Abstract Interpretation

Jan Midtgaard

Winter School, Day 1

http://janmidtgaard.dk/aiws15/

Saint Petersburg, Russia, 2015
What is this course about?

Crudely simplified the history of program analysis (or static analysis) can be split in two:

- an American school of program analysis
- a French school of program analysis

The former school has its roots in *data-flow analysis* and has given rise to many important results, e.g., within optimizing compilers.

Some of you may be familiar with data-flow analysis if you’ve taken a compiler course.

This course is concerned with the alternative, French approach.
What is abstract interpretation?

- It is a theory of *semantics-based program analysis*
- It was initially conceived in the late 1970’s by Patrick and Radhia Cousot
- It has been refined over the last 40 years
  - to new applications
  - to new kinds of semantics
  - to new programming paradigms
  - by new abstract domains
  - …
Which is the right approach?

None of them is right or wrong — it is simply an alternative view — an eye opener to a new world.

Why? To develop new techniques, to explain existing ones, to extend or strengthen them, to formalize them.

By Friday afternoon, you will be in a position to make an informed opinion.

It is not just an academic theory: it has been used to check/verify flight control software for both Airbus and Mars missions. By the end of this course, we will read papers about those.

It’ll get hairy: there will be mathematics and semantics
You take the red pill...
You take the red pill.

“...you stay in Wonderland and I show you how deep the rabbit-hole goes...”
On Friday afternoon, the participants should be able to:

- *describe* and *explain* basic analyses in terms of classical abstract interpretation.
- *apply* and *reason* about Galois connections.
- *implement* abstract interpreters on the basis of the derived program analyses.

In some sense: *thinking tools*

Like *$O$-analysis* is a tool for reasoning about execution time (and space), *abstract interpretation* is a tool for reasoning about analyses and properties.
Pedagogical choices / Contract

**Lectures**  – typically mornings, sometimes with a few exercises in class

**Reading**  – study research papers and slides (@home)

**Exercises**  – typically afternoons, both mathematics and programming. Over the week they build up a

**Project**  – a chance for you to apply your newly acquired skills – feel free to go crazy...
Your background

I’m assuming you all know about lexing, parsing, context-free grammars, abstract syntax trees (ASTs) and syntax-directed definitions/translations as taught in an undergraduate compiler course.

How many of you have taken

☐ a compiler course,
Your background

I’m assuming you all know about lexing, parsing, context-free grammars, abstract syntax trees (ASTs) and syntax-directed definitions/translations as taught in an undergraduate compiler course.

How many of you have taken

☐ a compiler course,

☐ a functional programming course,
I’m assuming you all know about lexing, parsing, context-free grammars, abstract syntax trees (ASTs) and syntax-directed definitions/translations as taught in an undergraduate compiler course.

How many of you have taken

□ a compiler course,
□ a functional programming course,
□ a formal semantics course?
Your background

I’m assuming you all know about lexing, parsing, context-free grammars, abstract syntax trees (ASTs) and syntax-directed definitions/translations as taught in an undergraduate compiler course.

How many of you have taken

☐ a compiler course,
☐ a functional programming course,
☐ a formal semantics course?

How many of you are

☐ students? BSc, MSc, PhD?
Your background

I’m assuming you all know about lexing, parsing, context-free grammars, abstract syntax trees (ASTs) and syntax-directed definitions/translations as taught in an undergraduate compiler course.

How many of you have taken

☐ a compiler course,
☐ a functional programming course,
☐ a formal semantics course?

How many of you are

☐ students? BSc, MSc, PhD?
☐ developers? @JetBrains?
My background

Researcher in programming languages (abstract interpretation, semantics, functional programming)

PhD in Computer Science, Aarhus University (2007). Since then:

- INRIA Rennes,
- Roskilde University,
- Aarhus University,
- Technical University of Denmark

I developed and ran this course in Aarhus 2010–2012

Fred Mesnard has since used it at Université de la Reunion in France
Outline

- What and how of the winter school
- Transition systems
- Math: Posets, CPOs, complete lattices, Galois connections, fixed points
- Abstract interpretation basics
- OCaml intro
Transition systems
Transition systems

**Definition.** A *transition system* is a triple (quadruple) \( \langle S, S_i, S_f, \rightarrow \rangle \) where

- \( S \) is a set of states
- \( S_i \subseteq S \) is a set of initial states
- \( S_f \subseteq S \) is an optional set of final states
- \( \forall s \in S_f, s' \in S : s \not\rightarrow s' \)
- \( \rightarrow \subseteq S \times S \) is a transition relation relating a state to its (possible) successors
Example 1: Euclid’s algorithm

Given two numbers \( x, y \in \mathbb{N} \) we can describe Euclid’s GCD algorithm as a transition system:

\[
S = \mathbb{N} \times \mathbb{N}
\]

\[
S_i = \{ \langle x, y \rangle \}
\]

\[
S_f = \{ \langle n, n \rangle \mid n \in \mathbb{N} \}
\]

\[
\rightarrow : \langle n, m \rangle \rightarrow \langle n - m, m \rangle \quad \text{if} \quad n > m
\]

\[
\langle n, m \rangle \rightarrow \langle n, m - n \rangle \quad \text{if} \quad n < m
\]

where we have written the transition relation using \textit{infix notation}.

We can write it out more formally as:

\[
\rightarrow = \{ (\langle n, m \rangle, \langle n - m, m \rangle) \mid n > m \}
\]

\[
\cup \{ (\langle n, m \rangle, \langle n, m - n \rangle) \mid n < m \}
\]
Example 1: Euclid’s algorithm

Given two numbers \( x, y \in \mathbb{N} \) we can describe Euclid’s GCD algorithm as a transition system:

\[
S = \mathbb{N} \times \mathbb{N}
\]

\[
S_i = \{\langle x, y \rangle\} \quad \text{←this is an "input-specific trans.sys."}
\]

\[
S_f = \{\langle n, n \rangle \mid n \in \mathbb{N}\}
\]

\[
\rightarrow : \langle n, m \rangle \rightarrow \langle n - m, m \rangle \quad \text{if } n > m
\]

\[
\langle n, m \rangle \rightarrow \langle n, m - n \rangle \quad \text{if } n < m
\]

where we have written the transition relation using infix notation.

We can write it out more formally as:

\[
\rightarrow = \{(\langle n, m \rangle, \langle n - m, m \rangle) \mid n > m\}
\]

\[
\cup \{(\langle n, m \rangle, \langle n, m - n \rangle) \mid n < m\}
\]
Example 1: Euclid’s algorithm

Given two numbers \( x, y \in \mathbb{N} \) we can describe Euclid’s GCD algorithm as a transition system:

\[
S = \mathbb{N} \times \mathbb{N}
\]

\[
S_i = S
\]

\[
S_f = \{ \langle n, n \rangle \mid n \in \mathbb{N} \}
\]

\[
\rightarrow : \langle n, m \rangle \rightarrow \langle n - m, m \rangle \quad \text{if} \quad n > m
\]

\[
\langle n, m \rangle \rightarrow \langle n, m - n \rangle \quad \text{if} \quad n < m
\]

where we have written the transition relation using \textit{infix notation}.

We can write it out more formally as:

\[
\rightarrow = \{ ((\langle n, m \rangle, \langle n - m, m \rangle)) \mid n > m \}
\]

\[
\cup \{ ((\langle n, m \rangle, \langle n, m - n \rangle)) \mid n < m \}
\]
Example 1: Euclid’s algorithm

Given two numbers \( x, y \in \mathbb{N} \) we can describe Euclid’s GCD algorithm as a transition system:

\[
S = \mathbb{N} \times \mathbb{N}
\]

\[
S_i = S \quad \text{← whereas this describes all possible inputs}
\]

\[
S_f = \{ \langle n, n \rangle \mid n \in \mathbb{N} \}
\]

\[
\rightarrow : \langle n, m \rangle \rightarrow \langle n - m, m \rangle \quad \text{if } n > m
\]

\[
\langle n, m \rangle \rightarrow \langle n, m - n \rangle \quad \text{if } n < m
\]

where we have written the transition relation using infix notation.

We can write it out more formally as:

\[
\rightarrow = \{ (\langle n, m \rangle, \langle n - m, m \rangle) \mid n > m \} \\
\quad \cup \{ (\langle n, m \rangle, \langle n, m - n \rangle) \mid n < m \}
\]
Example 2: Modeling a program

Modeling the program

\[
\begin{align*}
x & := 0; \\
\text{while } (x < 100) \{ \\
& \quad x := x + 1; \\
\}
\end{align*}
\]

as a transition system:

\[
S = \mathbb{Z} \\
S_i = \{0\} \\
\rightarrow = \{(x, x') \mid x < 100 \land x' = x + 1\}
\]

How to get from a program to a transition system is the topic of the next lecture.

For now we assume that we can model the semantics (the meaning) of a program as a transition system.
Mathematical foundations
Definition. A partially ordered set (poset) \( \langle S; \sqsubseteq \rangle \) is a set \( S \) equipped with a binary relation \( \sqsubseteq \subseteq S \times S \) with the following properties:

- **Reflexive**: \( \forall a \in S : a \sqsubseteq a \)
- **Antisymmetric**: \( \forall a, b \in S : a \sqsubseteq b \land b \sqsubseteq a \implies a = b \)
- **Transitive**: \( \forall a, b, c \in S : a \sqsubseteq b \land b \sqsubseteq c \implies a \sqsubseteq c \)

Example 1: \( \langle \mathbb{N}; \leq \rangle \) is a poset

Example 2: \( \langle \wp(S); \subseteq \rangle \) is a poset

Note: \( \wp(S) \) is sometimes written \( 2^S \)

Example 3: If \( \langle P; \sqsubseteq \rangle \) is a poset, then \( \langle P; \sqsubset \rangle \) is a poset
Upper and lower bounds

Let \( \langle P; \sqsubseteq \rangle \) be a partially ordered set.

**Definition.** \( u \in P \) is an upper bound of \( S \subseteq P \) iff
\[
\forall s \in S : s \sqsubseteq u
\]

**Definition.** \( l \in P \) is an lower bound of \( S \subseteq P \) iff
\[
\forall s \in S : l \sqsubseteq s
\]

**Definition.** \( u \in P \) is a least upper bound (lub) of \( S \subseteq P \) iff it is an upper bound of \( S \) and it is less than all other upper bounds:
\[
\forall u' \in P : (\forall s \in S : s \sqsubseteq u') \implies u \sqsubseteq u'
\]

**Definition.** \( l \in P \) is a greatest lower bound (glb) of \( S \subseteq P \) iff it is an lower bound of \( S \) and it is greater than all other lower bounds:
\[
\forall l' \in P : (\forall s \in S : l' \sqsubseteq s) \implies l' \sqsubseteq l
\]
Definition. A complete partial order is a poset such that all increasing chains \( c_i, i \in \mathbb{N} \) \( (\forall i \in \mathbb{N} : c_i \sqsubseteq c_{i+1}) \) have a least upper bound:

\[
\bigsqcup_{i \in \mathbb{N}} c_i
\]

Non-example: \( \langle \mathbb{N}; \leq \rangle \) is not a CPO. Why?

Example: \( \langle \mathcal{P}(S); \subseteq \rangle \) is a CPO.
Complete lattices

**Definition.** A complete lattice is a poset $\langle C; \sqsubseteq, \bot, \top, \sqcup, \sqcap \rangle$ such that

- the least upper bound $\sqcup S$ ('join') and
- the greatest lower bound $\sqcap S$ ('meet') exists for every subset $S$ of $C$.
- $\bot = \sqcap C$ ('bottom') denotes the infimum of $C$ and
- $\top = \sqcup C$ ('top') denotes the supremum of $C$.

**Example 1:** $\langle \wp(S); \subseteq, \emptyset, S, \cup, \cap \rangle$ is a complete lattice.

**Example 2:** The integers (extended with $-\infty$ and $+\infty$) is a complete lattice $\langle \mathbb{Z} \cup \{-\infty, +\infty\}; \leq, -\infty, +\infty, \max, \min \rangle$. 
Theorem. The set of total functions $D \rightarrow C$, whose codomain is a complete lattice $\langle C; \sqsubseteq, \bot, \top, \sqcup, \sqcap \rangle$, is itself a complete lattice $\langle D \rightarrow C; \sqsubseteq, \bot, \top, \sqcup, \sqcap \rangle$ under the pointwise ordering $f \sqsubseteq f' \iff \forall x. f(x) \sqsubseteq f'(x)$, and with

- $\bot = \lambda x. \bot$
- $\top = \lambda x. \top$
- $f \sqcup g = \lambda x. f(x) \sqcup g(x)$
- $f \sqcap g = \lambda x. f(x) \sqcap g(x)$

Here $\lambda x. \ldots$ is a mathematical function with argument $x$. 
A quick comparison

Complete Lattice

Complete Partial Order

Partially Ordered Set
Galois connections
Definition. A Galois connection is a pair of functions $\alpha$, $\gamma$ between two partially ordered sets:

\[
\langle C; \sqsubseteq \rangle \quad \overset{\gamma}{\longrightarrow} \quad \langle A; \leq \rangle
\]

\[
\langle A; \leq \rangle \quad \overset{\alpha}{\longrightarrow} \quad \langle C; \sqsubseteq \rangle
\]

\textit{Concretization} and \textit{abstraction}.
Definition. A Galois connection is a pair of functions $\alpha$, $\gamma$ between two partially ordered sets:

\[
\forall a \in A, \ c \in C : \alpha(c) \leq a \iff c \sqsubseteq \gamma(a)
\]
You already know the pattern of moving from one side of an inequation to another from high school:

\[ \forall x, y, z \in \mathbb{Z} : x + z \leq y \iff x \leq y - z \]

which we can write with \( \alpha \) and \( \gamma \) as:

\[ \forall x, y, z \in \mathbb{Z} : \alpha(x) \leq y \iff x \leq \gamma(y) \]

where \( \alpha(n) = n + z \)

\( \gamma(n) = n - z \)
Definition. A Galois connection is a pair of functions $\alpha$ and $\gamma$ satisfying

(a) $\alpha$ and $\gamma$ are monotone (read: order-preserving)
   (for all $c, c' \in C : c \sqsubseteq c' \implies \alpha(c) \leq \alpha(c')$ and
   for all $a, a' \in A : a \leq a' \implies \gamma(a) \sqsubseteq \gamma(a')$),

(b) $\alpha \circ \gamma$ is reductive (for all $a \in A : \alpha \circ \gamma(a) \leq a$),

(c) $\gamma \circ \alpha$ is extensive (for all $c \in C : c \sqsubseteq \gamma \circ \alpha(c)$).

Galois connections are typeset as $\langle C; \sqsubseteq \rangle \xleftrightarrow{\gamma}{\alpha} \langle A; \leq \rangle$. 
Theorem. For a Galois connection between two complete lattices $\langle C; \sqsubseteq, \bot_c, \top_c, \sqcup, \sqcap \rangle$ and $\langle A; \leq, \bot_a, \top_a, \lor, \land \rangle$, $\alpha$ is a complete join-morphism (CJM):

$$\text{for all } S_c \subseteq C : \alpha(\sqcup S_c) = \lor \alpha(S_c) = \lor \{ \alpha(c) \mid c \in S_c \}$$

and $\gamma$ is a complete meet morphism (CMM):

$$\text{for all } S_a \subseteq A : \gamma(\land S_a) = \land \gamma(S_a) = \land \{ \gamma(a) \mid a \in S_a \}$$

Again: we can view these as algebraic rewriting rules.
Theorem. **The composition of two Galois connections**

\[ \langle C; \sqsubseteq \rangle \leftrightarrow_{\alpha_1} \langle B; \subseteq \rangle \quad \text{and} \quad \langle B; \subseteq \rangle \leftrightarrow_{\alpha_2} \langle A; \leq \rangle \quad \text{is itself a} \]

**Galois connection:**

\[ \langle C; \sqsubseteq \rangle \leftrightarrow_{\alpha_2 \circ \alpha_1} \langle A; \leq \rangle \]

We can typeset this theorem as an inference rule:

\[
\langle C; \sqsubseteq \rangle \leftrightarrow_{\alpha_1} \langle B; \subseteq \rangle \quad \langle B; \subseteq \rangle \leftrightarrow_{\alpha_2} \langle A; \leq \rangle \\
\hline
\langle C; \sqsubseteq \rangle \leftrightarrow_{\alpha_2 \circ \alpha_1} \langle A; \leq \rangle
\]

Hence Galois connections stack up like Lego bricks!
Galois connections in which $\alpha$ is surjective / onto (or equivalently $\gamma$ is injective) are typeset as:

\[ \langle C; \sqsubseteq \rangle \leftrightarrow_{\alpha} \gamma \leftrightarrow \langle A; \leq \rangle \]

and sometimes called Galois surjections (or insertions)

Galois connections in which $\alpha$ is injective / one-to-one (or equivalently $\gamma$ is surjective) are typeset as:

\[ \langle C; \sqsubseteq \rangle \leftrightarrow_{\alpha} \gamma \leftrightarrow \langle A; \leq \rangle \]

and sometimes called Galois injections

When both $\alpha$ and $\gamma$ are surjective, the two domains are isomorphic, typeset as

\[ \langle C; \sqsubseteq \rangle \leftrightarrow_{\alpha} \gamma \leftrightarrow \langle A; \leq \rangle \]
Example: The Parity abstract domain

Galois connections capture *property extraction* which is essential for static analysis. Consider an abstraction into a Parity domain:

\[
\langle \varnothing(\mathbb{N}_0); \subseteq \rangle \xrightarrow{\gamma} \langle \text{Par}; \sqsubseteq \rangle \quad \text{Par : } \begin{array}{c}
\bot \\
oindent\downarrow \\
oindent\rightarrow \\
oindent\rightarrow \\
oindent\rightarrow \\
\text{odd} \\
\text{even} \\
\top
\end{array}
\]

(The above Hasse diagram defines the Parity ordering \( \bot \sqsubseteq \text{odd} \sqsubseteq \top \) and \( \bot \sqsubseteq \text{even} \sqsubseteq \top \))

The abstraction and concretization functions are:

\[
\gamma(\bot) = \emptyset \\
\gamma(\text{odd}) = \{ n \in \mathbb{N}_0 \mid n \mod 2 = 1 \} \\
\gamma(\text{even}) = \{ n \in \mathbb{N}_0 \mid n \mod 2 = 0 \} \\
\gamma(\top) = \mathbb{N}_0
\]

\[
\alpha(N) = \begin{cases} 
\bot & \text{if } N = \emptyset \\
\text{odd} & \text{if } \forall n \in N : n \mod 2 = 1 \\
\text{even} & \text{if } \forall n \in N : n \mod 2 = 0 \\
\top & \text{otherwise}
\end{cases}
\]
Example: an isomorphism

Since Galois connections is a generalization of isomorphisms, they also fit nicely into the theory.

For example, we can represent a set of pairs as a function that maps a first component to its second components:

\[
\langle \varnothing(A \times B); \subseteq \rangle \xleftarrow{\alpha} \langle A \to \varnothing(B); \subseteq \rangle
\]

where

\[
\alpha(R) = \lambda a. \{ b \mid (a, b) \in R \}
\]

\[
\gamma(F) = \{(a, b) \mid b \in F(a)\}
\]
Fixed points
Fixed points, briefly

Definition. A fixed point of a function \( f \), is a point \( x \) such that \( f(x) = x \)

Assume \( f : P \to P \) operates over a poset \( \langle P; \sqsubseteq \rangle \)

Definition. A pre-fixed point is a point \( x \) such that \( x \sqsubseteq f(x) \)

Definition. A post-fixed point is a point \( x \) such that \( f(x) \sqsubseteq x \)

Definition. A least fixed point (lfp) is a fixed point \( l \) such that for all other fixed points \( l' : (f(l') = l') \implies l \sqsubseteq l' \)

Definition. A greatest fixed point (gfp) is a fixed point \( l \) such that for all other fixed points \( l' : (f(l') = l') \implies l' \sqsubseteq l \)
Theorem. If $L$ is a complete lattice and $f : L \rightarrow L$ is a monotone function, $f$’s fixed points themselves form a complete lattice.

Hence Tarski tells us that there exists a least fixed point (and a greatest fixed point).
Abstract interpretation basics
Canonical abstract interpretation approximates the collecting semantics of a transition system.

A standard example of a collecting semantics is the reachable states from a given set of initial states \( S_i \). Given a transition function \( F \) defined as:

\[
F(\Sigma) = S_i \cup \{ \sigma \mid \exists \sigma' \in \Sigma : \sigma' \rightarrow \sigma \}
\]

we can express the reachable states of \( F \) as the least fixed point \( \text{lfp} F \) of \( F \).

For a fixed point \( F(\Sigma) = \Sigma \) of \( F \):

\[
S_i \subseteq \Sigma \quad \land \quad \forall \sigma' \in \Sigma : \sigma' \rightarrow \sigma \implies \sigma \in \Sigma
\]

which expresses the transitive closure of the states reachable from \( S_i \).
Abstract interpretation basics

Canonical abstract interpretation approximates the collecting semantics of a transition system.

A standard example of a collecting semantics is the reachable states from a given set of initial states $S_i$. Given a transition function $F$ defined as:

$$F(\Sigma) = S_i \cup \{\sigma | \exists \sigma' \in \Sigma : \sigma' \rightarrow \sigma\}$$

we can express the reachable states of $F$ as the least fixed point $\text{lfp } F$ of $F$.

We can obtain $\text{lfp } F$ by Kleene iteration$^1$:

$$\emptyset, F(\emptyset), F^2(\emptyset), F^3(\emptyset), \ldots$$

---

$^1$In general we can only obtain $\text{lfp } f$ this way if $f$ is continuous $f(\bigsqcup S) = \bigsqcup f(S)$
Example: Collecting semantics

<table>
<thead>
<tr>
<th>Program statement</th>
<th>State (on entry)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x := 1; )</td>
<td>{ }</td>
</tr>
<tr>
<td>( \text{while} \ (x &lt; 100) \ { } )</td>
<td>{ }</td>
</tr>
<tr>
<td>( x := x + 1; )</td>
<td>{ }</td>
</tr>
<tr>
<td>}</td>
<td>{ }</td>
</tr>
</tbody>
</table>
Example: Collecting semantics

Program statement State (on entry)

x := 1; {0}

while (x < 100) {

x := x + 1; {}

}

{}
<table>
<thead>
<tr>
<th>Program statement</th>
<th>State (on entry)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x := 1;</code></td>
<td><code>{0}</code></td>
</tr>
<tr>
<td><code>while (x &lt; 100) {</code></td>
<td><code>{1}</code></td>
</tr>
<tr>
<td><code>  x := x + 1;</code></td>
<td><code>{}</code></td>
</tr>
<tr>
<td><code>}</code></td>
<td><code>{}</code></td>
</tr>
<tr>
<td>Program statement</td>
<td>State (on entry)</td>
</tr>
<tr>
<td>-------------------</td>
<td>-----------------</td>
</tr>
<tr>
<td><code>x := 1;</code></td>
<td><code>{0}</code></td>
</tr>
<tr>
<td><code>while (x &lt; 100) {</code></td>
<td><code>{1}</code></td>
</tr>
<tr>
<td><code>  x := x + 1;</code></td>
<td><code>{1}</code></td>
</tr>
<tr>
<td><code>}</code></td>
<td><code>{}</code></td>
</tr>
</tbody>
</table>
### Example: Collecting semantics

<table>
<thead>
<tr>
<th>Program statement</th>
<th>State (on entry)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x := 1; )</td>
<td>{0}</td>
</tr>
<tr>
<td>( \text{while } (x &lt; 100) { )</td>
<td>{1, 2}</td>
</tr>
<tr>
<td>( \qquad x := x + 1; )</td>
<td>{1}</td>
</tr>
<tr>
<td>}</td>
<td>{}</td>
</tr>
</tbody>
</table>
Example: Collecting semantics

<table>
<thead>
<tr>
<th>Program statement</th>
<th>State (on entry)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x := 1;</code></td>
<td><code>{0}</code></td>
</tr>
<tr>
<td><code>while (x &lt; 100) {</code></td>
<td><code>{1, 2}</code></td>
</tr>
<tr>
<td><code>x := x + 1;</code></td>
<td><code>{1, 2}</code></td>
</tr>
<tr>
<td><code>}</code></td>
<td><code>{}</code></td>
</tr>
</tbody>
</table>
Example: Collecting semantics

<table>
<thead>
<tr>
<th>Program statement</th>
<th>State (on entry)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x := 1;</code></td>
<td><code>{0}</code></td>
</tr>
<tr>
<td><code>while (x &lt; 100) {</code></td>
<td><code>{1, 2, 3}</code></td>
</tr>
<tr>
<td><code>  x := x + 1;</code></td>
<td><code>{1, 2}</code></td>
</tr>
<tr>
<td><code>}</code></td>
<td><code>{}</code></td>
</tr>
</tbody>
</table>
### Example: Collecting semantics

<table>
<thead>
<tr>
<th>Program statement</th>
<th>State (on entry)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x := 1; )</td>
<td>{0}</td>
</tr>
<tr>
<td>while ((x &lt; 100)) {</td>
<td>{1, 2, 3}</td>
</tr>
<tr>
<td>( x := x + 1; )</td>
<td>{1, 2, 3}</td>
</tr>
<tr>
<td>}</td>
<td>{}</td>
</tr>
</tbody>
</table>
Example: Collecting semantics

Program statement

\[ x := 1; \]

\[ \text{while } (x < 100) \{ \]

\[ x := x + 1; \]

\[ \} \]

State (on entry)

\[ \{0\} \]

\[ \{1, 2, 3, \ldots, 98, 99\} \]

\[ \{1, 2, 3, \ldots, 98\} \]

\[ \{\} \]

Jumping forward in time...
Example: Collecting semantics

Program statement

\[ x := 1; \]
\[
\text{while } (x < 100) \{ \\
\quad x := x + 1; \\
\}
\]

State (on entry)

\{0\}

\{1, 2, 3, \ldots, 98, 99\}

\{1, 2, 3, \ldots, 98, 99\}

\{\}\n
Jumping forward in time...
Example: Collecting semantics

<table>
<thead>
<tr>
<th>Program statement</th>
<th>State (on entry)</th>
</tr>
</thead>
<tbody>
<tr>
<td>x := 1;</td>
<td>{0}</td>
</tr>
<tr>
<td>while (x &lt; 100) {</td>
<td>{1, 2, 3, \ldots, 98, 99, 100}</td>
</tr>
<tr>
<td>x := x + 1;</td>
<td>{1, 2, 3, \ldots, 98, 99}</td>
</tr>
<tr>
<td>}</td>
<td>{}</td>
</tr>
</tbody>
</table>
Example: Collecting semantics

Program statement

\[ x := 1; \]

\[ \text{while } (x < 100) \{ \]
\[ \quad x := x + 1; \]
\[ \} \]

State (on entry)

\[ \{0\} \]

\[ \{1, 2, 3, \ldots, 98, 99, 100\} \]

\[ \{1, 2, 3, \ldots, 98, 99\} \]

\[ \{100\} \]
Example: Collecting semantics

Program statement

\[ x := 1; \]
\[ \text{while} \ (x < 100) \ { \]
\[ x := x + 1; \]
\[ } \]

State (on entry)

\[ \{0\} \]
\[ \{1, 2, 3, \ldots, 98, 99, 100\} \]
\[ \{1, 2, 3, \ldots, 98, 99\} \]
\[ \{100\} \]

Fixed point
The strength of the collecting semantics

- The collecting semantics is ideal, i.e., it is the *most precise analysis*.

- Unfortunately it is in general uncomputable: an implementation is not guaranteed to terminate.

- We therefore approximate the collecting semantics, by computing a fixed point over an alternative and perhaps simpler domain: an *abstract* interpretation.
Abstractions are represented as Galois connections which connect complete lattices through $\alpha$ and $\gamma$.

We can derive an analysis systematically by composing the transition function with these functions: $\alpha \circ F \circ \gamma$ and gradually refine the collecting semantics into a computable analysis function by mere calculation.

Hence instead of inventing a static analysis, we arrive at one by a *structured abstraction* of the set of states $\wp(S)$.
By the \textit{fixed point transfer theorem} we can compute a sound approximation of the collecting semantics:

\begin{proposition}
Let $\langle C; \sqsubseteq \rangle \xleftrightarrow[\alpha]{} \langle A; \leq \rangle$ be a Galois connection between complete lattices. If $F$ and $F^\#$ are monotone and $\alpha \circ F \circ \gamma \leq F^\#$ then $\alpha(lfp F) \leq lfp F^\#$
\end{proposition}
<table>
<thead>
<tr>
<th>Program statement</th>
<th>Approx. state (on entry)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x := 1; )</td>
<td>( \perp )</td>
</tr>
<tr>
<td>( \textbf{while} \ (x &lt; 100) { )</td>
<td>( \perp )</td>
</tr>
<tr>
<td>\hspace{1em} ( x := x + 1; )</td>
<td>( \perp )</td>
</tr>
<tr>
<td>)</td>
<td>( \perp )</td>
</tr>
</tbody>
</table>
Example: Parity analysis

<table>
<thead>
<tr>
<th>Program statement</th>
<th>Approx. state (on entry)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x := 1;$</td>
<td>even</td>
</tr>
<tr>
<td>while ($x &lt; 100$)</td>
<td>$\bot$</td>
</tr>
<tr>
<td>$x := x + 1;$</td>
<td>$\bot$</td>
</tr>
<tr>
<td>}</td>
<td>$\bot$</td>
</tr>
</tbody>
</table>
Example: Parity analysis

<table>
<thead>
<tr>
<th>Program statement</th>
<th>Approx. state (on entry)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x := 1;</code></td>
<td><code>even</code></td>
</tr>
<tr>
<td><code>while (x &lt; 100) {</code></td>
<td><code>odd</code></td>
</tr>
<tr>
<td><code>  x := x + 1;</code></td>
<td><code>⊥</code></td>
</tr>
<tr>
<td><code>}</code></td>
<td><code>⊥</code></td>
</tr>
</tbody>
</table>
Example: Parity analysis

<table>
<thead>
<tr>
<th>Program statement</th>
<th>Approx. state (on entry)</th>
</tr>
</thead>
<tbody>
<tr>
<td>x := 1;</td>
<td>even</td>
</tr>
<tr>
<td>while (x &lt; 100) {</td>
<td></td>
</tr>
<tr>
<td>x := x + 1;</td>
<td>odd</td>
</tr>
<tr>
<td>}</td>
<td>odd</td>
</tr>
</tbody>
</table>
## Example: Parity analysis

<table>
<thead>
<tr>
<th>Program statement</th>
<th>Approx. state (on entry)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x := 1; )</td>
<td>even</td>
</tr>
<tr>
<td>while ((x &lt; 100)) {</td>
<td>T</td>
</tr>
<tr>
<td>( x := x + 1; )</td>
<td>odd</td>
</tr>
<tr>
<td>}</td>
<td>odd</td>
</tr>
</tbody>
</table>
Example: Parity analysis

<table>
<thead>
<tr>
<th>Program statement</th>
<th>Approx. state (on entry)</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{x := 1;}</td>
<td>\textit{even}</td>
</tr>
<tr>
<td>\texttt{while (x &lt; 100) { } x := x + 1;}</td>
<td>\texttt{T}</td>
</tr>
</tbody>
</table>
Example: Parity analysis

<table>
<thead>
<tr>
<th>Program statement</th>
<th>Approx. state (on entry)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x := 1; )</td>
<td>even</td>
</tr>
<tr>
<td>( \text{while} \ (x &lt; 100) \ { } )</td>
<td>( \top )</td>
</tr>
<tr>
<td>( x := x + 1; )</td>
<td>( \top )</td>
</tr>
<tr>
<td></td>
<td>( \top )</td>
</tr>
</tbody>
</table>

Fixed point
Example: Parity analysis

<table>
<thead>
<tr>
<th>Program statement</th>
<th>Approx. state (on entry)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x := 1; )</td>
<td>\textit{even}</td>
</tr>
<tr>
<td>while (( x &lt; 100 )) {</td>
<td></td>
</tr>
<tr>
<td>\hspace{1cm} ( x := x + 1; )</td>
<td>( \top )</td>
</tr>
<tr>
<td>}</td>
<td>( \top )</td>
</tr>
</tbody>
</table>

Fixed point

The result is sound: it accounts for all possible concrete executions \( \textit{albeit not very precisely...} \)
Example: Parity analysis

Program statement

\[
x := 1;
\]
\[
\text{while (} x < 100 \text{) }
\]
\[
\quad x := x + 1;
\]
\[
\}\]

Approx. state (on entry)

\[
\text{even } \sqsupseteq \alpha(\{0\})
\]
\[
\top \sqsupseteq \alpha(\{1,\ldots, 99, 100\})
\]
\[
\top \sqsupseteq \alpha(\{1,\ldots, 99\})
\]
\[
\top \sqsupseteq \alpha(\{100\})
\]

Fixed point

The result is sound: it accounts for all possible concrete executions (albeit not very precisely . . . )
Example: Parity analysis

Program statement

```c
x := 1;
while (x < 100) {
    x := x + 1;
}
```

Approx. state (on entry)

```
\( \gamma(even) \supseteq \{0\} \)
\( \gamma(\top) \supseteq \{1, \ldots, 99, 100\} \)
\( \gamma(\top) \supseteq \{1, \ldots, 99\} \)
\( \gamma(\top) \supseteq \{100\} \)
```

Fixed point

The result is sound: it accounts for all possible concrete executions (albeit not very precisely...)
Variations
An alternative approach

Rather than simplifying the abstract domains into finite ones, *widening* and *narrowing* permits infinite ones.

A first widening iteration overshoots the least fixed point but still ensures termination.

A second narrowing iteration improves the results of the widening iteration.
Example: Interval analysis without widening

<table>
<thead>
<tr>
<th>Program statement</th>
<th>Approx. state (on entry)</th>
</tr>
</thead>
<tbody>
<tr>
<td>x := 1;</td>
<td>⊥</td>
</tr>
<tr>
<td>while (x &lt; 100) {</td>
<td>⊥</td>
</tr>
<tr>
<td>x := x + 1;</td>
<td>⊥</td>
</tr>
<tr>
<td>}</td>
<td>⊥</td>
</tr>
</tbody>
</table>
Example: Interval analysis without widening

<table>
<thead>
<tr>
<th>Program statement</th>
<th>Approx. state (on entry)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x := 1; )</td>
<td>([0; 0])</td>
</tr>
<tr>
<td>while ((x &lt; 100)) {</td>
<td>(\bot)</td>
</tr>
<tr>
<td>\hspace{1em} ( x := x + 1; ) \hspace{1em}</td>
<td>(\bot)</td>
</tr>
<tr>
<td>}</td>
<td>(\bot)</td>
</tr>
</tbody>
</table>
Example: Interval analysis without widening

Program statement

\[ x := 1; \]
\[ \text{while } (x < 100) \{ \]
\[ x := x + 1; \]
\[ \}

Approx. state (on entry)

\[ [0; 0] \]
\[ [1; 1] \]
\[ ⊥ \]
\[ ⊥ \]
<table>
<thead>
<tr>
<th>Program statement</th>
<th>Approx. state (on entry)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x := 1;</code></td>
<td><code>[0; 0]</code></td>
</tr>
<tr>
<td><code>while (x &lt; 100) {</code></td>
<td><code>[1; 1]</code></td>
</tr>
<tr>
<td><code>  x := x + 1;</code></td>
<td><code>[1; 1]</code></td>
</tr>
<tr>
<td><code>}</code></td>
<td><code>⊥</code></td>
</tr>
</tbody>
</table>
**Example: Interval analysis without widening**

<table>
<thead>
<tr>
<th>Program statement</th>
<th>Approx. state (on entry)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x := 1; )</td>
<td>([0; 0])</td>
</tr>
<tr>
<td>while ((x &lt; 100)) {</td>
<td>([1; 2])</td>
</tr>
<tr>
<td>( x := x + 1; )</td>
<td>([1; 1])</td>
</tr>
<tr>
<td>}</td>
<td>(\perp)</td>
</tr>
</tbody>
</table>
### Example: Interval analysis without widening

<table>
<thead>
<tr>
<th>Program statement</th>
<th>Approx. state (on entry)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x := 1;</code></td>
<td><code>[0; 0]</code></td>
</tr>
<tr>
<td><code>while (x &lt; 100) {</code></td>
<td><code>[1; 2]</code></td>
</tr>
<tr>
<td><code>  x := x + 1;</code></td>
<td><code>[1; 2]</code></td>
</tr>
<tr>
<td><code>}</code></td>
<td><code>⊥</code></td>
</tr>
<tr>
<td>Program statement</td>
<td>Approx. state (on entry)</td>
</tr>
<tr>
<td>-------------------</td>
<td>-------------------------</td>
</tr>
<tr>
<td>$x := 1;$</td>
<td>[0; 0]</td>
</tr>
<tr>
<td>while (x &lt; 100) {</td>
<td>[1; 3]</td>
</tr>
<tr>
<td>$x := x + 1;$</td>
<td>[1; 2]</td>
</tr>
<tr>
<td>}</td>
<td>$\bot$</td>
</tr>
</tbody>
</table>
Example: Interval analysis without widening

<table>
<thead>
<tr>
<th>Program statement</th>
<th>Approx. state (on entry)</th>
</tr>
</thead>
<tbody>
<tr>
<td>x := 1;</td>
<td>[0; 0]</td>
</tr>
<tr>
<td>while (x &lt; 100) {</td>
<td>[1; 3]</td>
</tr>
<tr>
<td>x := x + 1;</td>
<td>[1; 3]</td>
</tr>
<tr>
<td>}</td>
<td>⊥</td>
</tr>
</tbody>
</table>
Example: Interval analysis without widening

Program statement                      | Approx. state (on entry)
----------------------------------------|------------------------
    x := 1;                             | [0; 0]

while (x < 100) {
    x := x + 1;
}                                         | [1; 99]

⊥                                            | [1; 98]

Jumping forward in time...
### Example: Interval analysis without widening

<table>
<thead>
<tr>
<th>Program statement</th>
<th>Approx. state (on entry)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x := 1; )</td>
<td>([0; 0])</td>
</tr>
<tr>
<td>while (( x &lt; 100 )) {</td>
<td>([1; 99])</td>
</tr>
<tr>
<td>( x := x + 1; )</td>
<td>([1; 99])</td>
</tr>
<tr>
<td>}</td>
<td>(\bot)</td>
</tr>
</tbody>
</table>

Jumping forward in time...
Example: Interval analysis without widening

<table>
<thead>
<tr>
<th>Program statement</th>
<th>Approx. state (on entry)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x := 1; )</td>
<td>([0; 0])</td>
</tr>
<tr>
<td>( \text{while } (x &lt; 100) { )</td>
<td>([1; 100])</td>
</tr>
<tr>
<td>( x := x + 1; )</td>
<td>([1; 99])</td>
</tr>
<tr>
<td>}</td>
<td>(\bot)</td>
</tr>
</tbody>
</table>
### Example: Interval analysis without widening

<table>
<thead>
<tr>
<th>Program statement</th>
<th>Approx. state (on entry)</th>
</tr>
</thead>
<tbody>
<tr>
<td>x := 1;</td>
<td>[0; 0]</td>
</tr>
<tr>
<td>while (x &lt; 100) {</td>
<td></td>
</tr>
<tr>
<td>x := x + 1;</td>
<td>[1; 99]</td>
</tr>
<tr>
<td>}</td>
<td>[100; 100]</td>
</tr>
</tbody>
</table>
Example: Interval analysis without widening

<table>
<thead>
<tr>
<th>Program statement</th>
<th>Approx. state (on entry)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x := 1; )</td>
<td>([0; 0])</td>
</tr>
<tr>
<td>( \text{while} (x &lt; 100) ) { }</td>
<td>([1; 100])</td>
</tr>
<tr>
<td>( x := x + 1; )</td>
<td>([1; 99])</td>
</tr>
<tr>
<td></td>
<td>([100; 100])</td>
</tr>
</tbody>
</table>

**Fixed point**
Example: Interval analysis without widening

<table>
<thead>
<tr>
<th>Program statement</th>
<th>Approx. state (on entry)</th>
</tr>
</thead>
<tbody>
<tr>
<td>x := 1;</td>
<td>[0; 0]</td>
</tr>
<tr>
<td>while (x &lt; 100) {</td>
<td>[1; 100]</td>
</tr>
<tr>
<td>x := x + 1;</td>
<td>[1; 99]</td>
</tr>
<tr>
<td>}</td>
<td>[100; 100]</td>
</tr>
</tbody>
</table>

Fixed point

In general, we’re not guaranteed to reach a fixed point in a finite number of steps (read: impl. may not halt)
We compute instead the limit of the sequence:

\[ X_0 = \perp \]
\[ X_{i+1} = X_i \nabla F^\#(X_i) \]

where \( \nabla \) denotes the *widening operator*: an operator with the following properties:

- For all \( x, y : x \sqsubseteq (x \nabla y) \land y \sqsubseteq (x \nabla y) \)
- For any increasing chain \( Y_0 \sqsubseteq Y_1 \sqsubseteq Y_2 \sqsubseteq \ldots \) the alternative chain defined as \( Y_0' = Y_0 \) and \( Y_{i+1}' = Y_i' \nabla Y_{i+1} \) stabilizes after a finite amount of steps.
**Example: Interval analysis with widening**

<table>
<thead>
<tr>
<th>Program statement</th>
<th>Approx. state (on entry)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x := 1; )</td>
<td>( \perp )</td>
</tr>
<tr>
<td>while (( x &lt; 100 )) { ( x := x + 1; ) }</td>
<td>( \perp )</td>
</tr>
</tbody>
</table>
Example: Interval analysis with widening

<table>
<thead>
<tr>
<th>Program statement</th>
<th>Approx. state (on entry)</th>
</tr>
</thead>
<tbody>
<tr>
<td>x := 1;</td>
<td>[0; 0]</td>
</tr>
<tr>
<td>while (x &lt; 100) {</td>
<td>⊥</td>
</tr>
<tr>
<td>x := x + 1;</td>
<td>⊥</td>
</tr>
<tr>
<td>}</td>
<td>⊥</td>
</tr>
<tr>
<td>Program statement</td>
<td>Approx. state (on entry)</td>
</tr>
<tr>
<td>-------------------</td>
<td>-------------------------</td>
</tr>
<tr>
<td>$x := 1;$</td>
<td>$[0; 0]$</td>
</tr>
<tr>
<td>while ($x &lt; 100$) {</td>
<td></td>
</tr>
<tr>
<td>$x := x + 1;$</td>
<td>$[1; 1]$</td>
</tr>
<tr>
<td>}</td>
<td>$\bot$</td>
</tr>
<tr>
<td></td>
<td>$\bot$</td>
</tr>
<tr>
<td>Program statement</td>
<td>Approx. state (on entry)</td>
</tr>
<tr>
<td>-------------------</td>
<td>------------------------</td>
</tr>
<tr>
<td>( x := 1; )</td>
<td>([0; 0])</td>
</tr>
<tr>
<td>( \text{while } (x &lt; 100) { )</td>
<td>([1; 1])</td>
</tr>
<tr>
<td>\hspace{0.5cm} ( x := x + 1; )</td>
<td>([1; 1])</td>
</tr>
<tr>
<td>}</td>
<td>( \perp )</td>
</tr>
</tbody>
</table>
Example: Interval analysis with widening

<table>
<thead>
<tr>
<th>Program statement</th>
<th>Approx. state (on entry)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x := 1; )</td>
<td>([0; 0])</td>
</tr>
<tr>
<td>while ((x &lt; 100)) { ( x := x + 1; )</td>
<td>([1; 1] \nabla [1; 2] = [1; +\infty])</td>
</tr>
<tr>
<td>}</td>
<td>([1; 1])</td>
</tr>
<tr>
<td></td>
<td>(\bot)</td>
</tr>
</tbody>
</table>
### Example: Interval analysis with widening

<table>
<thead>
<tr>
<th>Program statement</th>
<th>Approx. state (on entry)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x := 1; )</td>
<td>([0; 0])</td>
</tr>
<tr>
<td>( \text{while } (x &lt; 100) ) {</td>
<td>( [1; 1] \triangledown [1; 2] = [1; +\infty] )</td>
</tr>
<tr>
<td>( x := x + 1; )</td>
<td>([1; 99])</td>
</tr>
<tr>
<td>}</td>
<td>([100; +\infty])</td>
</tr>
</tbody>
</table>
## Example: Interval analysis with widening

<table>
<thead>
<tr>
<th>Program statement</th>
<th>Approx. state (on entry)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x := 1; )</td>
<td>([0; 0])</td>
</tr>
<tr>
<td>( \text{while } (x &lt; 100) { )</td>
<td>([1; +\infty] \uplus [1; 100] = [1; +\infty])</td>
</tr>
<tr>
<td>( \quad x := x + 1; )</td>
<td>([1; 99])</td>
</tr>
<tr>
<td>}</td>
<td>([100; +\infty])</td>
</tr>
<tr>
<td>Program statement</td>
<td>Approx. state (on entry)</td>
</tr>
<tr>
<td>-------------------</td>
<td>-------------------------</td>
</tr>
<tr>
<td>$x := 1;$</td>
<td>$[0; 0]$</td>
</tr>
<tr>
<td>while ($x &lt; 100$)</td>
<td>$[1; +\infty] \triangledown [1; 100] = [1; +\infty]$</td>
</tr>
<tr>
<td>$x := x + 1;$</td>
<td>$[1; 99]$</td>
</tr>
<tr>
<td>}</td>
<td>$[100; +\infty]$</td>
</tr>
</tbody>
</table>

Stabilized
Example: Interval analysis with widening

Program statement

\[
\begin{align*}
x & := 1; \\
\text{while } (x < 100) \{ \\
x & := x + 1;
\}
\end{align*}
\]

Approx. state (on entry)

\[
\begin{align*}
[0; 0] & \supseteq [0; 0] \\
[1; \infty] & \supseteq [1; 100] \\
[1; 99] & \supseteq [1; 99] \\
[100; \infty] & \supseteq [100; 100]
\end{align*}
\]

Stabilized (but we overshot the fixed point)

Thanks to widening, we stabilize in a finite number of steps (read: we always halt)
Narrowing (improved overshooting)

We can compute the limit of the sequence:

\[ X_0 = \lim_{i} Y_i \]

\[ X_{i+1} = X_i \triangle F^\#(X_i) \]

where \( \triangle \) denotes the narrowing operator: an operator with the following properties:

- For all \( x, y : (x \triangle y) \sqsubseteq x \)
- For all \( x, y, z : (x \sqsubseteq y \land x \sqsubseteq z) \implies x \sqsubseteq (y \triangle z) \)
- For any chain \( Y_i \) the alternative chain defined as \( Y_0' = Y_0 \) and \( Y_{i+1}' = Y_i' \triangle Y_{i+1} \) stabilizes after a finite amount of steps.
Example: Narrowing our interval analysis

Program statement

\[
x := 1;
\]

\[
\text{while } (x < 100) \{
\]

\[
x := x + 1;
\]

\}

Approx. state (on entry)

\[
[0; 0]
\]

\[
[1; +\infty]
\]

\[
[1; 99]
\]

\[
[100; +\infty]
\]

Starting from the overshot fixed point...
Example: Narrowing our interval analysis

<table>
<thead>
<tr>
<th>Program statement</th>
<th>Approx. state (on entry)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x := 1; )</td>
<td>([0; 0])</td>
</tr>
<tr>
<td>( \text{while } (x &lt; 100) ) { }</td>
<td>([1; +\infty] \Delta [1; 100] = [1; 100])</td>
</tr>
<tr>
<td>( x := x + 1; )</td>
<td>([1; 99])</td>
</tr>
<tr>
<td></td>
<td>([100; +\infty])</td>
</tr>
</tbody>
</table>

Starting from the overshot fixed point...
Example: Narrowing our interval analysis

<table>
<thead>
<tr>
<th>Program statement</th>
<th>Approx. state (on entry)</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ x := 1; ]</td>
<td>[ [0; 0] ]</td>
</tr>
<tr>
<td>[ \text{while } (x &lt; 100) { ]</td>
<td>[ [1; +\infty] \triangle [1; 100] = [1; 100] ]</td>
</tr>
<tr>
<td>[ \quad x := x + 1; ]</td>
<td>[ [1; 99] ]</td>
</tr>
<tr>
<td>}</td>
<td>[ [100; 100] ]</td>
</tr>
</tbody>
</table>
Example: Narrowing our interval analysis

Program statement

\[ x := 1; \]

while \( (x < 100) \) {

\[ x := x + 1; \]

}  

Approx. state (on entry)

\[ [0; 0] \]

\[ [1; +\infty] \Delta [1; 100] = [1; 100] \]

\[ [1; 99] \]

\[ [100; 100] \]

Stabilized
Example: Narrowing our interval analysis

Program statement

\begin{align*}
x & := 1; \\
\text{while} \ (x < 100) \{ \\
\quad x & := x + 1; \\
\}
\end{align*}

Approx. state (on entry)

\begin{align*}
[0; 0] & \\
[1; +\infty] \triangle [1; 100] = [1; 100] & \\
[1; 99] & \\
[100; 100] & \\
\end{align*}

Stabilized (and we even found the fixed point!)
Example: Narrowing our interval analysis

Program statement | Approx. state (on entry)
--- | ---
\[ x := 1; \] | \([0; 0]\)
\[ \text{while } (x < 100) \{ \] | \([1; +\infty] \triangle [1; 100] = [1; 100]\)
\[ x := x + 1; \] | \([1; 99]\)
\[ \} \] | \([100; 100]\)

Stabilized (and we even found the fixed point!)

In general, narrowing will stabilize in a finite number of steps on a sound result (may not be the fixed point)
Some words on functional programming and OCaml
Why FP and OCaml?

We’ll use a functional programming language to implement these constructs.
Why FP and OCaml?

We’ll use a functional programming language to implement these constructs.

Why?
Why FP and OCaml?

We’ll use a functional programming language to implement these constructs.

Why?

→ It’s a good fit for the job
   □ Algebraic datatypes and pattern matching are great for this kind of language processing
   □ Microsoft’s static device driver verifier is written in OCaml
   □ ASTREÉ is written in OCaml

You are welcome to use Scala, Haskell, SML, F#, . . . if you prefer.
OCaml is an ML dialect

Hence it

☐ is expression-based, hence everything has a value
☐ is strongly typed
☐ is statically scoped
☐ has algebraic datatypes, lists, tuples, and pattern matching
☐ has higher-order functions
☐ . . .

In addition it includes some object-oriented extensions (hence the O in OCaml).
Compilers and IDEs

There is both
- a bytecode compiler (`ocamlc`) and
- an optimizing native code compiler (`ocamlopt`)
- a compiler to JavaScript (`js_of_ocaml`)

IDE-wise, for
- emacs I recommend tuareg-mode
- IntelliJ: you tell me!
- Eclipse people recommend: OCaIDE
  http://www.algo-prog.info/ocaide/
  http://www.cs.jhu.edu/~scott/pl/caml/ocaide.shtml
- VIM: OMLet
- _: please let me know of your findings
You bind values to names using `let`:

```ocaml
let a = 42
let b = "a string"
let c = (a, b, "third tuple elem")
let d = ["a"; "string"; "list"]
```

You also use `let` to declare functions:

```ocaml
let double x = x + x
```

**Catch 0:** function application binds stronger than addition: Hence `f x+1` parses as `(f x)+1`

**Catch 1:** recursive functions must be marked `rec`:

```ocaml
let rec fac n = match n with
  | 0 -> 1
  | n -> n * fac (n - 1)
```
The `let` token is also used for local declarations (`[]` is nil, `::` is cons):

```ocaml
let concat xs ys =
  let rec walk xs = match xs with
  | [] -> ys
  | x::xs' -> x::(walk xs')
  in
  walk xs
```

however without an `end` to finish the block.

Note how OCaml uses `match ... with` to discriminate (pattern match) on a value.

**Exercise:** write in OCaml a function `sumlist` of type `sumlist : int list -> int`
Tuples (and pairs) can be written without parens!

**Catch 2:** Semicolon ‘;’ separates list elements (rather than comma ‘,’). For example, compare the types of [1,2,3] and [1;2;3]

**Catch 3:** Algebraic datatypes lets us build new datatypes as sums and products:

```plaintext
type 'a tree = Leaf of 'a
             | Node of 'a tree * 'a tree
```

However the constructors must be capitalized otherwise it’s a parse error!

**Catch 4:** The evaluation order is unspecified — however the compiler uses right-to-left in practice(!)
OCaml modules

OCaml has a powerful module system with

- signatures (think interface) and
- functors (think module \( \rightarrow \) module function)

Example:

```ocaml
module Intset = Set.Make (struct
  type t = ... (* element type *)
  let compare = ... (* element comparison *)
end)
```
OCaml has a powerful module system with
- signatures (think interface) and
- functors (think module -> module function)

Example:
```ocaml
module Intset = Set.Make (struct
    type t = int
    let compare n1 n2 =
      if n1 = n2 then 0 else
       if n1 > n2 then 1 else -1
    end)
```
OCaml modules

OCaml has a powerful module system with

- signatures (think interface) and
- functors (think module -> module function)

Example:

```ocaml
module Intset = Set.Make (struct
  type t = int
  let compare n1 n2 =
    if n1 = n2 then 0 else
    if n1 > n2 then 1 else -1
  end)
```

Builtin maps are similar:

```ocaml
module Mymap = Map.Make (struct ... end)
```
We can separate the implementation and the interface of a module into two separate files `x.ml` and `x.mli`. This is equivalent to

```ocaml
module X: sig (* contents of file x.mli *) end = struct (* contents of file x.ml *) end
```

**Catch 5**: Files are lower-case, but their module names are capitalized. Hence, the module in file `set.ml` is referred to as `Set`.

If we write

```ocaml
module S = struct let f = ... end
```

in a file `foo.ml` then we (need to) refer to `f` as `Foo.S.f`
Relevant links

- Tutorial and toplevel in your browser
  http://try.ocamlpro.com/

- A nice OCaml community site with lots of info:
  http://ocaml.org/

- OCaml reference manual
  http://caml.inria.fr/pub/docs/manual-ocaml/

- Standard library documentation
  http://caml.inria.fr/pub/docs/manual-ocaml/libref/

- Jason Hickey’s online book

- Two mailing lists (beginner + main list)

- ...
Let’s code something!

Let’s implement

- a transition system interface,
- an instantiation thereof, and
- the transition function from the reachable states collecting semantics
Summary
We have covered

- The what and the how of the course
- The basics of abstract interpretation (transition systems, reachable states collecting semantics, Galois connections, . . . )
- A crash course in OCaml