Taken from Lecture III of David Schmidt.
Outline

1. Small-step semantics: trace generation
2. State generation and collecting semantics
3. Data-flow analysis
4. Ensuring termination
5. Typing rules and big-step semantics
6. Interprocedural analysis
A **static analysis** of a program is a **sound, finite, and approximate** calculation of the program’s execution semantics.

**Approximate:** not exact — computes properties or aspects of the execution semantics, such as pre- or post-conditions, invariants, data types, patterns of trace, or ranges-of-values.

**Sound:** consistent with the concrete, execution semantics — a sound **overapproximation** describes a superset of the program’s executions (**safe descriptions**); a sound **underapproximation** describes a subset of the program’s executions (**live descriptions**). We will focus on overapproximations.

**Finite:** regardless of the program and its approximate semantics, the analysis terminates.
The most basic static analysis is *trace generation*

\[ p_0 : \text{ while } (x != 1) \{ \]
\[ p_1 : \text{ if Even}(x) \]
\[ p_2 : \text{ then } x = x \text{ div}2; \]
\[ p_3 : \text{ else } x = 3*x + 1; \]
\[ \} \]
\[ p_4 : \text{ exit} \]

**Note:** \( p_i, v \) abbreviates \( p_i, \langle x : v \rangle \)

Abstract overapproximating trace:

- \( p_0, \text{ even} \)
- \( p_1, \text{ even} \)
- \( p_2, \text{ even} \)
- \( p_0, \text{ any} \)
- \( p_4, \text{ odd} \)
- \( p_3, \text{ odd} \)
- \( p_1, \text{ any} \)

Two concrete traces:

- \( p_0,4 \)
- \( p_1,4 \)
- \( p_2,4 \)
- \( p_0,2 \)
- \( p_1,2 \)
- \( p_2,2 \)
- \( p_0,1 \)
- \( p_4,1 \)
- \( p_0,6 \)
- \( p_1,6 \)
- \( p_2,6 \)
- \( p_0,3 \)
- \( p_1,3 \)
- \( p_2,3 \)
- \( p_0,10 \)
- \( p_4,1 \)

The abstract tree is a static analysis of those concrete executions that use an even-valued input.
Each concrete transition, \( p_i, s \rightarrow p_j, f_i(s) \), is reproduced by a corresponding abstract transition, \( p_i, a \rightarrow p_j, f_i^\#(a) \), where \( s \in \gamma(a) \). (\( f_i^\# = \alpha \circ f_i \circ \gamma \).)

The traces embedded in the abstract trace tree simulate all the concrete traces, e.g., this concrete trace,

\[
p_0, 4 \rightarrow p_1, 4 \rightarrow p_2, 4 \rightarrow p_0, 2 \rightarrow p_1, 2 \rightarrow p_2, 2 \rightarrow p_0, 1 \rightarrow p_4, 1
\]

is simulated by this abstract trace, which is extracted from the abstract computation tree:

\[
p_0, \text{even} \rightarrow p_1, \text{even} \rightarrow p_2, \text{even} \rightarrow p_0, \text{even} \rightarrow p_1, \text{even} \rightarrow p_2, \text{even} \rightarrow p_0, \text{odd} \rightarrow p_4, \text{odd}
\]

because we used a Galois connection to justify the soundness of the transition steps in the abstract trace tree.

In this fashion, a static analysis can generate an abstract test or abstract model, which covers a range of concrete inputs.
State reachability and collecting semantics

If we are interested only in the reachable states and not their orderings in the trace, we compute the program’s collecting semantics as a nondecreasing sequence of sets of program states. State-reachability semantics is an abstraction of trace-generation semantics.

State-reachability semantics, concrete and abstract:

\[
\begin{align*}
\{p_0, 4\} & \quad \{p_0, \text{even}\} \\
\{p_0, 4; p_1, 4\} & \quad \{p_0, \text{even}; p_4, \text{even}; p_1, \text{even}\} \\
\{p_0, 4; p_1, 4; p_2, 4\} & \quad \{p_0, \text{even}; p_4, \text{even}; p_1, \text{even}; p_2, \text{even}\} \\
\{p_0, 4; p_1, 4; p_2, 4; p_0, 2\} & \quad \{p_0, \text{even}; p_4, \text{even}; p_1, \text{even}; p_2, \text{even}; p_0, \text{any}\} \\
\ldots & \quad \ldots \\
\{p_0, 4; p_1, 4; p_2, 4; p_0, 2; p_1, 2; p_2, 2; p_0, 1; p_4, 1\} & \quad \{p_0, \text{even}; p_4, \text{even}; p_1, \text{even}; p_2, \text{even}; p_0, \text{any}; p_4, \text{any}; p_1, \text{any}; p_3, \text{odd}\}
\end{align*}
\]
“Sticky” collecting semantics

A semantics of form, $\varphi(\text{ProgramPoint} \times \text{AbsStore})$, is “attaching” AbsStore values to each program point — the isomorphic representation, $\text{ProgramPoint} \rightarrow \varphi(\text{AbsStore})$, is called the (relational) “sticky” collecting semantics:

$$\begin{align*}
[p_0 \mapsto \{\text{even, any}\}; p_1 \mapsto \{\text{even, any}\}; p_2 \mapsto \{\text{even}\};
p_3 \mapsto \{\text{odd}\}; p_4 \mapsto \{\text{even, any}\}]\end{align*}$$

The above can be abstracted to a function in $\text{ProgramPoint} \rightarrow \text{AbsStore}$, the independent-attribute semantics:

$$\begin{align*}
[p_0 \mapsto \text{any}; p_1 \mapsto \text{any}; p_2 \mapsto \text{even}; p_3 \mapsto \text{odd}; p_4 \mapsto \text{any}]\end{align*}$$

which is based on this abstraction mapping:

$$\alpha : \varphi(\text{AbsStore}) \rightarrow \text{AbsStore}$$

$$\alpha(S) = \langle i : \bigsqcup_{s \in S} s(i) \rangle_{i \in \text{Identifier}}$$
Notice that the independent-attribute semantics is less precise than its relational ancestor; for example, variables $x$ and $y$ might have these values at program point $p_i$:

\[\ldots p_i \mapsto \{\langle x : \text{even}, y : \text{even} \rangle, \langle x : \text{odd}, y : \text{odd} \rangle\}\ldots\]

meaning that $x + y$ computes to \textit{even} at $p_i$.

But the independent-attribute abstraction,

\[\ldots p_i \mapsto \langle x : \text{any}, y : \text{any} \rangle\ldots\]

makes $x + y$ compute to \textit{any}, losing precision.

Note also that we could define a collecting version of a trace-generation semantics, which generates an analysis of form $\text{ProgramPoint} \rightarrow \wp(\text{Trace})$. 
Formalizing the “small steps”: transfer functions

The transitions, $p_i, s \rightarrow p_i, s'$, are computed by means of a control-flow graph annotated with transfer functions.

\[
\begin{align*}
p_0 &: \quad y = 1; \\
p_1 &: \quad \text{while Even}(x) \{ \\
& \quad p_2 : \quad y = y \times x; \\
& \quad p_3 : \quad x = x \div 2; \\
& \} \\
p_4 &: \quad \text{exit}
\end{align*}
\]

Concrete transfer functions:

\[
\begin{align*}
f_0(x : u, y : v) &= (x : u, y : 1) \\
f_{1t}(s) &= \begin{cases} s & \text{if } s = (x : 2u, y : v) \\ \bot & \text{otherwise} \end{cases} \\
f_{1f}(s) &= \begin{cases} s & \text{if } s = (x : 2u + 1, y : v) \\ \bot & \text{otherwise} \end{cases} \\
f_2(x : u, y : v) &= (x : u, y : v \times u) \\
f_3(x : u, y : v) &= (x : u/2, y : v)
\end{align*}
\]

Note: configurations of form, $p_i, \bot$, cannot appear in a trace.
The *abstract transfer functions* are derived as $f^\# = \alpha \circ f \circ \gamma$

$$
\begin{align*}
    p_0 & : y = 1; \\
    p_1 & : \text{while Even}(x) \{ \\
        & p_2 : y = y \times x; \\
        & p_3 : x = x \div 2; \\
    \}
    p_4 & : \text{exit}
\end{align*}
$$

As usual, $\langle u, v \rangle$ abbreviates $\langle x : u, y : v \rangle$

Note: all $f^\#$ are *totally strict*: $f^\# \langle u, \bot \rangle = f^\# \langle \bot, v \rangle = \langle \bot, \bot \rangle$

$$
\begin{align*}
    f^\#_0 \langle u, v \rangle &= \langle u, \text{odd} \rangle \\
    f^\#_1 s &= s \cap \langle \text{even}, \top \rangle \\
    f^\#_2 \langle u, v \rangle &= \langle u, w \rangle, \text{ where } w = \begin{cases} \\
        \text{even} & \text{if } u = \text{even} \text{ or } v = \text{even}, \text{ else} \\
        \text{odd} & \text{if } u = \text{odd} \text{ and } v = \text{odd}, \text{ else} \\
        \top & \text{otherwise}
    \end{cases} \\
    f^\#_3 \langle u, v \rangle &= \langle \top, v \rangle
\end{align*}
$$

Note: $\langle a, b \rangle \cap \langle a', b' \rangle = \langle a \cap a', b \cap b' \rangle$. 
Flow equations calculate the collecting semantics

The value “attached” to program point $p_i$ is defined by the equational pattern,

$$p_i\text{Store} = \bigsqcup_{p_j \in \text{pred}(p_i)} f^\#(p_j\text{Store})$$

The collecting semantics of $p_i$ is the join of the answers computed by $p_i$’s predecessor transfer functions.

Flow equations for previous example:

$$p_0\text{Store} = \langle x : \top, y : \top \rangle$$
$$p_1\text{Store} = f_0^\#(p_0\text{Store}) \sqcup f_3^\#(p_3\text{Store})$$
$$p_2\text{Store} = f_{1t}^\#(p_1\text{Store})$$
$$p_3\text{Store} = f_2^\#(p_2\text{Store})$$
$$p_4\text{Store} = f_{1f}^\#(p_1\text{Store})$$
We *solve* the flow equations by calculating approximate solutions in stages until *the least fixed point* is reached.

**Note:** $u, v$ abbreviates $\langle x : u, y : v \rangle$.

<table>
<thead>
<tr>
<th>stage</th>
<th>$p_0$Store</th>
<th>$p_1$Store</th>
<th>$p_2$Store</th>
<th>$p_3$Store</th>
<th>$p_4$Store</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$\perp, \perp$</td>
<td>$\perp, \perp$</td>
<td>$\perp, \perp$</td>
<td>$\perp, \perp$</td>
<td>$\perp, \perp$</td>
</tr>
<tr>
<td>1</td>
<td>$T, T$</td>
<td>$\perp, \perp$</td>
<td>$\perp, \perp$</td>
<td>$\perp, \perp$</td>
<td>$\perp, \perp$</td>
</tr>
<tr>
<td>2</td>
<td>$T, T$</td>
<td>$T$, odd</td>
<td>$\perp, \perp$</td>
<td>$\perp, \perp$</td>
<td>$\perp, \perp$</td>
</tr>
<tr>
<td>3</td>
<td>$T, T$</td>
<td>$T$, odd</td>
<td>even, odd</td>
<td>$\perp, \perp$</td>
<td>odd, odd</td>
</tr>
<tr>
<td>4</td>
<td>$T, T$</td>
<td>$T$, odd</td>
<td>even, odd</td>
<td>even, even</td>
<td>odd, odd</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>$T, T$</td>
<td>$T, T$</td>
<td>even, $T$</td>
<td>even, even</td>
<td>odd, $T$</td>
</tr>
<tr>
<td>9</td>
<td>$T, T$</td>
<td>$T, T$</td>
<td>even, $T$</td>
<td>even, even</td>
<td>odd, $T$</td>
</tr>
</tbody>
</table>

A faster algorithm uses a *worklist* that remembers exactly which equations should be recalculated at each stage.
To summarize, we annotate the control-flow graph with the non-
values that arrive at the program points:

\[
\begin{align*}
\text{\( p_0 \)} & : \quad y = 1; \\
& \quad \downarrow \quad \text{\( \top, \top \)} \\
\text{\( p_1 \)} & : \quad \text{while Even(x)} \\
& \quad \downarrow \quad \text{\( \top, \text{odd} \lor \top, \text{even} = \top, \top \)} \\
\text{\( p_2 \)} & : \quad y = y \ast x; \\
& \quad \downarrow \quad \text{\( \text{even} \land \text{odd} \lor \text{even} \land \top = \text{even} \land \top \)} \\
\text{\( p_3 \)} & : \quad x = x \text{ div} 2; \\
\text{\( p_4 \)} & : \quad \text{exit} \\
\end{align*}
\]

The analysis approximates the stores that arrive at the program points.

The equational format is called \textit{data-flow analysis}. It is the most popular static analysis format.
Variants of data-flow analysis

We might vary whether the “data flow” goes forwards or backwards; we might also vary whether information is “joined” (⊔) or “met” (⊓):

**Forwards-possibly:** initial value: \( \bot \in D \)

equation format: \( p_i \text{Store} = \sqcup_{p_j \in \text{pred}(p_i)} f_j(p_j \text{Store}) \)

computes: \( \text{lfp} \) (least fixed point)

**Forwards-necessarily:** initial value: \( \top \in D \)

equation format: \( p_i \text{Store} = \sqcap_{p_j \in \text{pred}(p_i)} f_j(p_j \text{Store}) \)

computes: \( \text{gfp} \) (greatest fixed point)

**Backwards-possibly:** initial value: \( \emptyset \in \wp(D) \)

equation format: \( p_i \text{Store} = \sqcup_{p_j \in \text{succ}(p_i)} f_j^{-1}(p_j \text{Store}) \)

computes: \( \text{lfp} \)

**Backwards-necessarily:** initial value: \( D \in \wp(D) \)

equation format: \( p_i \text{Store} = \sqcap_{p_j \in \text{succ}(p_i)} f_j^{-1}(p_j \text{Store}) \)

computes: \( \text{gfp} \)
A *forwards* analysis computes “histories” that arrive at a point:

\[
\text{forwards analysis} = \text{postcondition semantics}
\]

\[p_i \text{Store} = a\] approximates the set of traces of the form

\[p_0, s_0 \rightarrow p_1, s_1 \rightarrow \cdots \rightarrow p_i, s_i \quad (\text{where } s_i \in \gamma(a))\]

A *backwards* analysis computes the “futures” from a program point:

\[
\text{backwards analysis} = \text{precondition semantics}
\]

\[p_i \text{Store} = a\] approximates the set of traces of the form

\[p_i, s_i \rightarrow \cdots \rightarrow p_{\text{exit}}, s_{\text{final}} \quad (\text{where } s_i \in \gamma(a))\]

A *possibly* analysis predicts a “superset” of the actual computations: if a concrete value, \(c \sqsubseteq_C \gamma(a)\), arrives at \(p_i\), then we have \(p_i \text{Store} = a\) and \(c \sqsubseteq_C \gamma(a)\) — all possibilities are predicted.

A *necessarily* analysis predicts a “subset” of the actual computations: if \(p_i \text{Store} = a\) and \(c \sqsubseteq_C \gamma(a)\), then \(c\) will arrive at \(p_i\).
The data-flow example developed earlier in this Lecture computed answers of the form,

\[ p_i \text{Store} = a \]

which asserted, if store \( s \) arrives at program point \( p_i \), then \( s \in \gamma(a) \).

But there are data-flow analyses where \( p_i \text{Store} = a \) means that all execution traces that arrive at \( p_i \) contain some embedded pattern of program points and stores, described by \( a \).

We will develop the Galois-connection formalities in the next Lecture, but just now we study two examples, used by compilers for improving register allocation in target code. These examples compute sets of program phrases that describe patterns within execution traces.

The examples show variations of the forwards/backwards and possibly/necessarily forms of data-flow analysis.
Forwards-necessarily-reaching definitions: which assignments \textit{must} reach their successors

\[
\text{inReach}_{p_i} = \bigcap_{p_j \in \text{pred}(p_i)} \text{outReach}_{p_j}
\]

\[
f_i^\#(\text{inReach}_{p_i}) = \text{outReach}_{p_i} = (\text{inReach}_{p_i} - \text{kill}_i) \cup \text{gen}_i
\]

(the transfer function computes a set of assignment statements)

for \(p_i : x = e\), \(\left\{ \begin{array}{l}
\text{kill}_i = \{p_j \mid p_j : x = \ldots\}
\text{gen}_i = \{p_i\}
\end{array} \right. \)

for \(p_i : \text{if } e\), \(\left\{ \begin{array}{l}
\text{kill}_i = \{
\text{gen}_i = \{
\end{array} \right. \)

\(p_0 : x = 0\)

\(\{ p_0 \} \)

Sample analysis:

\(p_1 : \text{if } \ldots\)

\(\{ p_0 \} \rightarrow \{ p_0 \} = \text{inReach } p_3\)

\(p_2 : x = x + 1\)

\(\{ p_2 \} \rightarrow \{ p_0, p_3 \} = \text{outReach } p_3\)

\(p_3 : y = x\)

\(\{ p_2 \} \rightarrow \{ \} = \text{inReach } p_4\)

\(p_4 : \text{exit}\)
Explanation:

If $p' \in \text{inReach}_{p_i}$, where $p'$ labels the assignment, $p' : v \equiv e$, then all traces from $p_0$ to $p_i$ must possess the pattern,

$$p_0 \rightarrow \cdots \rightarrow p' \rightarrow \cdots \rightarrow p_i$$

and no assignment, $v = e'$, occurs between $p'$ and $p_i$ in the trace. If $p' \in \text{inReach}_{p_i}$ holds, then the assignment at $p'$ should save its right-hand-side value in a register for quick access by $p_i$. 
Backwards-possibly-live variables: which variables *might* be referenced in the future

\[ \text{outLive}_{pi} = \bigcup_{p_j \in \text{succ}(p_i)} \text{inLive}_{p_j} \]

\[ f^\#_i(\text{outLive}_{pi}) = \text{inLive}_{pi} = (\text{outLive}_{pi} \setminus \text{kill}_i) \cup \text{gen}_i \]

(the transfer function computes a set of variable names)

for \( p_i : x = e \) \[ \begin{align*}
  \text{kill}_i &= \{ x \} \\
  \text{gen}_i &= \{ v \mid v \text{ in } e \}
\end{align*} \]

for \( \text{print } e \) \[ \begin{align*}
  \text{kill}_i &= \{ \} \\
  \text{gen}_i &= \{ v \mid v \text{ in } e \}
\end{align*} \]

\( \{ x \} = \text{inLive } p0 \)

\( p0 : y = 1; \)

\( \{ x, y \} = \text{outLive } p0 = \text{inLive } p1 \)

Sample analysis:

\( p1 : \text{while Even}(x) \)

\( \{ y \} \)

\( p4 : \text{print } y \)

\( p2 : y = 2 \times x; \)

\( \{ x, y \} \)

\( p3 : x = x \text{ div} 2; \)

\( \{ x, y \} \)
Explanation:

If there is a concrete execution trace containing the pattern,

\[ p_i \rightarrow \cdots \rightarrow p' \rightarrow \cdots \rightarrow p_{\text{exit}} \]

such that \( p' \) references variable \( v \) and no assignment to \( v \) appears between \( p_i \) and \( p' \), then \( v \in \text{outLive}_{p_i} \).

If \( v \notin \text{outLive}_{p_i} \) holds, then \( v \)'s value should be removed from all registers upon completion of \( p_i \)'s execution — \( v \) is a "dead variable" after \( p_i \).
**Termination: Constant propagation reviewed**

\[ p_0 : x = 1; y = 2; \]
\[ p_1 : \text{while } (x < y + z) \]
\[ \quad \text{ } p_2 : x = x + 1; \]
\[ \quad \text{ } \} \]
\[ p_3 : \text{exit} \]

where \( m + n \) is interpreted

\[ k_1 + k_2 \rightarrow \text{sum}(k_1, k_2), \]
\[ \top \neq k_i \neq \bot, i \in 1..2 \]
\[ \top + k \rightarrow \top \]
\[ k + \top \rightarrow \top \]

The naive trace does not terminate.

**Abstract trace:**

- \[ p_0, \langle \top, \top, \top \rangle \]
- \[ p_1, \langle 1, 2, \top \rangle \]
- \[ p_2, \langle 1, 2, \top \rangle \]
- \[ p_1, \langle 2, 2, \top \rangle \]
- \[ p_2, \langle 2, 2, \top \rangle \]
- \[ p_1, \langle 3, 2, \top \rangle \]
- \[ p_3, \langle 1, 2, \top \rangle \]
- \[ p_3, \langle 2, 2, \top \rangle \]
- \[ p_3, \langle 2, 2, \top \rangle \]
- \[ \ldots \]

**Diagram:**

- \( Const \)
  - \(-1\)
  - \(0\)
  - \(1\)
  - \(2\)

- \( \text{var holds multiple values} \)
- \( \text{var holds this value only} \)
- \( \text{var holds no value (dead code)} \)
**Finite-height and \( \sqsubseteq \) give termination**

<table>
<thead>
<tr>
<th>stage</th>
<th>( p_0 \text{Store} )</th>
<th>( p_1 \text{Store} )</th>
<th>( p_2 \text{Store} )</th>
<th>( p_3 \text{Store} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( \top, \top, \top )</td>
<td>( \bot, \bot, \bot )</td>
<td>( \bot, \bot, \bot )</td>
<td>( \bot, \bot, \bot )</td>
</tr>
<tr>
<td>2</td>
<td>( \top, \top, \top )</td>
<td>( 1, 2, \top )</td>
<td>( \bot, \bot, \bot )</td>
<td>( \bot, \bot, \bot )</td>
</tr>
<tr>
<td>3</td>
<td>( \top, \top, \top )</td>
<td>( 1, 2, \top )</td>
<td>( 1, 2, \top )</td>
<td>( 1, 2, \top )</td>
</tr>
<tr>
<td>4</td>
<td>( \top, \top, \top )</td>
<td>( \top, 2, \top )</td>
<td>( 1, 2, \top )</td>
<td>( 1, 2, \top )</td>
</tr>
<tr>
<td>5</td>
<td>( \top, \top, \top )</td>
<td>( \top, 2, \top )</td>
<td>( \top, 2, \top )</td>
<td>( \top, 2, \top )</td>
</tr>
<tr>
<td>6</td>
<td>( \top, \top, \top )</td>
<td>( \top, 2, \top )</td>
<td>( \top, 2, \top )</td>
<td>( \top, 2, \top )</td>
</tr>
</tbody>
</table>

Termination is **guaranteed** because the transfer functions and \( \sqsubseteq \) are monotonic (each stage has values not smaller than its predecessors) and the abstract domain, \( \text{Const} \), has **finite height** — there are no infinitely ascending sequences (the stages cannot increase forever).

(Indeed, the longest sequence in \( \text{Const} \) goes: \( \bot \subseteq k \subseteq \top \).)
**Termination:** Array-bounds checking reviewed

Integer variables receive values from the *interval domain*,

\[ I = \{ [i, j] \mid i, j \in \text{Int} \cup \{-\infty, +\infty\} \}. \]

We define \([a, b] \sqcup [a', b'] = [\min(a, a'), \max(b, b')]\).

```java
int a = new int[10];
i = 0;  // i = [0,0]
while (i < 10) {
    ... a[i] ...
    i = i + 1;
}
```

This example terminates: \(i\)'s ranges are

- at \(p_1: [0..9]\)
- at \(p_2: [1..10]\)
- at loop exit: \([1..10] \sqcup [10, +\infty] = [10, 10]\)

[Note: The notation and syntax are for demonstration purposes and may not be directly executable.]
But others might not, because the domain is not finite height:

\[
\begin{align*}
&i = 0; \quad \downarrow[0,0] \\
&\text{while true } \\&\quad \rightarrow \quad i = [0,0] \uplus [1,1] \uplus [2,2] \ldots \downarrow \text{infinite limit is } [0, +\infty] \\
&\quad i = i + 1; \quad \downarrow \text{ (dead code)}
\end{align*}
\]

The analysis generates the infinite sequence of stages, 
[0, 0], [0, 1], ..., [0, i], ... as i’s value in the loop’s body.

*The domain of intervals, where \([i, j] \subseteq [i', j']\) iff \(i \leq j\) and \(j \leq j'\), has infinitely ascending chains.*

To forcefully terminate the analysis, we can replace the \(\uplus\) operation by \(\nabla\), called a *widening operator*:

\[
\begin{align*}
\downarrow[i, j]^{\nabla} &= [i, j] & [i, j]^{\nabla}[i', j'] &= \begin{cases} 
-\infty & \text{if } i' < i \\
\text{else } i 
\end{cases}, \\
&\quad \begin{cases} 
+\infty & \text{if } j' > j \\
\text{else } j 
\end{cases}
\end{align*}
\]
The widening operator, which guarantees finite convergence for all increasing sequences on the interval domain, quickly terminates the example:

\[
\begin{align*}
\text{i} &= 0; \quad i = [0,0] \\
\text{while true} \{ \\
\quad \ldots \quad i = [0,0] \triangleright [1,1] = [0, +\infty] \\
\quad \text{i} &= \text{i} + 1; \\
\} \\
\end{align*}
\]

but in general, it can lose much precision:

\[
\begin{align*}
\text{int a} &= \text{new int}[10]; \\
\text{i} &= 0; \quad i = [0,0] \\
\text{while} \ (i < 10) \ { \} \\
\quad \ldots \ \text{a}[i] \quad i = [0,0] \triangleright [1,1] = [0, +\infty] \\
\quad \text{i} &= \text{i} + 1; \\
\} \\
\end{align*}
\]
For this reason, a complementary operation, \( \triangle \), called a *narrowing operation*, can be used after \( \triangledown \) gives convergence to recover some precision and retain a fixed-point solution.

We will not develop \( \triangle \) here, but for the interval domain, a suitable \( \triangle \) tries to reduce \(-\infty\) and \(+\infty\) to finite values. For the last example, the convergent value, \([0, +\infty]\), in the loop body would be narrowed to \([0, 10]\), making \(i\)’s value on loop exit \([10, 10]\).

Another approach is to use multiple “thresholds” for widening, e.g. \(-\infty, (2^{31} - 1), 0\), etc. for lower limits, and \((2^{31} - 1)\) and \(+\infty\) for upper limits.
Example: Program Interval Analysis

Consider the following program:

1 : \textbf{x} := 1; \quad 2 : \textbf{while} \ x \leq 1000 \ \textbf{do} \ [3 : \textbf{x} := \textbf{x} + 1]; \quad 4 :

It gives rise to the following flow equations:

\begin{align*}
En(1) &= [\infty, \infty] \\
Ex(1) &= [1, 1] \\
En(3) &= Ex[2] \cap [-\infty, 1000] \\
Ex(3) &= En(3) + [1, 1]
\end{align*}

\begin{align*}
En(2) &= Ex[1] \sqcup Ex(3) \\
Ex(2) &= En(2) \\
En(4) &= Ex(2) \cap [1001, \infty] \\
Ex(4) &= En(4)
\end{align*}

This leads to the following iterations:

\begin{align*}
Ex(1) & : [1, 1] \\
En(2) & : [1, 1] [1, 2] [1, 3] \ldots \\
Ex(2) & : [1, 1] [1, 2] \\
En(3) & : [1, 1] [1, 2] \\
Ex(3) & : [2, 2] [2, 3] \\
En(4) & : 
\end{align*}

Takes very long to terminate.
Widening for Interval Analysis

We define the **widening** operator over intervals as follows:

1. \( \bot \triangledown [c, d] = [c, d] \)
2. \([a, b] \triangledown [c, d] = \)
   
   - if \( a \leq c \) then \( a \) else if \( 0 \leq c \) then \( 0 \) else \(-\infty\)
   - if \( b \geq d \) then \( b \) else if \( d \leq 0 \) then \( 0 \) else \( \infty \)
Analyze with Widening

1 : $x := 1$;  
2 : while $x \leq 1000$ do [3 : $x := x + 1$];  
4 : 

With equations:

\[
\begin{align*}
En(1) &= [\infty, \infty] \\
Ex(1) &= [1, 1] \\
En(2) &= Ex[1] \lor (Ex(1) \land Ex(3)) \\
Ex(2) &= En(2) \\
En(3) &= Ex[2] \land [\infty, 1000] \\
Ex(3) &= En(3) + [1, 1] \\
En(4) &= Ex[2] \land [1001, \infty] \\
Ex(4) &= En(4)
\end{align*}
\]

This leads to the following iterations:

\[
\begin{align*}
Ex(1) :& \quad [1, 1] \\
En(2) :& \quad [1, 1] \lor [1, 2] = [1, \infty] \\
Ex(2) :& \quad [1, 1] \lor [1, \infty] \\
En(3) :& \quad [1, 1] \lor [1, 1000] \\
Ex(3) :& \quad [2, 2] \lor [2, 1001] \\
En(4) :& \quad [1001, \infty]
\end{align*}
\]

Terminates quickly but with very poor precision.
Requirements on Widening

- For all elements $a$ and $b$, $a \sqcup b \subseteq a \uplus b$
- For all ascending chains $a_0 \subseteq a_1 \subseteq a_2 \subseteq \cdots$, the following sequence is finite:
  \[
  \begin{align*}
  y_0 &= a_0 \\
  y_{i+1} &= y_i \uplus a_{i+1} \text{ for all } i \geq 0
  \end{align*}
  \]

Let $f : L \mapsto L$ be a monotonic function. Define the sequence

\[
\begin{align*}
  x_0 &= \bot \\
  x_{i+1} &= x_i \uplus f(x_i) \text{ for all } i \geq 0
  \end{align*}
\]

Claim 6. There exists $k$ such that $x_k = x_{k+1}$ and $x_k \in \text{Red}(f)$, where $\text{Red}(f) = \{a \in L \mid f(a) \subseteq a\}$. 
Narrowing

To correct the coarse overapproximation introduced by widening, we sometimes compensate by applying narrowing.

- For all elements \( b \subseteq a, b \subseteq a \bigtriangleup b \subseteq a \).
- For all descending chains \( a_0 \sqsupseteq a_1 \sqsupseteq a_2 \sqsupseteq \cdots \), the following sequence is finite:

\[
\begin{align*}
y_0 &= a_0 \\
y_{i+1} &= y_i \bigtriangleup a_{i+1} \quad \text{for all } i \geq 0
\end{align*}
\]

Let \( f : L \mapsto L \) be a monotonic function, and \( x \in \text{Red}(f) \), Define the sequence

\[
\begin{align*}
y_0 &= x \\
y_{i+1} &= y_i \nabla f(y_i) \quad \text{for all } i \geq 0
\end{align*}
\]

**Claim 7.** There exists \( k \) such that \( y_k = y_{k+1} \) and \( y_k \in \text{Red}(f) \), where \( \text{Red}(f) = \{ a \in L \mid f(a) \subseteq a \} \).
Narrowing for Interval Analysis

- \([a, b] \triangle \bot = [a, b]\)

- \([a, b] \triangle [c, d] =\)
  
  \[
  \begin{align*}
  \text{if } a = -\infty & \text{ then } c \text{ else } a, \\
  \text{if } b = \infty & \text{ then } d \text{ else } b
  \end{align*}
  \]
Widening and Then Narrowing

1 : \( x := 1; \)  2 : \textbf{while} \( x \leq 1000 \) \textbf{do} [3 : \( x := x + 1 \); 4 :

With equations:

\[
\begin{align*}
\text{En}(1) &= [-\infty, \infty] & \text{Ex}(1) &= [1, 1] & \text{En}(2) &= \text{Ex}[1] \Delta (\text{Ex}(1) \sqcup \text{Ex}(3)) \\
\text{Ex}(1) &= [1, 1] & \text{Ex}(2) &= \text{En}(2) \\
\text{En}(3) &= \text{Ex}[2] \cap [-\infty, 1000] & \text{En}(4) &= \text{Ex}(2) \cap [1001, \infty]
\end{align*}
\]

\[
\begin{align*}
\text{Ex}(3) &= \text{En}(3) + [1, 1] & \text{Ex}(4) &= \text{En}(4)
\end{align*}
\]

This leads to the following iterations:

\[
\begin{align*}
\text{Ex}(1) : [1, 1] \\
\text{En}(2) : [1, 1] & \quad [1, 1] \triangledown [1, 2] = [1, \infty] & [1, \infty] \Delta [1, 1001] = [1, 1001] \\
\text{Ex}(2) : [1, 1] & \quad [1, \infty] \\
\text{En}(3) : [1, 1] & \quad [1, 1000] \\
\text{Ex}(3) : [2, 2] & \quad [2, 1001] \\
\text{En}(4) : [1001, \infty] & \quad [1001, 1001]
\end{align*}
\]
Chains

- A subset Y ⊆ L in a poset (L, ⊆) is a chain if every two elements in Y are ordered
  - For all l₁, l₂ ∈ Y: l₁ ⊆ l₂ or l₂ ⊆ l₁
- An ascending chain is a sequence of values
  - l₁ ⊆ l₂ ⊆ l₃ ⊆ ...
- A strictly ascending chain is a sequence of values
  - l₁ ⊂ l₂ ⊂ l₃ ⊂ ...
- A descending chain is a sequence of values
  - l₁ ⊳ l₂ ⊳ l₃ ⊳ ...
- A strictly descending chain is a sequence of values
  - l₁ ⊳ l₂ ⊳ l₃ ⊳ ...
- L has a finite height if every chain in L is finite
- Lemma A poset (L, ⊆) has finite height if and only if every strictly decreasing and strictly increasing chains are finite
Monotone Functions

- A poset \((L, \sqsubseteq)\)

- A function \(f: L \rightarrow L\) is monotone if for every \(l_1, l_2 \in L:\)
  
  \[- l_1 \sqsubseteq l_2 \Rightarrow f(l_1) \sqsubseteq f(l_2)\]
Fixed Points

- A monotone function \( f: L \rightarrow L \) where 
  \((L, \sqsubseteq, \sqcup, \sqcap, \bot, \top)\) is a complete lattice
- \( \text{Fix}(f) = \{ l: l \in L, f(l) = l \} \)
- \( \text{Red}(f) = \{ l: l \in L, f(l) \sqsubseteq l \} \)
- \( \text{Ext}(f) = \{ l: l \in L, l \sqsubseteq f(l) \} \)
  - \( l_1 \sqsubseteq l_2 \Rightarrow f(l_1) \sqsubseteq f(l_2) \)
- Tarski’s Theorem 1955: if \( f \) is monotone then:
  - \( \text{lfp}(f) = \sqcap \text{Fix}(f) = \sqcap \text{Red}(f) \in \text{Fix}(f) \)
  - \( \text{gfp}(f) = \sqcup \text{Fix}(f) = \sqcup \text{Ext}(f) \in \text{Fix}(f) \)
Chaotic Iterations

◆ A lattice $L = (L, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ with finite strictly increasing chains
◆ $L^n = L \times L \times \ldots \times L$
◆ A monotone function $f: L^n \rightarrow L^n$
◆ Compute lfp($f$)
◆ The simultaneous least fixed of the system $\{x[i] = f_i(x) : 1 \leq i \leq n \}$

\[
\text{for } i := 1 \text{ to } n \text{ do} \\
\quad x[i] = \bot \\
\text{WL} = \{1, 2, \ldots, n\} \\
\text{x := (, , , , )} \quad \text{while (WL \neq \emptyset) do} \\
\text{while (f(x) \neq x) do} \\
\quad \text{select and remove an element } i \in \text{WL} \\
\quad \text{new := } f_i(x) \\
\quad \text{if (new \neq x[i]) then} \\
\quad \quad x[i] := \text{new; } \\
\quad \text{Add all the indexes that directly depends on } i \text{ to WL}
\]