Verification of programs by abstract interpretation

Marc Chevalier

May 28, 2019

What is abstract interpretation?

A way to prove properties on programs

- No undefined behavior
- Some specification on output is matched
- Maximum execution time, constant execution path
- ... any other semantic property you can think of.

Why? – What happen when software fail

Figure 1: Ariane V, 4th June 1996

Why? – What happen when software fail

Figure 2: Ariane V, 40s later

Why? – Cost of software failure

Bugs have various annoying consequences:

- Deaths (Patriot MIM-104, Toyota, radiotherapy machines)
- A lot of money: Ariane V (payload: $\$370 \cdot 10^6$), $\$60 \cdot 10^9$/year in the US (NIST)
- Privacy (Heartbleed)
- ...

Why? – What we usually do

How developers think they can avoid bugs:

- Tests
- High level/safe language
- Strict code style

Still, Ariane V crashed. . . . "And here, poor fool[s], with all [their] lore, [they] stand no wiser than before".

The idea

- Check an execution: test, limited.
- Check all executions at once: ok, but not computable.
- Compute an over-approximation of all executions: sound, not complete.

Every possible behavior will be in our approximation (but maybe more).

An example

```c
int f(int x)
{                        // x ∈ [−2^31; 2^31 − 1]
    y = abs(x);          // y ∈ [0; 2^31 − 1] ∨ x = −2^31
    z = y + 1;           // z ∈ [1; 2^31 − 1] ∨ y = 2^31 − 1
    return 1/z;          // 0 ∉ [1; 2^31 − 1] ⇒ OK !
}
```

Let's generalize

$(D, \subseteq)$ a too big set (with good properties): typically, set of memory environments.

$$[\![P]\!] = f_1 \circ \cdots \circ f_n$$

We want that $c \subseteq$ *specification* holds at every program point.

Let's generalize

Abstract domain:

- $(D^\sharp, \subseteq^\sharp)$: a reasonable set (eg. $\overline{\mathbb{Z}}^2$)
- $\gamma : D^\sharp \to D$: concretization (eg. $(a, b) \mapsto \{x \in \mathbb{Z} \mid a \leqslant x \leqslant b\}$)

Sound if for all program point, $c \subseteq \gamma(a)$: we don't miss any behavior by executing in the abstract (but we lose precision).

Sound abstract operator: $f_i \circ \gamma \subseteq \gamma \circ f_i^\sharp$.

And we want $\gamma(a) \subseteq \mathit{specification}$.

Let's generalize

$$[\![\times 2]\!] \circ \gamma$$

$$\begin{array}{ccc} [-1;1] & & \\ \gamma \downarrow & & \\ \{-1,0,1\} & \xrightarrow{\times 2} & \{-2,0,2\} \end{array}$$

$$\gamma \circ [\![\times 2]\!]^\sharp$$

$$\begin{array}{ccc} [-1;1] & \xrightarrow{(\times 2)^\sharp} & [-2;2] \\ & & \downarrow \gamma \\ & & \{-2,-1,0,1,2\} \end{array}$$

We have every possible result by executing in the abstract.

In everyday life

Every way to infer some property about a system without knowing everything:

- Rule of signs for multiplication:

×	+	-
+	+	-
-	-	+

- Vote counting

The incompleteness

```
/*@ requires -10 <= x <= 10; */
int g(int x)
{                           // x ∈ [−10, 10]
    int y = x;              // y ∈ [−10, 10]
    int z = x * y;
    /* z ∈ Interval({a × b | a ∈ [−10, 10], b ∈ [−10, 10]})
    z ∈ [−100, 100]
    */
    int t = z + 1;          // t ∈ [−99, 101]
    return 1/t;             // 0 ∈ [−99, 101] ⇒ Alarm!
}
```

But this program is clearly safe.
What happens? This abstract domain cannot understand the relation between x and y.

In completeness in everyday life

Sometimes our partial knowledge is not enough:

- Rule of signs for addition:

+	+	-
+	+	?
-	?	-

- Vote counting without absolute majority

Other domains

- Numerical:
  Non relational:
  - Modulo: $x_i \equiv c_i[n_i]$
  - Bitwise: $x_i = 0?1??100010111????$
  - Sign: $x_i < 0,\ x_i > 0,\ x_i \leqslant 0 \dots$

  Relational:
  - Polytope: $\sum a_i x_i \leqslant c_i$
  - Octagon: $\pm x_i \pm x_j \leqslant c_i$

  And combination of domains.
- Memory: some value points to another, memory structures, separation logic. . . .
- Partitioning: $(x > 0 \Rightarrow ...) \wedge (x \leqslant 0 \Rightarrow ...)$
- All ad hoc domain you need.

The problem of loops – 1st iteration

```
int i = 0;              // i ∈ [0, 0]
while(i < 1000) {       // i ∈ [0, 0]
    i = i + 1;          // i ∈ [1, 1]
}
```

The problem of loops – 2^{nd} iteration

```
int i = 0;               // i ∈ [0,0]
while(i < 1000) {        // i ∈ [0,0] ∪♯ [1,1] = [0,1]
    i = i + 1;           // i ∈ [1,2]
}
```

The problem of loops – 3rd iteration

```
int i = 0;               // i ∈ [0,0]
while(i < 1000) {        // i ∈ [0,1] ∪♯ [1,2] = [0,2]
    i = i + 1;           // i ∈ [1,3]
}
```

The problem of loops – 1000$^{\text{th}}$ iteration

```
int i = 0;              // i ∈ [0, 0]
while(i < 1000) {       // i ∈ [0, 999]
    i = i + 1;          // i ∈ [1, 1000] = [1, 999] ∪♯ [1000, 1000]
}                       // i ∈ [1000, 1000]
```

Urgh! So long!

And what if a loop is really long? Or does not terminate?

The solution: the widening

Instead of using the abstract union ($\cup^\sharp$), we use a widening (∇).

- sound: $\forall (a, b) \in {D^\sharp}^2, \gamma(a) \cup \gamma(b) \subseteq \gamma(a \nabla b)$
- termination: for all $top_0 \in D^\sharp$ and $f^\sharp : D^\sharp \to D^\sharp$, the sequence

$$top_{n+1} = top_n \nabla f^\sharp(top_n)$$

is stationary.

Interval widening

Drop unstable constrains:

$$[a, b] \nabla [c, d] = \left[\begin{cases} a & \text{if } a \leqslant c \\ -\infty & \text{otherwise} \end{cases}, \begin{cases} b & \text{if } b \geqslant d \\ +\infty & \text{otherwise} \end{cases} \right]$$

Interval widening – 1st iteration

```
int i = 0;
while(i < 1000) { // i ∈ [0,0]
    i = i + 1;    // i ∈ [1,1]
}
```

Interval widening – 2nd iteration

```
int i = 0;
while(i < 1000) { // i ∈ [0, 0]∇[1, 1] = [0, +∞]
      i = i + 1;  // i ∈ [1, +∞] = [1, 999] ∪♯ [1000, +∞]
}                 // i ∈ [1000, +∞]
```

Discussion

We can add thresholds (e.g. constants ± 1). No widening at some iteration. . . .

Trade-off: convergence speed vs. precision.

We can still refine the invariant a posteriori.

What? Why?

Ok, but how?

Building an analyzer

Past and future

Abstract interpretation vs. the world

Good things:

- Works on existing code
- It really works: Astrée (A340, A380)
- Quite automatic (when you have the suited domains)
- The developer who knows its code can help the analyzer easily

Bad things:

- Incompleteness
- A lot of work if existing domains are not powerful enough
- Some properties are very difficult to prove with this method

My work

Study case: the OS of an host platform in planes at the border between trusted (flight control) and untrusted (potentially hostile) world.

We want to prove some security properties: memory isolation, hosted applications don't get more privileges. . . .

Properties are not visible from C (about CPU state, mainly): inline assembly $\Rightarrow$ analyze assembly. Impacts everything.