A Gentle Introduction to Abstract Interpretation

Patrick Cousot

cims.nyu.edu/~pcousot

TASE 2015
The 9th International Symposium on Theoretical Aspects of Software Engineering

September 12—14, 2015 — Nanjing, China

Example of picture abstraction

Abstractions of a man / crowd

Height
Fingerprint
Eye color
DNA

Individual heights
min, max
Numerical abstractions used in Astrée

Collecting semantics:
partial traces

Intervals.
\( x \in [a, b] \)

Simple congruences:
\( x \equiv a[b] \)

Octagons:
\( \pm x \pm y \leq a \)

Ellipses.
\( x^2 + by^2 - axy \leq d \)

Exponentials:
\( -a^{bt} \leq y(t) \leq a^{bt} \)

% cat retro.c
typedef enum {FALSE = 0, TRUE = 1} BOOLEAN;
typedef enum {F=0,T=1} BOOL;
volatile BOOLEAN switch;
volatile float E;
float P, X, A, B;

void dev() {
    X=E;
    if (FIRST) { P = X; }
    else {
        P = (P - (((2 * P) - A) - B) * 5.0e-05) + (S[1]*1.5) - (S[1]*0.7);
    }
}

void main() {
    FIRST = TRUE;
    while (TRUE) {
        dev( );
        FIRST = FALSE;
        _ASTREE_wait_for_clock();
    }
}

% cat retro.config

#define _ASTREE_volatile_input(E [15.0, 15.0])
#define _ASTREE_volatile_input(SWITCH [0,1])
#define _ASTREE_max_clock(36000000)

astree -exec-fn main -config-sem retro.config

cat retro.c | grep "#[^]*" | tail -1

The boolean relation abstract domain is parameterized by the height of the decision tree (an analyzer option) and the abstract domain at the leafs

void filter () {
    static float E[2], S[2];
    if (INIT) { S[0] = X; P = X; E[0] = X; }
    else { P = (((0.5 * X) - (E[0] * 0.7)) + (E[1] * 0.4)) + (S[0] * 1.5) - (S[1] * 0.7)); }
    E[1] = E[0]; E[0] = X; S[1] = S[0]; S[0] = P;
/* S[0], S[1] in [-1327.02698354, 1327.02698354] */
}

void main () {
    X = 0.2 * X + 5; INIT = TRUE;
    while (1) {
        X = 0.9 * X + 35; /* simulated filter input */
        filter (); INIT = FALSE;
    }
}
Reduction of Abstractions

Example: reduction of intervals [CC76] by simple congruences [Gra89]

```c
% cat -n congruence.c
1 /* congruence.c */
2 int main()
3 {
4     int X;
5     X = 0;
6     while (X <= 128)
7             X = X + 4;
8     __ASTREE_log_vars(X);
9 }

% astree congruence.c -no-relational -exec-fn main | k grep "(WARN)\(X in\)"

direct = <integers (intv+cong+bitfield+set): X in {132}>

Intervals : X ∈ [129,132] + congruences : X = 0 mod 4 ⟷ X ∈ {132}.
```

Examples of Programs Analyzed by Astrée

- Automatic proofs of absence of runtime errors in Electric Flight Control Software:
  - A340/600: 132.000 lines of C, 40mn on a PC 2.8 GHz, 300 Mb (Nov. 2003)
  - A380: 1.000.000 lines of C, 34h, 8 Gb (Nov. 2005)
  - no false alarm, World premières!

- Automatic proofs of absence of runtime errors in the ATV software:\(^{2}\):
  - C version of the automatic docking software: 102.000 lines of C, 23s on a Quad-Core AMD Opteron™ processor, 16 Gb (Apr. 2008)

\(^{2}\) the Jules Verne Automated Transfer Vehicle (ATV) enabling ESA to transport payloads to the International Space Station.

Examples of Static Analyzers in Industrial Use

- For C critical synchronous embedded control/command programs (for example for Electric Flight Control Software)
  - aiT [FHL+01] is a static analyzer to determine the Worst Case Execution Time (to guarantee synchronization in due time)

- Astrée [BCC+03] is a static analyzer to verify the absence of runtime errors

Content

2.1 Mathematical Semantics ..........................
2.2 Mathematical Invariants ..........................
2.3 Mathematical Invariant Equations ...............
2.4 Solutions to the Mathematical Invariant Equations ...
2.5 Solving the Fixpoint Equations by Infinite Iteration ...
2.6 Machine Invariants ..........................
2.7 Interval Abstraction ..........................
2.8 An Interval Abstract Interpreter .................
2.9 Finite but Slow Iteration ........................
2.10 Convergence Speed Up ..........................
2.11 Convergence Acceleration ......................
    2.11.1 Convergence Acceleration with Widening  
    2.11.2 Convergence Acceleration with Narrowing
2.12 Chaotic and Structural Iteration ...............
2.13 Verification ..........................
In this gentle introduction to Abstract Interpretation

Mathematical Semantics

A sample program

Let us start with the following example program.

\[ P \triangleq 1x := 1; \text{while } true \text{ do } 3x := (x + 1); \text{ od}^4. \]

The mathematical semantics of this program can be informally described as follows.

- Execution start at program point 1 by assigning 1 to program variable \( x \) and goes on at program point 2.
- When at program point 2 the evaluation of the loop test yields the value true so execution continues at program 3 where the value of variable \( x \) is incremented by 1 before coming back to 2.
- Since the loop condition is never false, program point 4 is unreachable so program execution never ends.

A sample program

Let us start with the following example program.

\[ P \triangleq 1x := 1; \text{while } true \text{ do } 3x := (x + 1); \text{ od}^4. \]
More formally, we write \( (\ell, x) \) for the state of program execution where execution is at program point \( \ell, \ell = 1, 2, 3, 4 \), and variable \( x \) has integer value \( x \in \mathbb{Z} \) (where \( \mathbb{Z} \) is the set of all mathematical integers).

**Trace semantics**

\[
P \triangleq 1x := 1; \text{while}^{2}\text{true} \text{do} 3x := (x + 1); \text{od}^4.
\]

So the set of all such execution traces is

\[
\{ (1, z)^{(2)} 1^{(3)}, 1^{(2)}, 2^{(3)}, 2 \ldots (2, i)^{(3)}, i^{(2)}, i + 1 \ldots | z \in \mathbb{Z} \}
\]

**Execution trace**

\[
P \triangleq 1x := 1; \text{while}^{2}\text{true} \text{do} 3x := (x + 1); \text{od}^4.
\]

A complete program execution can be described by the following execution trace which is an infinite sequence of states

\[
(1, z)^{(2)} 1^{(3)}, 1^{(2)}, 2^{(3)}, 2 \ldots (2, i)^{(3)}, i^{(2)}, i + 1 \ldots
\]

where \( z \in \mathbb{Z} \) can be any initial integer value of \( x \).

**Mathematical Invariants**
Mathematical Invariant Equations

Let us now consider an abstraction of the set of all possible execution traces, which consists in remembering for each program point \( \ell \), \( \ell = 1, 2, 3, 4 \) the set \( I_\ell \) of possible values that can be taken by variable \( x \) when execution reaches program point \( \ell \) along any of these traces.

**Traces to invariants abstraction**

\[
\alpha(T) = \lambda l.\{ x \mid \exists \sigma, \sigma' : \sigma(l, x)\sigma' \in T \}
\]

The abstraction \( \alpha \) maps a set \( T \) of traces to a map \( \alpha(T) \) from program points \( l \) to the set \( \alpha(T)_l \) of reachable values \( x \) of program variable \( x \) during any possible execution in \( T \).
**Invariance Equations**

$P \triangleq 1 \text{x := 1; while } 2 \text{true do } 3 \text{x := (x + 1); od} 4$.

Observe that the set $l_\ell$ of possible values of variable x at program point $\ell = 1, 2, 3, 4$ satisfies the following conditions.

$$
\begin{align*}
    &x_1 = Z \\
    &x_2 = \{1\} \cup \{x + 1 \mid x \in x_3\} \\
    &x_3 = x_2 \cap \{x \in Z \mid \text{true}\} \\
    &x_4 = x_2 \cap \{x \in Z \mid \text{false}\}
\end{align*}
$$

(3.1)

**Fixpoint Equations**

These conditions can be understood as a system of fixpoint equations $X = f(X)$ of the form

$$
\begin{align*}
    &X_i = f_i(X_1, \ldots, X_4) \\
    &i = 1, \ldots, 4
\end{align*}
$$

with unknowns $X = (X_1, \ldots, X_4)$.

**Fixpoint Solutions**

$$
\begin{align*}
    &x_1 = Z \\
    &x_2 = \{1\} \cup \{x + 1 \mid x \in x_3\} \\
    &x_3 = x_2 \cap \{x \in Z \mid \text{true}\} \\
    &x_4 = x_2 \cap \{x \in Z \mid \text{false}\}
\end{align*}
$$

(3.1)

- At program point 1 the variable x can be initialized by any integer value $z \in Z$ and so $x_1 = Z$.

- At program point 2, either execution comes from program point 1 and so the value of variable x is 1 or execution comes from program point 2 and so the value of variable x is the value x that x had at this point 3 incremented by 1. So $x_2 = \{1\} \cup \{x + 1 \mid x \in x_3\}$.

- At program point 3, the possible values of x are those at point 2 for which the loop condition is true so $x_3 = x_2 \cap \{x \in Z \mid \text{true}\} = x_2$.

- At program point 4, the possible values of x are those at point 2 for which the loop condition is false so $x_4 = x_2 \cap \{x \in Z \mid \text{false}\} = \emptyset$.

So solving this system of equations might lead to the desired invariant I.

However these equations do not have a unique solution. For example $x_1 = x_2 = x_3 = Z$ and $x_4 = \emptyset$ is another solution which is larger for componentwise set inclusion $\subseteq$.

So we will prefer the smallest solution (called the least fixpoint lfp $f$), which is included in all other solutions 1 and turns out to be I.

1by Tarski fixpoint theorem
Tarski’s fixpoint theorem

**Fixpoint**
- Let $S$ be a set
- Let $F$ be a function $F : S \rightarrow S$
- A **fixpoint** of $F$ is $x \in S$ such that $x = F(x)$
- i.e. a solution to the equation

**Least fixpoint**
- Let $\langle S, \leq \rangle$ be a set partially ordered by $\leq$
- The least fixpoint, if any, of $F : S \rightarrow S$ is
  - a fixpoint $x = F(x)$
  - $\leq$-smaller than any other fixpoint $y = F(y) \quad \Rightarrow \quad x \leq y$
- Notation: $\text{Lfp } F$
Solving the Equations iteratively ...

The least solution \( l = \text{ifp} f \) of \( X = f(X) \) for \( \subseteq \) can be calculated iteratively, essentially by enumeration of all possible states reachable from the initial states.

Solving the equations iteratively ... (Cont’d)

\[- X^0 = (X_1^0, X_2^0, X_3^0, X_4^0) = (\emptyset, \emptyset, \emptyset) \quad \text{(starting with the smallest possible approximation)}]
Solving the equations iteratively ... (Cont’d)

- \( X^0 = \langle X^0_1, X^0_2, X^0_3, X^0_4 \rangle = \langle \emptyset, \emptyset, \emptyset, \emptyset \rangle \) (starting with the smallest possible approximation)
- \( X^1 = \langle X^1_1, X^1_2, X^1_3, X^1_4 \rangle = f(X^0) = \langle Z, \{1\} \cup \{x + 1 \mid x \in X^0_3\}, X^0_2 \cap \{x \in Z \mid \text{true}\}, X^0_2 \cap \{x \in Z \mid \text{false}\} \rangle = \langle Z, \{1\}, \emptyset, \emptyset \rangle \)
- \( X^2 = \langle X^2_1, X^2_2, X^2_3, X^2_4 \rangle = f(X^1) = \langle Z, \{1\} \cup \{x + 1 \mid x \in X^1_3\}, X^1_2 \cap \{x \in Z \mid \text{true}\}, X^1_2 \cap \{x \in Z \mid \text{false}\} \rangle = \langle Z, \{1, 1\}, \emptyset, \emptyset \rangle \)
- \( X^3 = \langle X^3_1, X^3_2, X^3_3, X^3_4 \rangle = f(X^2) = \langle Z, \{1\} \cup \{x + 1 \mid x \in X^2_3\}, X^2_2 \cap \{x \in Z \mid \text{true}\}, X^2_2 \cap \{x \in Z \mid \text{false}\} \rangle = \langle Z, \{1, 2\}, \{1\}, \emptyset \rangle \)

This calculation can go on like this ad infinitum since each iteration \(X^{i+1} = f(X^i)\) of the equations corresponds to an iteration in the program loop and so adds one more possible value of variable \(x\) at program point \(Z\). The solution is to use mathematical induction which requires to invent the following inductive hypothesis:

Solving the equations iteratively ... (Cont’d)

- \( X^0 = \langle X^0_1, X^0_2, X^0_3, X^0_4 \rangle = \langle \emptyset, \emptyset, \emptyset, \emptyset \rangle \) (starting with the smallest possible approximation)
- \( X^1 = \langle X^1_1, X^1_2, X^1_3, X^1_4 \rangle = f(X^0) = \langle Z, \{1\} \cup \{x + 1 \mid x \in X^0_3\}, X^0_2 \cap \{x \in Z \mid \text{true}\}, X^0_2 \cap \{x \in Z \mid \text{false}\} \rangle = \langle Z, \{1\}, \emptyset, \emptyset \rangle \)
- \( X^2 = \langle X^2_1, X^2_2, X^2_3, X^2_4 \rangle = f(X^1) = \langle Z, \{1\} \cup \{x + 1 \mid x \in X^1_3\}, X^1_2 \cap \{x \in Z \mid \text{true}\}, X^1_2 \cap \{x \in Z \mid \text{false}\} \rangle = \langle Z, \{1\}, \{1\}, \emptyset \rangle \)
- \( X^3 = \langle X^3_1, X^3_2, X^3_3, X^3_4 \rangle = f(X^2) = \langle Z, \{1\} \cup \{x + 1 \mid x \in X^2_3\}, X^2_2 \cap \{x \in Z \mid \text{true}\}, X^2_2 \cap \{x \in Z \mid \text{false}\} \rangle = \langle Z, \{1, 2\}, \{1\}, \emptyset \rangle \)

The solution is to use mathematical induction which requires to invent the following inductive hypothesis:
Solving the equations iteratively ... (Cont’d)

- \( X^{2n} = \langle X_1^{2n}, X_2^{2n}, X_3^{2n}, X_4^{2n} \rangle = \langle Z, \{1, \ldots, n\}, \emptyset \rangle \)  
  induction hypothesis which holds for the basis \( n = 1 \)

- \( X^{2n+1} = \langle X_1^{2n+1}, X_2^{2n+1}, X_3^{2n+1}, X_4^{2n+1} \rangle = f(X^{2n}) = \langle Z, \{1\} \cup \{x + 1 \mid x \in X_3^{2n}\}, X_2^{2n} \cap \{x \in Z \mid \text{true}\}, X_2^{2n} \cap \{x \in Z \mid \text{false}\} \rangle = \langle Z, \{1, \ldots, n + 1\}, \emptyset \rangle \)

- By recurrence on \( n \), we have proved that

\[ \forall n : X^{2n} = \langle X_1^{2n}, X_2^{2n}, X_3^{2n}, X_4^{2n} \rangle = \langle Z, \{1, \ldots, n\}, \{1, \ldots, n\}, \emptyset \rangle \]

Solving the equations iteratively ... (Cont’d)

- \( X^{2n} = \langle X_1^{2n}, X_2^{2n}, X_3^{2n}, X_4^{2n} \rangle = \langle Z, \{1, \ldots, n\}, \emptyset \rangle \)
  induction hypothesis which holds for the basis \( n = 1 \)

- \( X^{2n+1} = \langle X_1^{2n+1}, X_2^{2n+1}, X_3^{2n+1}, X_4^{2n+1} \rangle = f(X^{2n}) = \langle Z, \{1\} \cup \{x + 1 \mid x \in X_3^{2n}\}, X_2^{2n} \cap \{x \in Z \mid \text{true}\}, X_2^{2n} \cap \{x \in Z \mid \text{false}\} \rangle = \langle Z, \{1, \ldots, n + 1\}, \emptyset \rangle \)

- By recurrence on \( n \), we have proved that

\[ \forall n : X^{2n} = \langle X_1^{2n}, X_2^{2n}, X_3^{2n}, X_4^{2n} \rangle = \langle Z, \{1, \ldots, n\}, \{1, \ldots, n\}, \emptyset \rangle \]
Solving the equations iteratively ... (Cont’d)

- \(X^{2n} = (X_1^{2n}, X_2^{2n}, X_3^{2n}, X_4^{2n}) = (Z, \{1, \ldots, n\}, \{1, \ldots, n\}, \emptyset)\)

induction hypothesis which holds for the basis \(n = 1\)

- \(X^{2n+1} = (X_1^{2n+1}, X_2^{2n+1}, X_3^{2n+1}, X_4^{2n+1}) = f(X^{2n}) = (Z, \{1\} \cup \{x + 1 \mid x \in X_3^{2n}\}, X_2^{2n} \cap \{x \in Z \mid \text{true}\}, \{x \in Z \mid \text{false}\}) = (Z, \{1, \ldots, n + 1\}, \{1, \ldots, n\}, \emptyset)\)

By recurrence on \(n\), we have proved that

\(\forall n : X^{2n} = (X_1^{2n}, X_2^{2n}, X_3^{2n}, X_4^{2n}) = (Z, \{1, \ldots, n\}, \{1, \ldots, n\}, \emptyset)\)

\(\Rightarrow\) A fundamental property of the invariants equations \(X = f(X)\) is that \(f\) is increasing.

This means that if \(X \subseteq Y\) then \(f(X) \subseteq f(Y)\) where \((X_1, \ldots, X_n) \subseteq (Y_1, \ldots, Y_n)\) if and only if \(\forall i \in [1, n] : X_i \subseteq Y_i\).

The intuition is that if more states can be reached at some program point then more states will be reachable at next program point.

It follows that the iterates form an ascending chain meaning \(X^0 \subseteq X^1 \subseteq \ldots \subseteq X^n \subseteq X^{n+1} \subseteq \ldots \subseteq \lim_{n\to\infty} X^n = \text{lfp} f\).

---

Machine Invariants
Machine Integers

- No computer can represent any, arbitrary large, integer. In practice integer variables like \( x \) take their values in an interval \([\text{min\_int}, \text{max\_int}]\) where \( \text{min\_int} < 0 < \text{max\_int} \) are machine dependant.\(^2\)
- It follows that we have to decide what happens in case of overflow when evaluating expression \( (x + 1) \).
- We will assume that execution immediately stops in case of integer overflow.\(^3\)

\(^2\)e.g. in two’s complement representation on 64 bits, we have generally have \( \text{min\_int} = -2147483648 \) and \( \text{max\_int} = 2147483647 \).

\(^3\)Which is a rather simplifying hypothesis since most computers will go on providing a result modulo \( \text{max\_int} \) so that e.g. \( \text{max\_int} + 1 = \text{min\_int} \) in two’s complement representation.

Machine Invariant Equations

\[ P \triangleq 1x := 1; \text{while} 2\text{true do } 3x := (x + 1); \text{od}^4. \]

It follows that the machine invariant satisfies the following equations

\[
\begin{align*}
X_1 &= [\text{min\_int}, \text{max\_int}] \\
X_2 &= \{1\} \cup \{x + 1 \in [\text{min\_int}, \text{max\_int}] \mid x \in X_3\} \\
X_3 &= X_2 \cap \{x \in [\text{min\_int}, \text{max\_int}] \mid \text{true}\} \\
X_4 &= X_2 \cap \{x \in [\text{min\_int}, \text{max\_int}] \mid \text{false}\}
\end{align*}
\]

\[ (3.2) \]

Machine states and execution traces

Hence the set of program states \( S \triangleq \{1, 2, 3, 4\} \times [\text{min\_int}, \text{max\_int}]\) is now finite and the execution traces are now finite of the form

\[ \{(1, z)(2, 1)...(2, i)(3, i)(2, i+1)...(3, \text{max\_int}) \mid z \in [\text{min\_int}, \text{max\_int}]\}. \]

Convergence

- Now the convergence of the iterations is guaranteed but is so slow that it cannot be of any practical use, but for programs with very few program variables.
- Moreover, mathematical sets of integers can be arbitrarily complex hence very expensive to represent in computer memory which is likely to produce memory overflows after lengthy computations, a flaw of all program verification methods based upon the exhaustive enumeration of all possible cases.
Interval Abstraction

- A further abstraction must be used to solve the machine invariant computer representation problem.

We will use intervals \([l, h] \triangleq \{x \in \mathbb{Z} \mid l \leq x \leq h\}\) with the convention that \([l, h] = \emptyset\) whenever \(h < l\). In doing so we perform an approximation of a non-empty set \(X \subseteq [\min_{\text{int}}, \max_{\text{int}}]\) by the interval \([\min X, \max X]\).
Interval Abstraction

- A further abstraction must be used to solve the machine invariant computer representation problem.
- We will use intervals \([l, h] \triangleq \{x \in \mathbb{Z} \mid l \leq x \leq h\}\) with the convention that \([l, h] = \emptyset\) whenever \(h < l\).
- In doing so we perform an approximation of a non-empty set \(X \subseteq [\min, \max]\) by the interval \([\min X, \max X]\).
- This approximation is sound in that whenever the value of variable \(x\) belongs to a set \(X_i\) whenever execution reaches program point \(i\), it definitely also belongs to the set \([\min X_i, \max X_i]\).

Traces to intervals abstraction

\[
\alpha(T) = \lambda l.\text{let } X = \{ x \mid \exists \sigma, \sigma': \sigma(l, x)\sigma' \in T \} \text{ in } [\min X, \max X]
\]

The abstraction \(\alpha\) maps a set \(T\) of traces to a map \(\alpha(T)\) from program points \(l\) to the pair \((m, M) = \alpha(T)l\) of minimal \(m\) and maximal \(M\) reachable values \(x\) of program variable \(x\) during any possible execution in \(T\).

Interval Invariance Equations

\[
P \triangleq 1 x := 1; \text{while}^2 \text{true do} 3 x := (x + 1); \text{od}^4.
\]

The interval invariance equations are now:

\[
\begin{cases}
X_1 &= [\min, \max] \\
X_2 &= [1, 1] \cup \{ X_3 = \emptyset \cup \emptyset : \text{let} (a, b) = X_3 \text{ in } \{[\min(a + 1, \max), \min(b + 1, \max)]\} \}
\end{cases}
\]

\[
X_3 = X_2 \cap [\min, \max]
\]

\[
X_4 = X_2 \cap \emptyset
\]
Interval Operations

- where the interval join is \( \emptyset \cup \emptyset \triangleq \emptyset, \emptyset \cup [l, h] \triangleq [l, h], \) and
  \[ [a, b] \cup [c, d] \triangleq [\min(a, c), \max(b, d)] \]
- and the interval meet is \( \emptyset \cap \emptyset \triangleq \emptyset, \emptyset \cap [l, h] \triangleq [l, h], \) and
  \[ [a, b] \cap [c, d] \triangleq [\max(a, c), \min(b, d)] \]
  where the interval join is
  \[ [a, b] \cup [c, d] \triangleq [\min(a, c), \max(b, d)] \]
  and the interval meet is
  \[ [a, b] \cap [c, d] \triangleq [\max(a, c), \min(b, d)] \]
  when \( b \geq c \) and \( d \geq a \)
  \[ [a, b] \cap [c, d] \triangleq \emptyset \]
  when \( b < c \) or \( d < a \)

Over-approximation

- The interval equations over-estimate the machine invariant in than they will provide in general more states that possible in actual program executions.
- For example the set \( \{1, 2, 5\} \) will be overapproximated by \( [1, 5] \) which introduces the spurious values 3 and 4.
- Notice that overapproximation preserve invariance. For example if the values of variable \( x \) are always greater than one at some program point then they are certainly positive (although the value 0 is spurious).

Example of incorrect approximations

For \( x \in \{1, 2, 5\} \)
- Underapproximations (such as \( x \) are always greater than 10) would be incorrect.
- Similarly, incomparable approximations (such as \( x \) is negative) are also unsound. In particular the interval join \( \cup \) overapproximates the interval union \( \cup \) and the interval meet \( \cap \) overapproximates the interval intersection \( \cap \).
Objective

- We now briefly sketch the design and functional encoding in OCAML of the interval abstract interpreter.
- Such an interval abstract interpreter reads any program, builds the interval invariance equations, and then solves them.
- For simplicity, we concentrate on the second part and will provide encodings of the interval invariance equations manually.

The Interval Abstract Domain

- We first encode the interval abstract domain, implementing a computer representation of abstract interval properties with a type interval (where EMPTY encodes the empty set ∅). In OCAML, we have max_int = 1073741823 and min_int = –1073741824.
- We also encode the basic interval operations ∈ (less, interval inclusion), ∪ (interval join), ∩ (interval meet), interval printing (print) and interval incrementation (add1).
- Of course many more interval operations are needed to handle a full language, but we aim at extreme simplicity.

\[ \text{One of the } 64 \text{ bits is used for garbage collection.} \]
\[ \text{or max_int = 4611686018427387903 depending on the machine/compiler} \]

Abstract Environments

- For programs with more than one variable, we would have to encode an abstract environment assigning intervals to program variables.
- Writing \[ X = \{ x_1 \leftarrow v_1, \ldots, x_n \leftarrow v_n \} \] for the function \( X \) mapping \( x_i \) to \( v_i \) such that \( X(x_i) = v_i, i = 1, \ldots, n \), the interval invariance equations would be

\[
\begin{align*}
X_1 & = \{ x \leftarrow [\text{min_int}, \text{max_int}] \} \\
X_2 & = \{ x \leftarrow [1, 1] \cup \{ x_3(x) = \emptyset \ ? \emptyset \ ? \text{let}\left[ a, b \right] = X_3(x) \text{ in} \\
& \quad \{ \min(a + 1, \text{max_int}), \min(b + 1, \text{max_int}) \} \} \\
X_3 & = X_2 \cap \{ x \leftarrow [\text{min_int}, \text{max_int}] \} \\
X_4 & = X_2 \cap \{ x \leftarrow \emptyset \}
\end{align*}
\]

where the abstract operations are extended pointwise such as \( \{ x_1 \leftarrow v_1, \ldots, x_n \leftarrow v_n \} \cap \{ x_1 \leftarrow v_1', \ldots, x_n \leftarrow v_n' \} = \{ x_1 \leftarrow v_1 \cap v_1', \ldots, x_n \leftarrow v_n \cap v_n' \} \).
- Since our example has only one variable, this boils down to using the interval abstract domain (and leaving implicit the variable name \( x \)).
Abstract Invariants

- Then we have to encode an abstract domain for representing abstract invariants ($X^1$, $X^2$, $X^3$, $X^4$) which attach to each program point $i$ an abstract local invariant $X^i$ which holds whenever controls reaches program point $i$.
- Each abstract local invariant $X^i$ is represented by an abstract environment (abstract intervals in our simplified case).
- The encoding is very simple as a 4-tuple specifying the value of program variable $x$ at each program point ($1$, $2$, $3$, $4$).

• terminate (or terminates with a runtime error, e.g. out of memory).

\[
\begin{aligned}
&\text{let } cless (x_1,x_2,x_3,x_4) (x'_1,x'_2,x'_3,x'_4) = \\
&\quad (\text{less } x_1 x'_1, \text{less } x_2 x'_2, \text{less } x_3 x'_3, \text{less } x_4 x'_4); \\
&\text{let } pless x x' = \\
&\quad \text{let b1, b2, b3, b4 = cless x x' in} \\
&\quad b1 \land b2 \land b3 \land b4; \\
&\text{let } pgreater x x' = \text{pless } x' x; \\
&\text{let } pbot = (\text{EMPTY}, \text{EMPTY}, \text{EMPTY}, \text{EMPTY}); \\
&\text{let } pmeet (x_1,x_2,x_3,x_4) (x'_1,x'_2,x'_3,x'_4) = \\
&\quad (\text{meet } x_1 x'_1, \text{meet } x_2 x'_2, \text{meet } x_3 x'_3, \text{meet } x_4 x'_4); \\
&\text{let } \text{pprint } (x_1,x_2,x_3,x_4) = \\
&\quad \text{print_string } "1:"; \text{print } x_1; \text{print_string } "2:"; \\
&\quad \text{print } x_2; \text{print_string } "3:"; \text{print } x_3; \\
&\quad \text{print_string } "4:"; \text{print } x_4; \text{print_newline }();
\end{aligned}
\]

Abstract Invariants (Cont’d)

- We essentially have to represent the logical structure, which boils down to
  - the partial order $\subseteq$ (pless), encoding abstract implication ($\subseteq$ in set theory and $\Rightarrow$ in logic);
  - $\supseteq$ (pgreater), the abstract inverse implication ($\supseteq$ in set theory and $\Leftarrow$ in logic);
  - the pointwise infimum ($\cap$) (pbot), the abstract encoding of false,
  - the pointwise meet (for later use in section 3.9), and
  - the printing of local abstract invariants attached to program points (pprint).

The Iterator

- Next the iterator module implements the iterative computation of the least solution of the invariance equations ($\text{lfp}$).\(^3\)
- It is parameterized by the order (leq), the starting point (a) and the abstract transformer ($f$) so as to compute $a$, $f(a)$, $f^2(a)$, ..., $f^n(a)$, ..., until reaching the limit $f^e(a)$ such that $f(f^e(a)) \subseteq f^e(a)$.
- Of course, convergence may not be guaranteed in which case lfp does not terminate (or terminates with a runtime error, e.g. out of memory).

\(^3\)least fixpoint.
Abstract Invariant Equations \( X = f(X) \)

Then we encode the abstract reachable state transformer \( f(X) = f((X_1, \ldots, X_4)) \) using the environment abstract domain (the intervals in our simplified case).

\[
\begin{align*}
X_1 &= [\text{min\_int}, \text{max\_int}] \\
X_2 &= [1, 1] \sqcup (X_3 = \emptyset ? \emptyset : \text{let}\{a, b\} = X_3 \text{ in} \\
&\quad \text{min}(a + 1, \text{max\_int}), \text{min}(b + 1, \text{max\_int})) \\
X_3 &= X_2 \cap [\text{min\_int}, \text{max\_int}] \\
X_4 &= X_2 \cap \emptyset
\end{align*}
\]

The Abstract Interpreter

The abstract interpreter performs the iterative abstract reachability fixpoint computation and prints the least fixpoint result.

\[
\begin{align*}
(* \text{reachability interval analysis} *) \\
\text{open Interval} \\
\text{open Invariant} \\
\text{let analyzer()} = \text{pprint}(\text{lfp pless pbot } f); \\
\text{analyzer}();
\end{align*}
\]
Iterative Resolution of the Interval Equations

Because the abstract domains are finite, the static analysis will always terminate. In our case, after more that 40mn of computation, we get

```
% ocamlc interval.ml invariant.ml transformerUnbounded.ml iterator.ml

% time ./a.out
1:(-1073741824,1073741823) 2:(1,1073741823) 3:(1,1073741823) 4:--]
2977.460u 9.632s 50:43.46 98.1% 0+0k 0+0io 0pf+0w
%
```

6On a MacBook Pro with Intel Core 2 Duo at 2.6 GHz.

---

Infinitary Iteration

A look at the iterates...

The Jacobi iterates are as follows

```
% ocamlc interval.ml invariant.ml transformerUnbounded.ml \ iterator.ml

% time ./a.out
1:--] 2:--] 3:--] 4:--]
1:(-1073741824,1073741823) 2:(1,1) 3:--] 4:--]
1:(-1073741824,1073741823) 2:(1,1) 3:(1,1) 4:--]
1:(-1073741824,1073741823) 2:(1,2) 3:(1,1) 4:--]
1:(-1073741824,1073741823) 2:(1,2) 3:(1,2) 4:--]
1:(-1073741824,1073741823) 2:(1,3) 3:(1,2) 4:--]
1:(-1073741824,1073741823) 2:(1,3) 3:(1,3) 4:--]
...
1:(-1073741824,1073741823) 2:(1,1073741823) 3:(1,1073741823) 4:--]
1:(-1073741824,1073741823) 2:(1,1073741823) 3:(1,1073741823) 4:--]
1:(-1073741824,1073741823) 2:(1,1073741823) 3:(1,1073741823) 4:--]
1:(-1073741824,1073741823) 2:(1,1073741823) 3:(1,1073741823) 4:--]
1:(-1073741824,1073741823) 2:(1,1073741823) 3:(1,1073741823) 4:--]
1:(-1073741824,1073741823) 2:(1,1073741823) 3:(1,1073741823) 4:--]
converged to lfp.
1:(-1073741824,1073741823) 2:(1,1073741823) 3:(1,1073741823) 4:--]
3115.012u 7.706s 52:49.34 98.5% 0+0k 0+0io 0pf+0w
```

On the Convergence Criterion

- Notice that the abstract invariance equations $X = f(X)$ are increasing, if $X \subseteq Y$ then $f(X) \subseteq f(Y)$. 
On the Convergence Criterion

- Notice that the abstract invariance equations $X = f(X)$ are increasing, if $X \sqsubseteq Y$ then $f(X) \sqsubseteq f(Y)$.
- The intuition is that the interval of possible value of a variable is larger at a program point, it should be also larger at the next program point.
- Since the abstract interpreter stops iterating when reaching of postfixpoint $f(\lim_{n \to +\infty} X^n) \sqsubseteq \lim_{n \to +\infty} X^n$, the limit satisfies $f(\lim_{n \to +\infty} X^n) = \lim_{n \to +\infty} X^n$ by antisymmetry.

On Slow Convergence!

Of course the convergence is extremely slow and in practice must be accelerated.
Convergence Acceleration

Objective

- When convergence requires infinitely many steps or is very slow, it may not be possible, due to undecidability or high complexity, to exactly calculate the least solution to the abstract system of equations. (*)

The only sound solution is then to have overapproximations of the desired result.

(*) Of course direct solutions do sometimes exist e.g. linear equations on regular languages

---

Objective

- When convergence requires infinitely many steps or is very slow, it may not be possible, due to undecidability or high complexity, to exactly calculate the least solution to the abstract system of equations. (*)

- The only sound solution is then to have overapproximations of the desired result.
- We have already exploited the overapproximation idea when replacing sets of integer values in the invariant equations by interval of values.
Objective

- When convergence requires infinitely many steps or is very slow, it may not be possible, due to undecidability or high complexity, to exactly calculate the least solution to the abstract system of equations. (*)
- The only sound solution is then to have overapproximations of the desired result.
- We have already exploited the overapproximation idea when replacing sets of integer values in the invariant equations by interval of values.
- We now exploit the approximation idea a second time now while computing the solution of the invariance equations.

(*) Of course direct solutions do sometimes exist e.g. linear equations on regular languages

Convergence Acceleration by Widening

- The intuition for convergence acceleration is to speed up the increasing iteration $X^0 = \bot, \ldots, X^{n+1} = f(X^n), \ldots, \lim_{n \to +\infty} X^n$ so as to reach an overapproximation $\hat{A}$ of the least solution $\lim_{n \to +\infty} X^n$ of the fixpoint equation $X = f(X)$.

\[\text{The justification is again by Tarski since } f(\hat{A}) \subseteq \hat{A} \text{ implies } \uparrow f \subseteq \hat{A}.\]
Convergence Acceleration by Widening

- The intuition for convergence acceleration is to speed up the increasing iteration $X^0 = \bot, \ldots, X^{n+1} = f(X^n)$, $\ldots$, $\lim_{n \to +\infty} X^n$ so as to reach an overapproximation $\hat{A}$ of the least solution $\lim_{n \to +\infty} X^n$ of the fixpoint equation $X = f(X)$\(^7\).
- Convergence acceleration means that $X^{n+1}$ will be a function of $X^n$ and $f(X^n)$\(^8\) and so $X^{n+1} = X^n \square f(X^n)$ where $\square$ is called a widening\(^9\).

\(^7\)The justification is again by Tarski, since $f(\hat{A}) \subseteq \hat{A}$ implies $\llcorner f \lrcorner \subseteq \hat{A}$.

\(^8\)and more generally $X^{n+1}$ could depend on the sequence of previous iterates $X^0, f(X^0), \ldots, X^n, f(X^n)$, but we can use a reencoding to prove that a proof by strong induction can always be done by a weak recurrence and inversely.

\(^9\)We use a binary operator notation rather than a functional notation because of the analogy between widenings $\square$ and joins $\lor$, $\lor$, etc.

Soundness

- For soundness, the widening must perform over-approximations, that is $x \subseteq x \downarrow y$ and $y \subseteq x \downarrow y$.

Example: Interval Widening

For example, a widening for intervals could be:

\[
\begin{align*}
\emptyset \downarrow y & \triangleq y \\
x \downarrow \emptyset & \triangleq x \\
[a, b] \downarrow [c, d] & \triangleq [(c < a \iff -\infty \leq a), (d > b \iff +\infty \leq b)]
\end{align*}
\]
Example: Interval Widening

For example, a widening for intervals could be

\[
\emptyset \bigtriangledown y \triangleq y \\
x \bigtriangledown \emptyset \triangleq x \\
[a, b] \bigtriangledown [c, d] \triangleq [\langle c < a \ ? -\infty \ ; a \rangle, \langle d > b \ ? +\infty \ ; b \rangle]
\]

- Recall that in \( x \bigtriangledown y \) the \( x \) is an iterate and \( y \) is the next iterate \( f(x) \). So in \( [a, b] \bigtriangledown [c, d] \) if \( c < a \) the next iterate decreases the lower limit of the interval so widening to \(-\infty\) ensures this cannot happen infinitely often.

Example: Interval Widening (Cont’d)

The extrapolation of bounds to infinity is illustrated on the following iteration (for two variables).

Example: Interval Widening

For example, a widening for intervals could be

\[
\emptyset \bigtriangledown y \triangleq y \\
x \bigtriangledown \emptyset \triangleq x \\
[a, b] \bigtriangledown [c, d] \triangleq [\langle c < a \ ? -\infty \ ; a \rangle, \langle d > b \ ? +\infty \ ; b \rangle]
\]

- Recall that in \( x \bigtriangledown y \) the \( x \) is an iterate and \( y \) is the next iterate \( f(x) \). So in \( [a, b] \bigtriangledown [c, d] \) if \( c < a \) the next iterate decreases the lower limit of the interval so widening to \(-\infty\) ensures this cannot happen infinitely often.

- Similarly, if \( d > b \) then the next iterate increases the upper limit of the interval so widening to \(+\infty\) ensures this cannot happen infinitely often. Moreover the widened interval is larger which ensures that we perform an overapproximation.

Widenings are not increasing!

- Observe that the interval widening is not increasing. For example \([0, 1] \subseteq [0, 2]\) but \([0, 1] \bigtriangledown [0, 2] = [0, +\infty] \not\subseteq [0, 2] = [0, 2] \bigtriangledown [0, 2] \), a point discussed at length in chapter 30.
Widenings are not increasing!

- Observe that the interval widening is not increasing. For example $[0, 1] \subseteq [0, 2]$ but $[0, 1] \vee [0, 2] = [0, \infty] \nsubseteq [0, 2] \vee [0, 2]$, a point discussed at length in chapter 30.
- It can be shown that if the widening stops losing information when a solution is found and is increasing then it cannot enforce termination (*)

(*) see P. Cousot, VMCAI 2015

Environment Widening

If we had abstract environments to handle several variables, the widening would have to be applied individually for each of these variables.

Encoding the interval widening

A functional encoding in of the widening in OCaml could be

(* intervalWidening.ml, interval widening *)
open Interval
let widen x y = match x,y with
    | EMPTY, _ -> y
    | _, EMPTY -> x
    | INT (a,b), INT (c,d) ->
        let a' = if c<a then min_int else a in
        let b' = if d>b then max_int else b in
        INT (a',b');;

Invariant widening

$P \triangleq 1x := 1; \text{while} 2\text{true do } 3x := (x + 1); \text{od}^4$.

We must also extend the widening to local invariants attached to program points. In our example, the widening is applied once around the loop at program point 2 as follows.

(* invariantWidening.ml, invariant widening *)
open IntervalWidening
let pwiden (x1,x2,x3,x4) (x'1,x'2,x'3,x'4) =
    (x'1,widen x2 x'2,x'3,x'4);;
Trace of the Iterations with Widening

The Jacobi iterates with widening are extremely fast as shown below.

```ocaml
% ocamlc interval.ml intervalWidening.ml invariant.ml 
% invariantWidening.ml transformerUnbounded.ml iterator.ml 
% reachability_widening.ml
% time .a.out
1:|- 2:|- 3:|- 4:|- 1:|- (1073741824,1073741823) 2:|(1,1) 3:|- 4:|- 1:|- (1073741824,1073741823) 2:|(1,1) 3:|- 4:|- 1:|- (1073741824,1073741823) 2:|(1,1073741823) 3:|(1,1) 4:|- 1:|- (1073741824,1073741823) 2:|(1,1073741823) 3:|(1,1073741823) 4:|- converged to lfp.
%```

Imprecision of the Widening

Of course, the widening cannot, in general, provide the exact result! To see that, consider the bounded iteration

\[ P \triangleq 1x := 1 ; \text{while} ^2 (x \leq 100) \text{do} ^3x := (x + 1); \text{od} ^4 . \]

so that the abstract interval equations become

\[
\begin{align*}
X_1 &= \{ x \leftarrow [\text{min\_int}, \text{max\_int}] \} \\
X_2 &= \{ x \leftarrow [1, 1] \cup \{ X_3(x) = \emptyset \ ? \emptyset \ : \text{let} (a, b) = X_3(x) \text{ in} \ [\text{min}(a + 1, \text{max\_int}), \text{min}(b + 1, \text{max\_int})] \} \\
X_3 &= X_2 \cap \{ x \leftarrow [\text{min\_int}, 100] \} \\
X_4 &= X_2 \cap \{ x \leftarrow [101, \text{max\_int}] \}
\end{align*}
\]
I) Direct iteration (without widening)

A direct iteration

(* reachability interval analysis *)
open Invariant
open TransformerBounded
open Iterator
let analyzer () = pprint (lfp pless pbot f);
 analyzer ();;

yields

% ocamlc interval.ml invariant.ml transformerBounded.ml \
? iterator.ml reachability_bounded.ml
% time ./a.out
1:(1073741824,1073741823) 2:(1,101) 3:(1,100) 4:(101,101)
0.001u 0.000s 0:00.00 0.0% 0+0k 0+0io 0pf+0w
%

II) Iteration with widening

(* reachability analysis with widening *)
open Invariant
open InvariantWidening
open TransformerBounded
open Iterator
let analyzer () =
  let fw x = pwiden x (f x) in
  pprint (lfp pless pbot fw);
 analyzer ();;

we rapidly get a strictly less precise result.

% ocamlc interval.ml intervalWidening.ml invariant.ml \
? invariantWidening.ml transformerBounded.ml iterator.ml \
? reachability_widening_bounded.ml
% time ./a.out
1:(1073741824,1073741823) 2:(1,101) 3:(1,100) 4:(101,101)
0.000u 0.000s 0:00.00 0.0% 0+0k 0+0io 0pf+0w
%
Again convergence is guaranteed but slow.

In more details the widening effect is not compensated by the test on loop exit.

% ocamlc invariant.ml intervalWidening.ml invariant.ml \
? invariantWidening.ml transformerBounded.ml iteratorTrace.ml
% time ./a.out
1:(1073741824,1073741823) 2:(1,101) 3:(1,100) 4:(101,101)
0.000u 0.000s 0:00.00 0.0% 0+0k 0+0io 0pf+0w
%
Convergence Acceleration by Narrowing

Intuition for Convergence Acceleration with Narrowing

- Because the upward iteration sequence with widening converges to a post-fixpoint \( \hat{A} \) of \( f \) such that \( \text{lfp} f \subseteq \hat{A} \land f(\hat{A}) \subseteq \hat{A} \), we have, by recurrence and since \( f \) is increasing, that \( \text{lfp} f \subseteq f^n(\hat{A}) \subseteq \hat{A} \).
- When \( \hat{A} \) is not a fixpoint of \( f \), any iterate in the sequence \( Y^0 = \hat{A}, \ldots, Y^{n+1} = f(Y^n) = f^n(\hat{A}) \) is an overapproximation of the unknown \( \text{lfp} f \) more precise than \( \hat{A} \).

Intuition for Convergence Acceleration with Narrowing

- Because the upward iteration sequence with widening converges to a post-fixpoint \( \hat{A} \) of \( f \) such that \( \text{lfp} f \subseteq \hat{A} \land f(\hat{A}) \subseteq \hat{A} \), we have, by recurrence and since \( f \) is increasing, that \( \text{lfp} f \subseteq f^n(\hat{A}) \subseteq \hat{A} \).
- When \( \hat{A} \) is not a fixpoint of \( f \), any iterate in the sequence \( Y^0 = \hat{A}, \ldots, Y^{n+1} = f(Y^n) = f^n(\hat{A}) \) is an overapproximation of the unknown \( \text{lfp} f \) more precise than \( \hat{A} \).
- However, this downward iteration \( \langle Y^n, n \in \mathbb{N} \rangle \) might be infinite or converging slowly.
Intuition for Convergence Acceleration with Narrowing

- Because the upward iteration sequence with widening converges to a post-fixpoint $\hat{A}$ of $f$ such that $\text{lfp } f \subseteq \hat{A} \land f(\hat{A}) \subseteq \hat{A}$, we have, by recurrence and since $f$ is increasing, that $\text{lfp } f \subseteq f^n(\hat{A}) \subseteq \hat{A}$.
- When $\hat{A}$ is not a fixpoint of $f$, any iterate in the sequence $Y^0 = \hat{A}, \ldots, Y^{n+1} = f(Y^n) = f^n(\hat{A})$ is an overapproximation of the unknown $\text{lfp } f$ more precise than $\hat{A}$.
- However, this downward iteration $(Y^n, n \in \mathbb{N})$ might be infinite or converging slowly.
- It is therefore necessary to ensure its fast convergence. Convergence acceleration means that $Y^{n+1}$ will be a function of $Y^n$ and $f(Y^n)$ and so $Y^{n+1} = Y^n \Delta f(Y^n)$ where $\Delta$ is called a narrowing.$^{11}$

---

Convergence

- For convergence, the narrowing must ensure termination with a fixpoint.

---

Soundness

- For soundness, the narrowing must perform over-approximations, that is $\text{y} \subseteq x \Delta y$, so as to stay above the unknown least fixpoint, which requires remaining above any fixpoint (which we have no way to distinguish from the least one)$^{12}$.

---

Example: Interval Narrowing

For example, a narrowing for intervals could be

\[
\begin{align*}
\emptyset \Delta y & \triangleq \emptyset \\
x \Delta \emptyset & \triangleq \emptyset \\
[a, b] \Delta [c, d] & \triangleq \left( \{ a = -\infty \ ? c = a \}, \{ b = +\infty \ ? d = b \}\right)
\end{align*}
\]

Recall that in $x \Delta y$ the $x$ is an iterate and $y$ is the next iterate $f(x)$. So $[a, b] \Delta [c, d]$ will just eliminate the infinite bounds in $[a, b]$ and replace them by the bounds of the next iterate $[c, d]$.

So the narrowed interval is larger than $[c, d]$ that is $f(x)$ which ensures that we perform an overapproximation. Because only finitely many bounds can be infinite hence potentially removed, termination is guaranteed.

---

\[\text{\textcopyright P. Cousot}\]
Example: Interval Narrowing (Cont’d)

A functional encoding of the narrowing in OCAML could be:

```ocaml
let narrow x y = match x, y with
  | EMPTY, _ -> EMPTY
  | _, EMPTY -> EMPTY
  | INT (a, b), INT (c, d) ->
    let a' = if a = min_int then c else a in
    let b' = if b = max_int then d else b in
    INT (a', b');
```

Invariant Narrowing

\[ P \triangleq 1 \text{x := 1}; \text{while true do } 3 \text{x := (x + 1); od} 4. \]

In our example, the narrowing is applied once around the loop at program point
2, like the widening.

Abstract Interpreter with Widening/Narrowing

The abstract interpreter now calls the iterator using the invariant widening until reaching a postfixpoint and then calls the iterator using the invariant narrowing until reaching a fixpoint.

```ocaml
let analyzer() =
  let fw x = pwidth x (f x) in
  let w = (lfp pless pbot fw) in
  let fn x = pnarrow x (f x) in
  pprint (lfp pgreater w fn);
  analyzer();
```
Example of convergence acceleration by widening/narrowing

The result is now almost instantaneous.

```
% ocamlc interval.ml intervalWidening.ml intervalNarrowing.ml 
% invariant.ml invariantWidening.ml invariantNarrowing.ml 
% transformerBounded.ml iterator.ml 
% reachability_narrowing_bounded.ml 
% time ./a.out
1:(-1073741824,1073741823) 2:(1,101) 3:(1,100) 4:(101,101)
0.000u 0.000s 0:00.00 0.0% 0+0k 0+0io 0pf+0w
%
```

**Details of the iteration with Narrowing/Widening**

When compared to the Jacobi iterations, the chaotic iterates with widening and narrowing are extremely fast as shown below.

```
% ocamlc interval.ml intervalWidening.ml intervalNarrowing.ml 
% invariant.ml invariantWidening.ml invariantNarrowing.ml 
% transformerBounded.ml iteratorTrace.ml 
% reachability_narrowing_bounded_trace.ml 
% time ./a.out
1:_-1073741824 3:_101 4:_10100 2:_10101
1:(-1073741824,1073741823) 2:(1,101) 3:(1,100) 4:(101,101)
converged to lfp.
1:(-1073741824,1073741823) 2:(1,101) 3:(1,100) 4:(101,1073741823)
converged to lfp.
1:(-1073741824,1073741823) 2:(1,101) 3:(1,100) 4:(101,101)
0.000u 0.000s 0:00.00 0.0% 0+0k 0+0io 0pf+0w
%
```

**On the (im)precision of the analysis...**

Of course the narrowing cannot always recover all information lost by the widening, in particular because it is blocked by fixpoints jumped over by the widening.

```
let analyzer () =
  analyzer ();;
```

**Widening/Narrowing are not duals**

So we need four different notations, as follows.

<table>
<thead>
<tr>
<th></th>
<th>Iteration starts from</th>
<th>Iteration stabilizes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Widening ▽</td>
<td>below</td>
<td>above</td>
</tr>
<tr>
<td>Narrowing △</td>
<td>above</td>
<td>above</td>
</tr>
<tr>
<td>Dual widening △</td>
<td>above</td>
<td>below</td>
</tr>
<tr>
<td>Dual narrowing △</td>
<td>below</td>
<td>below</td>
</tr>
</tbody>
</table>

No dual widening △ has ever been found but trivial ones such as bounded execution (bounded model-checking), execution on a few cases (debugging), etc.
On actual abstract interpreters

Remark 3.1 For simplicity, we have designed a specific abstract interpreter for a specific program.
- In practice, abstract interpreters are parameterized by the program they have to analyze, and by the abstraction which should be used for the analysis.

Chaotic Iterations: A Structural Instance

Chaotic iterations

the iteration of the abstract equations need not follow the Jacobi iteration strategy and can be done in any chaotic order provided no equation is forgotten forever (or equivalently every equation is evaluated infinitely often) until it is stabilized.

A particular instance of such an efficient chaotic iteration follows program execution as defined by induction on its syntax

Starting from the entry condition at program point \(^1\), we can stabilize the loop \(^2-^3\) before computing the invariant at program point \(^4\).
Structural iterations

(* structural reachability analysis with widening and
  narrowing *)

open Interval
open IntervalWidening
open IntervalNarrowing
open Invariant
open TransformerBounded
open Iterator

let analyzer () =
  let p1 = f1 () in
  let p2 = let f x2 = f2 p1 (f3 x2) in
    let fw x2 = widen x2 (f x2) in
    let w = (lfp less EMPTY fw) in
    let fn x2 = narrow x2 (f x2) in
    (lfp greater w fn) in
  let p3 = f3 p2 in
  let p4 = f4 p2 in
  pprint (p1, p2, p3, p4);
  analyzer ();;

Structural iterations (cont’d)

and get exactly the same global result (the trace shows the iteration with
widening and then the iteration with narrowing for the loop 2—3)

Verifer

- The abstract interpreter that we have designed is a sound static analyzer.
  Given a program it produces interval information always valid at runtime.
- We can turn it into a verifier checking an interval specification.
Verifier

- The abstract interpreter that we have designed is a sound static analyzer. Given a program it produces interval information always valid at runtime.
- We can turn it into a verifier checking an interval specification.
- The specification can be provided by the user or remain implicit (e.g. absence of runtime errors such as overflows).
- One kind of user specification is a type declaration, for example an interval declaration for integer variables like $\text{var } x : 1..100$;
- Let us understand this declaration as: “only values between 1 and 100 can be assigned to $x$, otherwise execution stops” (with a runtime error).
- Observe that this does not mean that $x$ always has a value between 1 and 100 because it can be initialized with any integer value.\(^{13}\)

---

Example of Interval Verification

For the following example
\[
P' \triangleq \text{var } x : 1..100 \; ; \; ^1 x := 1 \; ; \; \text{while } ^2 (x <= 100) \; \text{do } ^3 x := (x + 1) \; ; \; \text{od}^4.
\]
the abstract interval equations become
\[
\begin{align*}
X_1 &= \{ x \leftarrow \text{min$_\text{int}$, max$_\text{int}$}\} \\
X_2 &= \{ x \leftarrow ([1, 1] \cup \{(X_3(x) = \emptyset ? \emptyset : \text{let } [a, b] = X_3(x) \in \text{min}(a + 1, \text{max$_\text{int}$}), \text{min}(b + 1, \text{max$_\text{int}$})]) \cap [1, 100])
\\
X_3 &= X_2 \cap \{ x \leftarrow \text{min$_\text{int}$, 100}\} \\
X_4 &= X_2 \cap \{ x \leftarrow [101, \text{max$_\text{int}$}]\}
\end{align*}
\]
since execution stops if and when a value outside $[1, 100]$ is going to be assigned to $x$. The result of the analysis is now the following. This declaration

---

Encoding the Declaration

This declaration is encoded in OCaml as follows

```ocaml
(* declaration.ml *)
open Interval
open Invariant
let d =
  (INT (min_int, max_int),
   INT (1,100),
   INT (min_int, max_int),
   INT (min_int, max_int));;
```

Encoding the Verifier

The abstract interpreter performs the iterative abstract reachability fixpoint
overapproximation with widening/narrowing and intersection with the decla-
ration, then prints the least fixpoint result, and finally checks for errors.

```ocaml
(* reachability verification with widening and narrowing *)
open Invariant
open InvariantWidening
open InvariantNarrowing
open TransformerBounded
open Iterator
open Declaration
open Verifier
let verifier () =
  let fw x = (pmeet (pwiden x (f x)) d) in
  let w = (lfp pless pbott fw) in
  let fn x = pnarrow x (f x) in
  let a = (lfp pgreater w fn) in
  pprint a; pverify cless f a d;;
verifier ();;
```

Encoding the Verification Phase

The verification of absence of errors checks that at any point during an
execution without error up to some point in the computation will not have an
error at the next execution step.

```ocaml
(* verifier.ml, interval invariant abstract domain *)
let pwarning (b1, b2, b3, b4) =
  let m = "Potential error at line " in
  if not b1 then print_string (m+"1\n");
  if not b2 then print_string (m+"2\n");
  if not b3 then print_string (m+"3\n");
  if not b4 then print_string (m+"4\n");
  let pverify leq f a d =
    let b = leq (f a) d in
    pwarning b;
```

Result of the Analysis

```ocaml
% ocamlc interval.ml intervalWidening.ml intervalNarrowing.ml \
? invariant.ml invariantWidening.ml invariantNarrowing.ml \
? transformerBounded.ml iterator.ml declaration.ml \
? verifier.ml reachability_narrowing_declaration.ml \
% time ./a.out
1:(-1073741824,1073741823) 2:(1,100) 3:(1,100) 4:0
Potential error at line 2
0.000u 0.000s 0:00.00 0.0% O+0k O+0io Opf+0w
%
```

- Observe that the program execution always stops at program point 3 with
  an overflow outside the range [1, 100] so program point 4 is now unreachable
  (with an overapproximation we can prove the presence of dead code but not
  its absence).
- Notice that the error is signaled as potential (with an overapproximation
  we can prove the values to definitely be within given bounds but not to prove
  that execution ever assigns a given value to a variable). Here is a trace of the
  analysis.
Details of the Analysis
%
% ocamlc interval.ml intervalWidening.ml intervalNarrowing.ml \
? invariant.ml invariantWidening.ml invariantNarrowing.ml \
? transformerBounded.ml iteratorTrace.ml declaration.ml \
? verifier.ml reachability_narrowing_declaration_trace.ml 
% time ./a.out
1:1 2:1 3:1 4:1
1:(-1073741824,1073741823) 2:(1,1) 3:1 4:1
1:(-1073741824,1073741823) 2:(1,1) 3:(1,1) 4:1
1:(-1073741824,1073741823) 2:(1,100) 3:(1,1) 4:1
1:(-1073741824,1073741823) 2:(1,100) 3:(1,100) 4:1
converged to lfp.
1:(-1073741824,1073741823) 2:(1,100) 3:(1,100) 4:1
Potential error at line 2
0.000u 0.000s 0:00.00 0.0% 0+0k 0+0io 0pf+0w
%
When to do the verification?
Notice that in general the verification cannot be done during the analysis since a widening may cause an overapproximation potentially raising a potential error while the narrowing may refine the analysis well enough to that this potential error disappears.

Correcting the Declaration
(* declarationCorrect.ml *)
open Interval
open Invariant
let d =
  (INT (min_int,max_int),
  INT (1,101),
  INT (min_int,max_int),
  INT (min_int,max_int));;
yields no error, the verification is completed.
%
% ocamlc interval.ml intervalWidening.ml intervalNarrowing.ml \
? invariant.ml invariantWidening.ml invariantNarrowing.ml \
? transformerBounded.ml iterator.ml declarationCorrect.ml \
? verifier.ml reachability_narrowing_declaration_correct.ml 
% time ./a.out
1:(-1073741824,1073741823) 2:(1,101) 3:(1,100) 4:(101,101)
0.000u 0.000s 0:00.00 0.0% 0+0k 0+0io 0pf+0w
%
A Touch of Abstract Interpretation Theory


Patrick Cousot & Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Emptiness. POPL 1977: 238-262


Patrick Cousot. Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique des programmes. Thèse És Sciences Mathématiques, Université Joseph Fourier, Grenoble, France, 21 March 1978

Abstract Interpreters

- **Transitional abstract interpreters**: proceed by induction on program steps
- **Structural abstract interpreters**: proceed by induction on the program syntax
- **Common main problem**: over/under-approximate fixpoints in non-Noetherian\(^(*)\) abstract domains \(^(**)\)

\(^(*)\) Iterative fixpoint computations may not converge in finitely many steps
\(^(**)\) Or convergence may be guaranteed but to slow.

Fixpoints

- **Poset** (or pre-order) \(\langle D, \sqsubseteq, \bot, \top \rangle\)
- **Transformer**: \(F \in D \mapsto D\)
- **Least fixpoint**: \(\text{lfp}^\subseteq F = \bigsqcup_{n \in \mathbb{N}} F^n(\bot)\) (under appropriate hypotheses)

\[ F \sqsubseteq F(X) \]
\[ X \sqsubseteq F(X) \]
\[ F(X) \subseteq X \]

\[ F^n(\bot) \]
\[ F^2(\bot) \]
\[ F^1(\bot) \]

\[ \text{lfp}^\subseteq F = \bigsqcup_{n \in \mathbb{N}} F^n(\bot) \]

\[ F(X) = X \]

Fixpoint Iteration

Convergence Acceleration by Extrapolation and Interpolation

Patrick Cousot, Radhia Cousot
Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. POPL 1977: 238-252

Patrick Cousot, Radhia Cousot
Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation. PLILP 1992: 260-295

Patrick Cousot: Abstracting Induction by Extrapolation and Interpolation. VMCAI 2015: 19-42

Convergence acceleration with widening

\[ \text{lfp} F \]

Infinite iteration
Convergence acceleration with widening

- **Infinite iteration**
  - Accelerated iteration with widening (e.g. with a widening based on the derivative as in Newton-Raphson method)

Extrapolation by Widening

- $X^0 = \perp$ (increasing iterates with widening)

- $X^\ell = X^n \land F(X^n)$ when $F(X^n) \not\subseteq X^n$

- $X^{\ell+1} = X^n$ when $F(X^n) \subseteq X^n$

- Widening $\nabla$:
  - $Y \subseteq X \nabla Y$ (extrapolation)
  - Enforces convergence of increasing iterates with widening (to a limit $X^\ell$)

The oldest widenings

- **Primitive widening** [1,2]

- Widening with thresholds [3]

Extrapolation with widening

Interpolation with narrowing

- \( Y^0 = X^c \) (decreasing iterates with narrowing)
  \[ Y^{n+1} = Y^n \triangleright F(Y^n) \] when \( F(Y^n) \sqsubseteq Y^n \)
  \[ Y^{n+1} = Y^n \] when \( F(Y^n) = Y^n \)

- Narrowing \( \triangleright \):
  
  - \( Y \subseteq X \implies Y \triangleleft X \triangleleft Y \subseteq X \) (interpolation)

- Enforces convergence of decreasing iterates with narrowing (to a limit \( Y^k \))

**The oldest narrowing**

- [2]

\[ [a_1, b_1] \preceq [a_2, b_2] = \]
\[ \begin{align*}
&\text{if } a_1 = -\infty \text{ then } a_2 \text{ else MIN } (a_1, a_2), \\
&\text{if } b_1 = -\infty \text{ then } b_2 \text{ else MAX } (b_1, b_2)
\end{align*} \]

**Duality**

<table>
<thead>
<tr>
<th></th>
<th>Convergence above the limit</th>
<th>Convergence below the limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Increasing iteration</td>
<td>Widening ( \triangleright )</td>
<td>Dual-narrowing ( \overset{\wedge}{\triangleright} )</td>
</tr>
<tr>
<td>Decreasing iteration</td>
<td>Narrowing ( \triangleright )</td>
<td>Dual widening ( \triangleright )</td>
</tr>
</tbody>
</table>

**Extrapolators**

**Interpolators**


--

Could stop when \( F(X) \not\sqsubseteq X \land F(F(X)) \not\sqsubseteq F(X) \) but not the current practice.
Extrapolators, Interpolators, and Duals

Interpolation with dual narrowing

- \( Z^0 = \bot \) (increasing iterates with dual-narrowing)
  
  \[
  Z^{n+1} = F(Z^n) \tilde{\Delta} Y^k \quad \text{when} \quad F(Z^n) \not\subseteq Z^n
  
  Z^{n+1} = Z^n \quad \text{when} \quad F(Z^n) \subseteq Z^n
  
  \]

- Dual-narrowing \( \tilde{\Delta} \):
  
  \[
  X \subseteq Y \implies X \subseteq X \tilde{\Delta} Y \subseteq Y
  
  \]

- Enforces convergence of increasing iterates with dual-narrowing

Example of dual-narrowing

- \([a, b] \tilde{\Delta} [c, d] \triangleq \lbrack c = -\infty \iff a \in ([a + c]/2)\rbrack, [d = \infty \iff b \in ([b + d]/2)\rbrack\]

- The first method we tried in the late 70’s with Radhia

  - Slow
  - Does not easily generalize (e.g. to polyhedra)
Relationship between narrowing and dual-

- $\Delta = \Delta^{-1}$
- $Y \subseteq X \implies Y \subseteq X \Delta Y \subseteq X$ (narrowing)
- $Y \subseteq X \implies Y \subseteq Y \Delta X \subseteq X$ (dual-narrowing)
- Example: Craig interpolation
- Why not use a bounded widening (bounded by $B$)?
  - $F(X) \subseteq B \implies F(X) \subseteq F(X) \Delta B \subseteq B$ (dual-narrowing)

Example of widenings (cont’d)

- Bounded widening (in $[\ell, h]$):
  
  \[
  [a,b] \nabla_{[\ell,h]} [c,d] \triangleq \frac{[c+2\ell, b+d+2h]}{2}
  \]

Widenings are not increasing

- A well-known fact
  
  $[1,1] \subseteq [1,2]$ but $[1,1] \nabla [1,2] = [1,\infty] \nsubseteq [1,2] \nabla [1,2] = [1,2]$

- A widening cannot both:
  - Be increasing in its first parameter
  - Enforce termination of the iterates
  - Avoid useless over-approximations as soon as a solution is found($^\dagger$)

($^\dagger$) A counter-example is $x \nabla y = \top$
Soundness

• The fixpoint approximation soundness theorems can be expressed with minimalist hypotheses [1]:
  • No need for complete lattices, complete partial orders (CPO’s):
    • The concrete domain is a poset
    • The abstract domain is a pre-order
    • The concretization is defined for the abstract iterates only.


Soundness (cont’d)

• No need for increasingness/monotony hypotheses for fixpoint theorems (Tarski, Kleene, etc)
• The concrete transformer is increasing and the limit of the iterations does exist in the concrete domain
• No hypotheses on the abstract transformer (no need for fixpoints in the abstract)
• Soundness hypotheses on the extrapolators/interpolators with respect to the concrete
• In addition, termination hypotheses on the extrapolators/interpolators ensure convergence in finitely many steps
Examples of interpolators

Craig interpolation
• Craig interpolation:
  Given $P \implies Q$ find $I$ such that $P \implies I \implies Q$ with
  $\text{var}(I) \subseteq \text{var}(P) \cap \text{var}(Q)$

  is a dual narrowing (already observed by Vijay D’Silva and Leopold Haller as an inverted narrowing)

• This evidence looked very controversial to some reviewers

• The generalization of an idea does not diminish in any way the merits and originality of this idea
Conclusion

- The presentation relied purely on intuition, can be made formal (see references)
- The abstraction ideas can scale up with enough precision, e.g.
  - ASTRÉE:
    - http://www.astree.ens.fr/
    - http://www.absint.de/astree/
  - CCCheck: code contract Static checker
  - MSR, Redmond (try online), public domain: https://github.com/Microsoft/CodeContracts

Bibliography

Patrick Cousot, Radhia Cousot:
A gentle introduction to formal verification of computer systems by abstract interpretation.
Logics and Languages for Reliability and Security 2010: 1-29

An online introduction (in French): http://www.di.ens.fr/~cousot/COUSOTtalks/CollegeDeFrance08.shtml

An online course: http://web.mit.edu/afs/athena.mit.edu/course/16/16.399/www/
Introductions

- http://www.di.ens.fr/~cousot/COUSOTpapers/TSI00.shtml (in french)
- http://www.di.ens.fr/~cousot/COUSOTpapers/Marktoberdorf98.shtml

References


