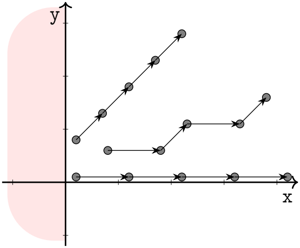
States in a given zone $\mathfrak{D}$ should not be reached by any execution

Example: $\mathfrak{D} = \{(x, y) \mid x < 0\}$

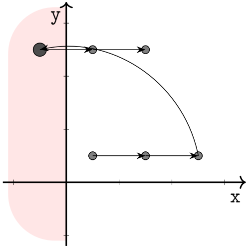


Correct executions and incorrect execution

Some **correct executions**:



An **incorrect execution**:



Our goal

Set up a **static analysis** (no execution of the program required) to **detect and report all possible incorrect executions**

Outline

Notion of abstraction: example of signs

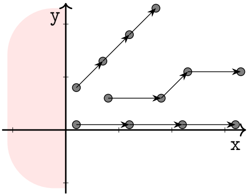
Observation:

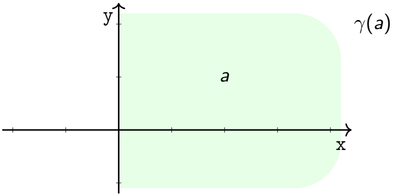
- sets of points contain far more information than necessary
- as a first step, we may retain only the **signs of variables**

Abstraction principle

Use **predicates** a which describe sets of points noted $\gamma(a)$

Example, using only **sign predicates**:



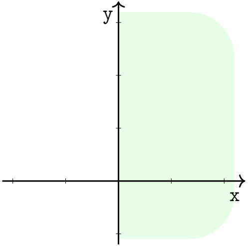


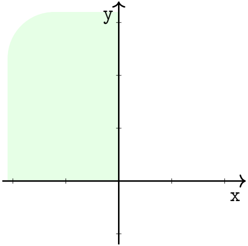
Abstraction with signs

Abstract domain definition

- **Predicates**: one sign predicate per variable
  nonpositive, zero, or nonnegative
- **Representation**: enum type with three values

Example abstract states:





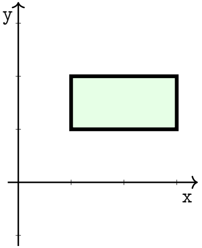
Abstraction with boxes

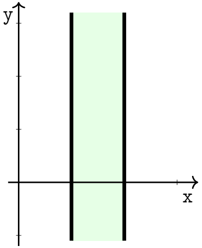
Limitation of signs: **cannot deal with simple translations precisely**

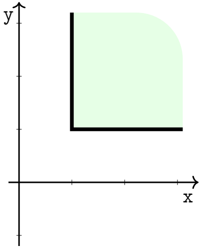
Abstract domain definition

- **Predicates**: a range for each variable
  i.e., a pair of bounds
- **Representation**: two values per variable

Example abstract states:







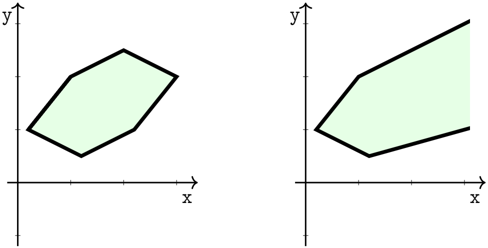
Abstraction with polygons

Limitation of boxes: **cannot express any relational constraint**

Abstract domain definition

- **Predicates**: a conjunction of linear inequality constraints
- **Representation**: either inequalities
  or geometric view (edges + vertices)

Example abstract states:



Outline

Goal of the analysis

Given a program P, compute an abstract element a such that **the set of all reachable states of P is included in $\gamma(a)$**. Such an a is a **sound abstraction**.

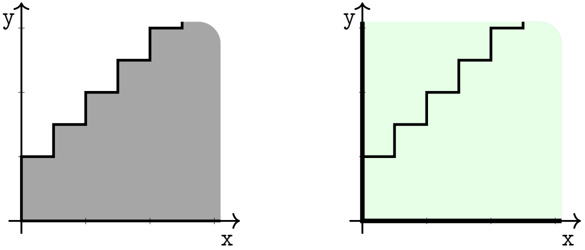
A basic program:

```
init([0,1] × [0,1]);
iter{
    {
        translation(1,0);
    }or{
        translation(0.5,0.5);
    }
}
```

Goal of the analysis

Given a program P, compute an abstract element a such that **the set of all reachable states of P is included in $\gamma(a)$**. Such an a is a **sound abstraction**.

Reachable states (exact set) and **a sound abstraction**:



Principle of the analysis

Very similar to an **interpreter**, but based on **abstract states**:

1. start with an over-approximation of initial states
2. consider the program operations in sequence
   for each operation, compute an over-approximate effect
   all on abstract states

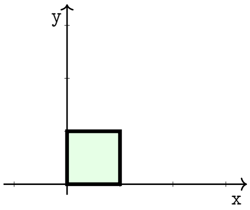
We now consider the main program operations...

Analysis of initialization

Program start with a random initialization command $\texttt{init}(\mathfrak{R})$.
How to analyze its effect?

- produce **any abstract state a such that $\mathfrak{R} \subseteq \gamma(a)$**

Example for $\texttt{init}([0,1] \times [0,1])$;



- note that the choice of a is not unique...
- ... but smaller is better: **more precise abstraction = tighter fit**

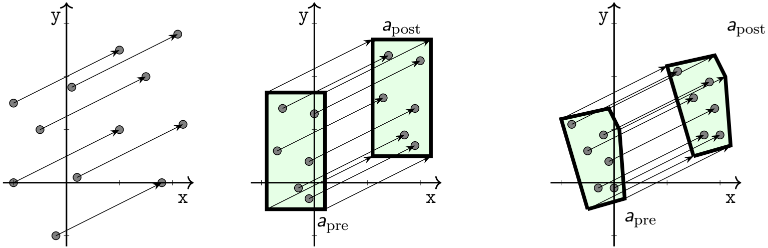
Analysis of a translation

Command `translation`(u, v) transforms a state into another.

The analysis should also describe a transformation, but over abstract states.

- the analysis **returns an abstract state containing the translation of the input abstract state, by the same vector** u, v

Over **intervals** and over **convex polyhedra**:



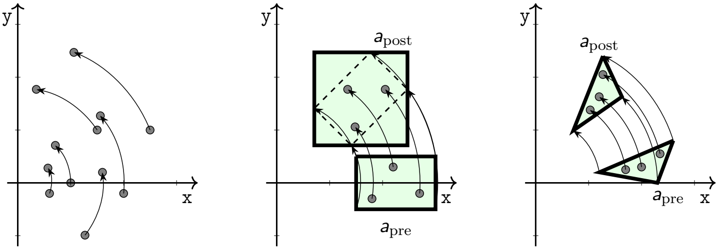
- **soundness**: forget no real behaviors
- **completeness**: no "noise" added, tight abstract post-condition

Analysis of a rotation

Command `rotation`(u, v, θ) also transforms a state into another, hence so does its analysis.

- the analysis **returns an abstract state containing the rotation of the input abstract state, with the same angle, origin parameters**

Over **intervals** and over **convex polyhedra**:



- **soundness**: forget no real behaviors
- **unavoidable imprecision** with intervals, but not with polyhedra

Outline

Principle and analysis of a basic program

Status so far:

- for **initialization**: produce a state that over-approximates the initial states
- for **basic command** p: a function f_{p} that maps an abstract state (set of input states) to an over-approximate abstract state (super-set of output states)

Can we generalize this for composite commands ?

Easy for sequence commands:

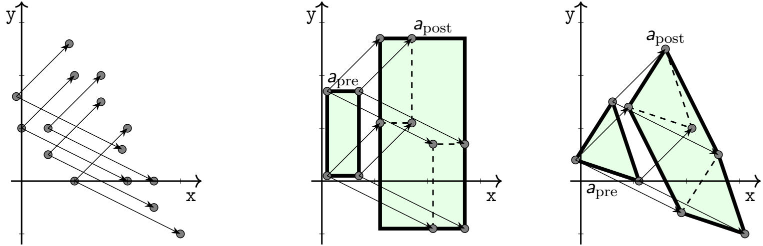
$$f_{\mathrm{p}_0;\mathrm{p}_1} = f_{\mathrm{p}_1} \circ f_{\mathrm{p}_0}$$

Analysis of a choice

Command $\{\mathtt{p}_0\}\mathtt{or}\{\mathtt{p}_1\}$ boils down to non-deterministic choice + standard execution.

1. analyze both $\mathtt{p}_0$ and $\mathtt{p}_1$
2. compute an over-approximation of the results of these analyses typically, by a **abstract union**/**convex closure** algorithm

Over **intervals** and over **convex polyhedra**:



- **convex closure** typically loses a lot of precision

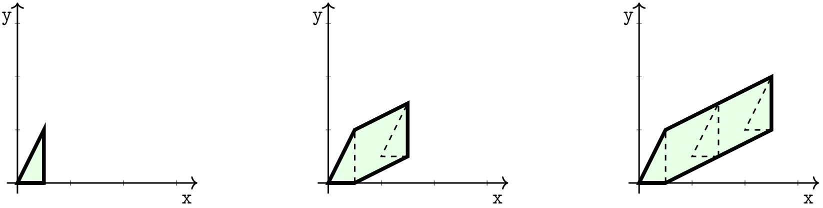
Analysis of a loop: a few iterates

A first attempt: rewriting a loop using choice and sequence

$$\mathtt{iter}\{\mathtt{p}\} \qquad \text{is equivalent to} \qquad \left\{ \begin{array}{l} \{\} \\ \mathtt{or}\{\mathtt{p};\} \\ \mathtt{or}\{\mathtt{p};\mathtt{p};\} \\ \mathtt{or}\ldots \end{array} \right.$$

Example

$\mathtt{init}(\{(\mathtt{x},\mathtt{y}) \mid 0 \leq \mathtt{y} \leq 2\mathtt{x} \text{ and } \mathtt{x} \leq 0.5\}); \mathtt{iter}\{\mathtt{translation}(1, 0.5)$
using convex polyhedra, and covering just a few iterations:



Issue: algorithm unclear to compute this sequence!

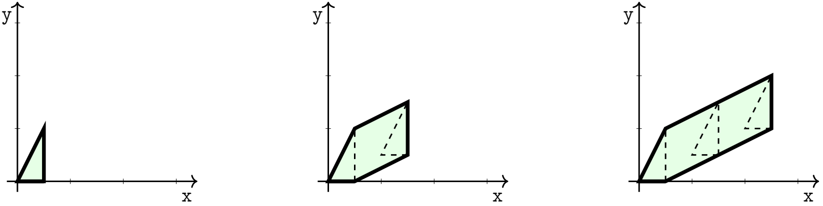
Analysis of a loop: use of widening

Another approach:

- we note $\mathtt{p}_0$ for $\{\}$, $\mathtt{p}_1$ for $\{\}\mathtt{or}\{\mathtt{b}\}$, $\mathtt{p}_2$ for $\{\}\mathtt{or}\{\mathtt{b}\}\mathtt{or}\{\mathtt{b};\mathtt{b}\}$ and so on;
- we remark that $\mathtt{p}_{k+1}$ is equivalent to $\mathtt{p}_k\mathtt{or}\{\mathtt{p}_k;\mathtt{b}\}$
- thus, we can do an iterative analysis

$$\mathtt{analysis}(\mathtt{p}_{k+1}, a) = \mathtt{union}(\mathtt{analysis}(\mathtt{p}_k, a), \mathtt{analysis}(\mathtt{b}, \mathtt{analysis}(\mathtt{p}_k, a)))$$

Same example, with an algorithm to compute the iterations:



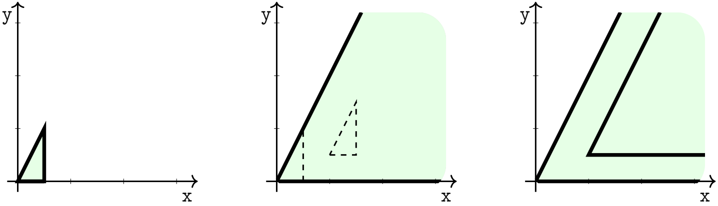
Issue: termination not guaranteed!

Analysis of a loop: use of widening

Let us **speed up convergence**:

- termination follows from replacing `union` with a coarser **widening** operation such that all such sequence terminates
- typical widening technique: let $\texttt{widen}(a_0, a_1)$ return a conjunction of constraints that retains only constraints in a_0 that still hold in a_0; starting from finitely many constraints, termination is guaranteed

Same example:



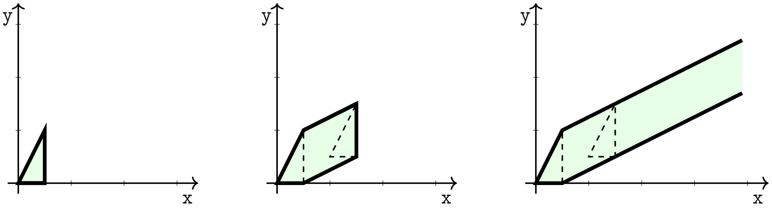
Issue: precision is not all that great...

Analysis of a loop: use of widening and unrolling

Solution: **combine regular union and widening**

- first iteration with `union`
- next iterations using `widen`

Same example:



Outline

Semantic style: compotional versus transitional

- compositional semantics function:
  - semantics of p is defined by the semantics of the sub-parts of p.

$$[\![AB]\!] = \cdots [\![A]\!] \cdots [\![B]\!] \cdots$$

  - proving its soundness is thus by structural induction on p.
- for some realistic programming languages, defining their compositional (“denotational”) semantics is a hurdle.
  - function calls, exceptions, gotos, functions and jump labels as values

Transitional-style (“operational”) semantics avoids the hurdle

$$[\![AB]\!] = \{s_0 \hookrightarrow s_1 \hookrightarrow \cdots, \cdots\}$$

Semantics as state transitions

Definition (Transitional semantics)

An execution of a program is a sequence of transitions between states.

- a state is a pair (l, p) of statement label l and an (x,y) point p.
- a single transition

$$(l, p) \hookrightarrow (l', p')$$

whenever the program statement at l moves the point p to p'.

$$s_1 \hookrightarrow s_2 \hookrightarrow s_5 \hookrightarrow s_3 \hookrightarrow s_8 \hookrightarrow \cdots$$
$$s_6 \hookrightarrow s_7 \hookrightarrow s_8 \hookrightarrow s_3 \hookrightarrow s_4$$
$$s_9 \hookrightarrow s_{10} \hookrightarrow s_8 \hookrightarrow s_{11} \hookrightarrow s_8 \hookrightarrow s_{11} \hookrightarrow s_{13}$$
$$s_{12} \hookrightarrow s_7 \hookrightarrow s_2 \hookrightarrow s_3 \hookrightarrow s_4 \hookrightarrow s_{14}$$

States s_1, s_6, s_9, and s_{12} are initial states.

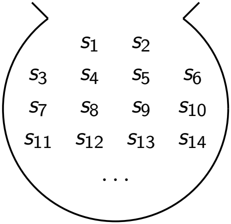


Figure: Transition sequences and the set of occurring states

Example language, again

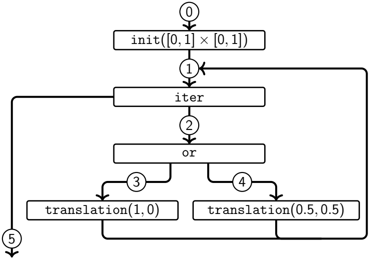
$$
\begin{array}{rcll}
\mathtt{p} & ::= & \mathtt{init}(\mathfrak{R}) & \text{initialization, with a state in } \mathfrak{R} \\
& | & \mathtt{translation}(u, v) & \text{translation by vector } (u, v) \\
& | & \mathtt{rotation}(u, v, \theta) & \text{rotation by center } (u, v) \text{ and angle } \theta \\
& | & \mathtt{p}\ ;\ \mathtt{p} & \text{sequence of operations} \\
& | & \{\mathtt{p}\}\mathtt{or}\{\mathtt{p}\} & \text{non-deterministic choice} \\
& | & \mathtt{iter}\{\mathtt{p}\} & \text{non-deterministic iterations}
\end{array}
$$

We will consider a more dynamic language when covering chapter 4 later.

Statement labels

```
(0) init([0,1] × [0,1]);
(1) iter{
      (2) {
            (3) translation(1,0);
          }or{
            (4) translation(0.5,0.5);
          }
    }
(5)
```

(a) Text view, with labels

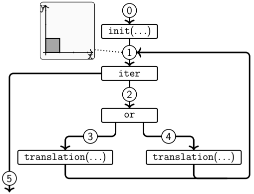


(b) Graph view, with labels

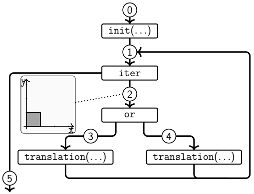
Figure: Example program with statement labels

States in a transition sequence

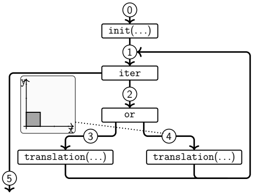
State $(1, p_1)$



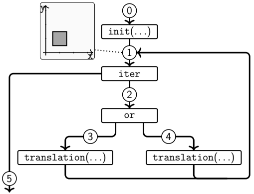
State $(2, p_1)$



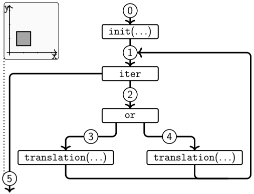
State $(4, p_1)$



State $(1, p_3)$



State $(5, p_3)$



Reachability problem and abstraction of states

Reachability problem:

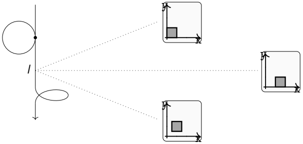
- we are interested in the set of all states that can occur during all transition sequences of the input program.

An abstract state is:

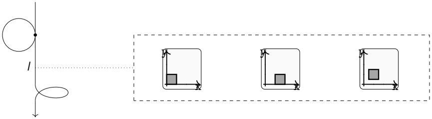
- a set of pairs of statement labels and abstract pre conditions.

Statement-wise abstraction of reachable states

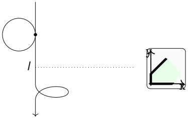
Collection of all states



Statement-wise collection:



Statement-wise abstraction:



Abstract state transition

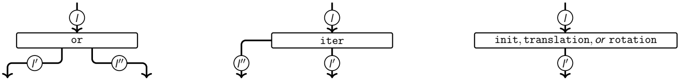
$$Step^\sharp : \begin{array}{l} \text{a set of pairs of labels and abstract pre conditions} \\ \mapsto \\ \text{a set of pairs of labels and abstract post conditions} \end{array}$$

is

$$Step^\sharp(X) = \{x' \mid x \in X, x \hookrightarrow^\sharp x'\}$$

where

$$\begin{array}{rcl} (\mathtt{or}_l, a_{\mathrm{pre}}) & \hookrightarrow^\sharp & (\mathtt{next}(l), a_{\mathrm{pre}}) \\ (\mathtt{iter}_l, a_{\mathrm{pre}}) & \hookrightarrow^\sharp & (\mathtt{next}(l), a_{\mathrm{pre}}) \\ (\mathtt{p}_l, a_{\mathrm{pre}}) & \hookrightarrow^\sharp & (\mathtt{next}(l), \mathtt{analysis}(\mathtt{p}_l, a_{\mathrm{pre}})) \end{array}$$



Analysis by global iterations

The analysis goal is to accumulate from the initial abstract state I:

$$\mathit{Step}^{\sharp^0}(I) \cup \mathit{Step}^{\sharp^1}(I) \cup \mathit{Step}^{\sharp^2}(I) \cup \cdots$$

which is the limit C_∞ of $C_i = \mathit{Step}^{\sharp^0}(I) \cup \mathit{Step}^{\sharp^1}(I) \cup \cdots \cup \mathit{Step}^{\sharp^i}(I)$ where

$$C_{k+1} = C_k \cup \mathit{Step}^\sharp(C_k).$$

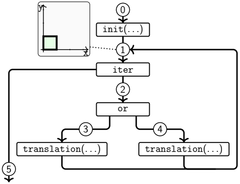
Thus the analysis algorithm should iterate the operation $\mathtt{C} \leftarrow \mathtt{C} \cup \mathit{Step}^\sharp(\mathtt{C})$ from I until stable:

$$\mathtt{analysis}_{\mathcal{T}}(\mathtt{p}, I) = \begin{cases} \mathtt{C} \leftarrow I \\ \text{repeat} \\ \qquad \mathtt{R} \leftarrow \mathtt{C} \\ \qquad \mathtt{C} \leftarrow \mathtt{widen}_{\mathcal{T}}(\mathtt{C}, \mathit{Step}^\sharp(\mathtt{C})) \\ \text{until } \mathtt{inclusion}_{\mathcal{T}}(\mathtt{C}, \mathtt{R}) \\ \text{return } \mathtt{R} \end{cases}$$

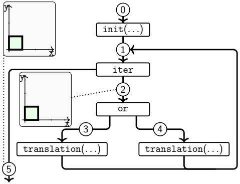
where $\mathtt{widen}_{\mathcal{T}}$ over-approximates unions and enforces finite convergence.

Analysis in action

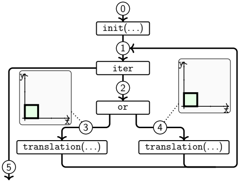
State $(1, a_{\mathbf{1}})$



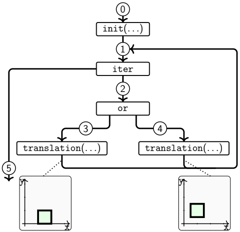
States $(2, a_{\mathbf{1}})$ and $(5, a_{\mathbf{1}})$



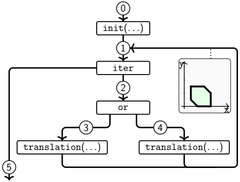
States $(3, a_{\mathbf{1}})$ and $(4, a_{\mathbf{1}})$



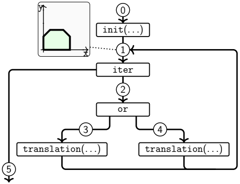
States $(1, a_{\mathbf{2}})$ and $(1, a_{\mathbf{3}})$



State $(1, \mathrm{union}(\{a_{\mathbf{2}}, a_{\mathbf{3}}\}))$



State $(1, \mathrm{union}(\{a_{\mathbf{1}}, a_{\mathbf{2}}, a_{\mathbf{3}}\}))$



Outline

Important points to remember, and what to learn next

A quick **summary** of the approach that we followed:

1. start from a **semantics**, describing program behaviors
2. set up an **abstraction**, that defines a set of logical predicates and a machine representation
3. seek for **analysis algorithms**
   computation of abstract post-conditions,
   abstract union and widening...
4. set up an iteration algorithm: **compositional** or **transitional**

Next lectures:

formalize these steps and
provide step-by-step frameworks for designing sound static analyses