Abstract Interpretation: Achievements and Perspectives

Patrick COUSOT
École Normale Supérieure
45 rue d’Ulm, 75230 Paris cedex 05, France

SSGRR-2000, L’Aquila, Italy July 31st – August 6th, 2000

Motivations & Introduction

The initial application: program analysis

• Prove automatically that:
  for all programs $P$ of a given programming language $\mathcal{L}$:
  for all possible executions of that program $P$ in any conceivable environment:
  a given specification $S$ is always satisfied.

• Initially the considered specifications $S$ were simple safety specifications (such as absence of runtime errors).

The methodology [CC-POPL’77]

• Define formally the program executions by a fixpoint semantics of the programs of the language $\mathcal{L}$;

• Since the semantics of a program is not computable, use a manually designed approximation/abstraction of that semantics to check the specification.

Reference

Semantics: intuition

- The **semantics of a language** defines the semantics of any program written in this language;
- The **semantics of a program** provides a formal mathematical model of all possible behaviors of a computer system executing this program (interacting with any possible environment);
- Any **semantics** of a program can be defined as the solution of a fixpoint equation;
- All **semantics** of a program can be organized in a hierarchy by abstraction.

Example: trace semantics [4, 6]

![Diagram of trace semantics]

**Initial states**
- a
- b
- c
- d

**Intermediate states**
- e
- f
- g
- h

**Final states of the finite traces**
- i
- j

**Infinite traces**
- k
- l
- m
- n
- o
- p

**Discrete time**
0 1 2 3 4 5 6 7 8 9
Examples of computation traces

- **Finite** ($C+1=\cdots$):
- **Erroneous** ($C+1+1+1\cdots$):
- **Infinite** ($C+0+0+0\cdots$):

---

**Least Fixpoints**: intuition [4, 6]

\[
\text{Behaviors} = \{ \bullet \mid \bullet \text{ is a final state} \}
\cup \{ \bullet \cdots \bullet \mid \bullet \text{ is an elementary step} \& \bullet \cdots \bullet \in \text{Behaviors}^+ \}
\cup \{ \bullet \cdots \bullet \cdots \cdots \mid \bullet \text{ is an elementary step} \& \bullet \cdots \bullet \cdots \bullet \in \text{Behaviors}^\infty \}
\]

- In general, the equation has multiple solutions.
- Choose the least one for the partial ordering:

  \[ \text{« more finite traces \& less infinite traces »} \]

---

**Abstraction**: intuition

- Abstract interpretation is a theory of the approximation of the behavior of discrete dynamic systems, including the semantics of (programming or specification) languages [8, 9, 2];

- Abstract interpretation formalizes the intuitive idea that a semantics is more or less precise according to the considered observation level.
**Example 1 of abstraction**

![Diagram showing states and transitions](image)

**Example 2 of abstraction**

![Diagram showing states and transitions](image)

**Example 3 of abstraction**

![Diagram showing states and transitions](image)

**Computable abstractions**

- If the approximation is rough enough, the abstraction of a semantics can lead to a version which is less precise but is effectively computable by a computer;

- The computation of this abstract semantics amounts to the effective iterative resolution of fixpoint equations;

- By effective computation of the abstract semantics, the computer is able to analyze the behavior of programs and of software before and without executing them \[7\].

---

Computable abstractions of an [in]finite set of points;

Example 1: signs

\[
\{ \ldots, (19, 88), \ldots, (19, 99), \ldots \}
\]

Computable abstractions of an [in]finite set of points; Example 2: intervals \([7, 8]\)

\[
\begin{aligned}
& x \in [19, 88] \\
& y \in [19, 99]
\end{aligned}
\]

Computable abstractions of an [in]finite set of points; Example 3: octagons

\[
\begin{aligned}
& 1 \leq x \leq 9 \\
& x + y \leq 88 \\
& 1 \leq y \leq 9 \\
& x - y \leq 99
\end{aligned}
\]
Computable abstractions of an \([\text{in}]\)finite set of points; Example 4: polyhedra [13]

\[
\begin{align*}
19x + 88y & \leq 2000 \\
19x + 99y & \geq 0
\end{align*}
\]

Computable abstractions of an \([\text{in}]\)finite set of points; Example 5: simple congruences [15]

\[
\begin{align*}
x & = 19 \mod 88 \\
y & = 19 \mod 99
\end{align*}
\]

Computable abstractions of an \([\text{in}]\)finite set of points; Example 6: linear congruences [16]

\[
\begin{align*}
x + 9y & = 8 \mod 8 \\
x - 9y & = 9 \mod 9
\end{align*}
\]

Computable abstractions of an \([\text{in}]\)finite set of points; Example 7: trapezoidal linear congruences [18, 19]

\[
\begin{align*}
x + 9y & \in [0, 88] \mod 10 \\
x - 9y & \in [0, 99] \mod 11
\end{align*}
\]
Computable abstractions of symbolic structures

- Most structures manipulated by programs are symbolic structures such as control structures (call graphs), data structures (search trees), communication structures (distributed & mobile programs), etc;
- It is very difficult to find compact and expressive abstractions of such sets of objects (languages, automata, trees, graphs, etc.).

Example of abstractions of infinite sets of infinite trees

Binary Decision Graphs: [20]

Tree schemata: [22, 21]

Information loss

- All answers given by the abstract semantics are always correct with respect to the concrete semantics;
- Because of the information loss, not all questions can be definitely answered with the abstract semantics;
- The more concrete semantics can answer more questions;
- The more abstract semantics are more simple.

Example of information loss

- Is the operation $1/(x+1-y)$ well defined at run-time?
- Concrete semantics: yes

Note that $E$ is the equality relation.
Example of information loss

- Is the operation $\frac{1}{x+1-y}$ well defined at run-time?
- Abstract semantics 1: I don’t know

Program Static Analysis

Objective of program static analysis

- Programming bugs should be eradicated before they lead to disastrous catastrophes!
- Fully automatic bug detection is impossible (undecidability);
- Program static analysis uses abstract interpretation to derive, from a standard semantics, an approximate and computable semantics. This derivation is itself not (fully) mechanizable;
- It follows that the computer is able to analyze the behavior of software before and without executing it;
- This is essential for computer-based safety-critical systems (for example: planes, trains, launchers, nuclear plants, etc.).
Example: interval analysis (1975)

Program to be analyzed:

\[
\begin{align*}
1: & \quad x := 1; \\
2: & \quad \text{while } x < 10000 \text{ do} \\
3: & \quad \quad x := x + 1 \\
4: & \quad \text{od;}
\end{align*}
\]

Example: interval analysis (1975)

Equations (abstract interpretation of the semantics):

\[
\begin{align*}
\begin{cases}
X_1 &= [1, 1] \\
X_2 &= (X_1 \cup X_3) \cap [-\infty, 9999] \\
X_3 &= X_2 \oplus [1, 1] \\
X_4 &= (X_1 \cup X_3) \cap [10000, +\infty]
\end{cases}
\end{align*}
\]

The analyzer reads the program text and produces a representation of the above equations and then solve them iteratively. The equations are an abstraction of the trace semantics of the program. The formal derivation of the algorithm producing the constraints by abstract interpretation of the program trace semantics is (mainly) manual.

Example: interval analysis (1975)

Constraints (abstract interpretation of the semantics):

\[
\begin{align*}
\begin{cases}
X_1 &= [1, 1] \\
X_2 &= (X_1 \cup X_3) \cap [-\infty, 9999] \\
X_3 &= X_2 \oplus [1, 1] \\
X_4 &= (X_1 \cup X_3) \cap [10000, +\infty]
\end{cases}
\end{align*}
\]

The analyzer reads the program text and produces (a representation of) the above constraints and then solve them iteratively. The constraints are an abstraction of the trace semantics of the program. The formal derivation of the algorithm producing the constraints by abstract interpretation of the program trace semantics is (mainly) manual.

Example: interval analysis (1975)

Increasing chaotic iteration, initialization:

\[
\begin{align*}
\begin{cases}
X_1 &= [1, 1] \\
X_2 &= (X_1 \cup X_3) \cap [-\infty, 9999] \\
X_3 &= X_2 \oplus [1, 1] \\
X_4 &= (X_1 \cup X_3) \cap [10000, +\infty]
\end{cases}
\end{align*}
\]

The analyzer reads the program text and produces a representation of the above equations and then solve them iteratively. The equations are an abstraction of the trace semantics of the program. The formal derivation of the algorithm producing the equation by abstract interpretation of the program trace semantics is (mainly) manual.

\[\text{© P. Cousot & R. Cousot, ISOP 1975, POPL 77.}\]
Increasing chaotic iteration:

Example: interval analysis (1975)

- $x := 1$
- while $x < 10000$ do
- 
  - $x := x + 1$

Example: interval analysis (1975)

- $x := 1$
- while $x < 10000$ do
- 
  - $x := x + 1$

Example: interval analysis (1975)

- $x := 1$
- while $x < 10000$ do
- 
  - $x := x + 1$
Example: interval analysis (1975)
Increasing chaotic iteration: convergence?

\[
\begin{align*}
x & := 1; \\
1: & \text{ while } x < 10000 \text{ do } \\
2: & \quad x := x + 1 \\
3: & \quad \text{ od; } \\
4: & \end{align*}
\]

Example: interval analysis (1975)
Increasing chaotic iteration: convergence??

\[
\begin{align*}
x & := 1; \\
1: & \text{ while } x < 10000 \text{ do } \\
2: & \quad x := x + 1 \\
3: & \quad \text{ od; } \\
4: & \end{align*}
\]

Example: interval analysis (1975)
Increasing chaotic iteration: convergence???

\[
\begin{align*}
x & := 1; \\
1: & \text{ while } x < 10000 \text{ do } \\
2: & \quad x := x + 1 \\
3: & \quad \text{ od; } \\
4: & \end{align*}
\]
Example: interval analysis (1975)
Increasing chaotic iteration: convergence??????

\[
\begin{align*}
x &:= 1 ; \\
\text{while } x < 10000 & \text{ do} \\
\phantom{x} &:= x + 1 \\
\od; \\
\end{align*}
\]

Example: interval analysis (1975)
Increasing chaotic iteration: convergence???????

\[
\begin{align*}
x &:= 1 ; \\
\text{while } x < 10000 & \text{ do} \\
\phantom{x} &:= x + 1 \\
\od; \\
\end{align*}
\]

Example: interval analysis (1975)
Convergence speed-up by extrapolation:

\[
\begin{align*}
x &:= 1 ; \\
\text{while } x < 10000 & \text{ do} \\
\phantom{x} &:= x + 1 \\
\od; \\
\end{align*}
\]
Widening

Example: interval analysis (1975)

Decreasing chaotic iteration:

\[
\begin{align*}
X_1 &= [1,1] \\
X_2 &= (X_1 \cup X_3) \cap [-\infty, 9999] \\
X_3 &= X_2 \oplus [1,1] \\
X_4 &= (X_1 \cup X_3) \cap [10000, +\infty]
\end{align*}
\]

1:

while \( x < 10000 \) do

2:

\( x := x + 1 \)

3:

od;

4:

\]

Example: interval analysis (1975)

Decreasing chaotic iteration:

\[
\begin{align*}
X_1 &= [1,1] \\
X_2 &= (X_1 \cup X_3) \cap [-\infty, 9999] \\
X_3 &= X_2 \oplus [1,1] \\
X_4 &= (X_1 \cup X_3) \cap [10000, +\infty]
\end{align*}
\]

1:

while \( x < 10000 \) do

2:

\( x := x + 1 \)

3:

od;

4:

\]
Final solution:

\[
\begin{align*}
x & := 1; \\
1: & \quad \text{while } x < 10000 \text{ do} \\
2: & \quad x := x + 1 \\
3: & \quad \text{od}; \\
4: & \quad \text{Result of the interval analysis:}
\end{align*}
\]

\[
\begin{align*}
X_1 & = [1, 1] \\
X_2 & = (X_1 \cup X_3) \cap [-\infty, 9999] \\
X_3 & = X_2 \oplus [1, 1] \\
X_4 & = (X_1 \cup X_3) \cap [10000, +\infty]
\end{align*}
\]

Exploitation of the result of the interval analysis:

\[
\begin{align*}
x & := 1; \\
1: & \quad \text{while } x < 10000 \text{ do} \\
2: & \quad x := x + 1 \\
3: & \quad \text{od}; \\
4: & \quad \text{no overflow}
\end{align*}
\]

Some applications of static analysis by abstract interpretation

- data flow and set-based analysis for program optimization & transformation (including partial evaluation) \[9, 12\];
- type inference (including undecidable systems)/soft typing \[5\];
- abstract model-checking of infinite systems \[11, 12\];
- abstract debugging & testing \[2, 1\];
- probabilistic analysis \[24\];
- communication topology analysis for mobile/distributed code \[25\];
- ...
Some other recent applications of abstract interpretation

- **Fundamental applications:**
  - design of hierarchies of semantics \([10, 3, 6], \ldots\);

- **Practical applications:**
  - generation of heuristics for search problems in AI;
  - automatic differentiation of numerical programs;
  - security (analysis of cryptographic protocols \([23]\), mobile code \([14]\));
  - semantic tattooing/watermarking of software, \ldots;

Present-day and forthcoming research

A lot of fundamental research remains to be done:

- modularity,
- higher order functions & modules,
- floating point numbers,
- probabilistic analyses,
- liveness properties with fairness,
- \ldots;

Industrialization of Program Static Analysis

An impressive application (1996/97)

- Abstract interpretation has been used (including interval analysis) for the static analysis of the embedded ADA software of the Ariane 5 launcher \(^5\); \([17]\)
- Automatic detection of the definiteness, potentiality, impossi-
  bility or inaccessibility of run-time errors \(^6\);
- Automatic discovery of the 501 flight error;
- Success for the 502 & 503 flights and the ARD \(^7\).

---

\(^4\) Flight software (60,000 lines of Ada code) and Inertial Measurement Unit (30,000 lines of Ada code).

\(^5\) such as scalar and floating-point overflows, array index errors, divisions by zero and related arithmetic exceptions,

\(^6\) uninitialized variables, data races on shared data structures, etc.

\(^7\) Atmospheric Reentry Demonstrator: module coming back to earth.
Industrialization of static analysis by abstract interpretation

- Connected Components Corporation (U.S.A.), L. Harrison, 1993;
- AbsInt Angewandte Informatik GmbH (Germany), R. Wilhelm & C. Ferdinand, 1998;

Pointers to references

Starter:


On the web:

http://www.di.ens.fr/~cousot/

References


