Abstract Interpretation and Static Analysis

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

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

IFIP WG 10.4, 40th Meeting on Formal Methods, Stenungsund, Sweden, July 4–8, 2001

The Software Reliability Problem

- The evolution of hardware by a factor of 10^6 over the past 25 years has lead to the explosion of the program sizes;
- The scope of application of very large software is likely to widen rapidly in the next decade;
- These big programs will have to be modified and maintained during their lifetime (often over 20 years);
- The size and efficiency of the programming and maintenance teams in charge of their design and follow-up cannot grow up in similar proportions;

Introductory Motivations on Software Reliability

The Software Reliability Problem (Cont'd)

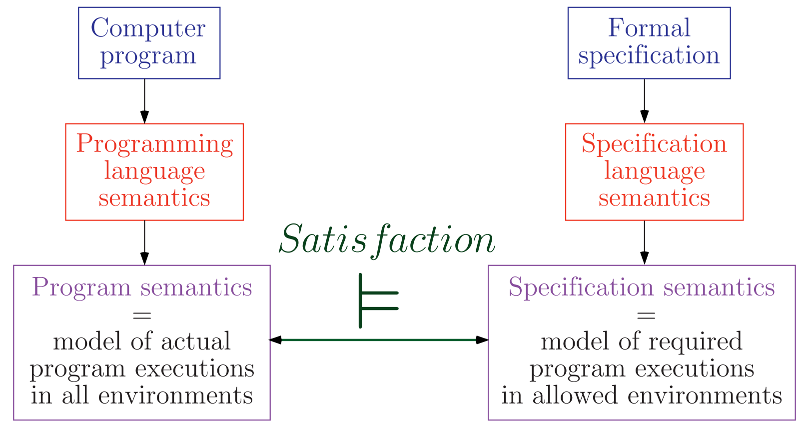
- At a not so uncommon (and often optimistic) rate of one bug per thousand lines such huge programs might rapidly become hardly manageable in particular for safety critical systems;
- Therefore in the next 10 years, the *software reliability problem* is likely to become a major concern and challenge to modern highly computer-dependent societies.

What Can We Do About It?

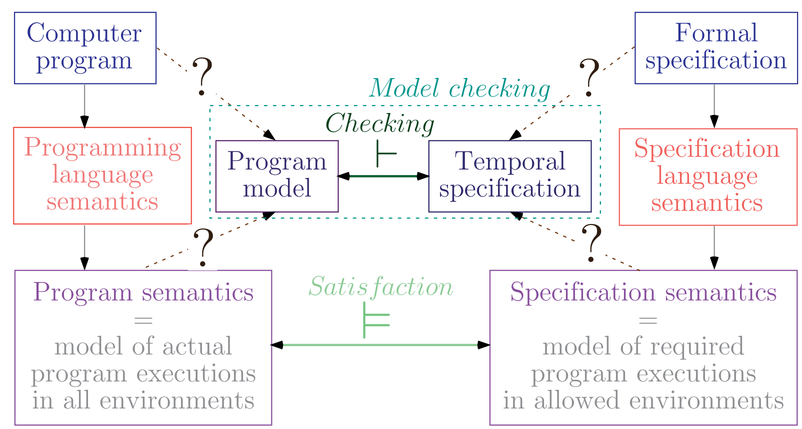
- Use our intelligence (thinking/intellectual tools: abstract interpretation);
- Use our computer (mechanical tools : static program analysis/checking/testing, the early idea of using computers to reason about computer).

Software Verification and Validation

The Verification/Validation Problem



Example: Model Checking



Other Examples of Software Verification/Validation Techniques

- Software testing;
- Simulation and prototyping;
- Technical reviews;
- Requirements tracing;
- Formal correctness proofs;
- Etc.

Practical Limitations

- Testing:
  - Testing all data on all paths is impossible;
- Formal methods:
  - No formal specification perfectly reflects informal human expectations;
  - Proofs grow exponentially in the size of programs/specifications which is incompatible with friendly user interaction and full automation;
- etc.

Fundamental Theoretical Limitations

- Undecidability: full automation of software verification/validation is impossible;
- Examples of undecidable questions:
  - Is my program bug-free? (i.e. correct with respect to a given specification);
  - Can a program variable take two different values during execution?

Undecidability and Approximation

- Since program verification is undecidable, computer aided program verification methods are all partial/incomplete;
- They all involve some form of approximation:
  - restricted specifications or programs (e.g. finiteness),
  - decidable questions or semi-algorithms,
  - practical time/memory complexity limitations,
  - require user interaction;
- Most of these approximations are formalized by **Abstract Interpretation**.

Examples of approximations

- Testing: coverage is partial (so errors are frequently found until the end of the software lifetime);
- Proofs: specifications are often partial, debugging proofs is often harder that testing programs (so only parts of very large software can be formally proved correct);
- Model checking: the model must fit machine limitations (so some facets of program execution must be left out) and be redesigned after program modifications;
- Typing: types are weak program properties (so type verification cannot be generalized to complex specifications).

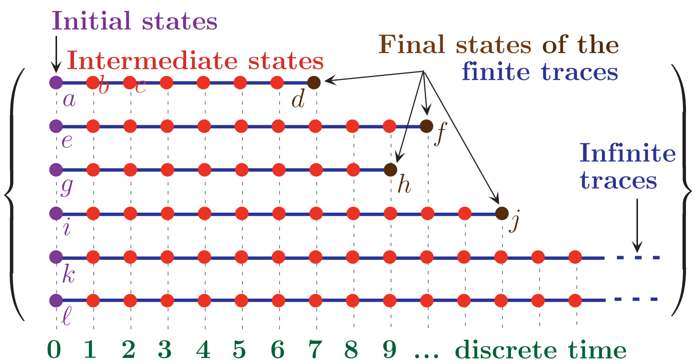
Content

Semantics

Semantics: intuition

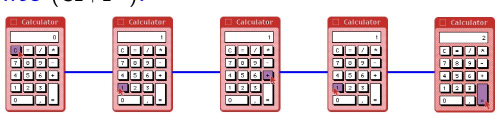
- The semantics of a language defines the semantics of any program written in this language;
- The semantics of a program provides a formal mathematical model of all possible behaviors of a computer system executing this program (interacting with any possible environment);
- Any semantics of a program 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

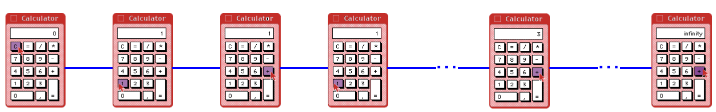


Examples of computation traces

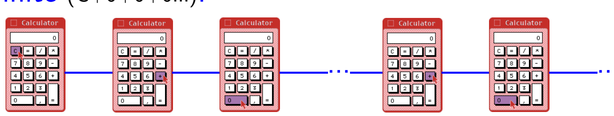
- Finite (C1+1=):



- Erroneous  (C1+1+1+1...):



- Infinite (C+0+0+0...):



Least Fixpoints: intuition

$$\begin{aligned}\textbf{Behaviors} = \ & \{\bullet \mid \bullet \text{ is a final state}\} \\ \cup \ & \{\bullet\!-\!\bullet\!-\!\ldots\!-\!\bullet \mid \bullet\!-\!\bullet \text{ is an elementary step } \& \\ & \quad \bullet\!-\!\ldots\!-\!\bullet \in \textbf{Behaviors}^{+}\} \\ \cup \ & \{\bullet\!-\!\bullet\!-\!\ldots\!-\!\ldots \mid \bullet\!-\!\bullet \text{ is an elementary step } \& \\ & \quad \bullet\!-\!\ldots\!-\!\ldots \in \textbf{Behaviors}^{\infty}\}\end{aligned}$$

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

« *more finite traces & less infinite traces* ».

Abstract Interpretation

Abstract Interpretation [1]

- Formalizes the idea of approximation of sets and set operations as considered in set (or category) theory;
- A theory of approximation of the semantics of programming languages;
- Main application: formal method for inferring general runtime properties of programs.

Reference

[1] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conf. Record of the 4th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages POPL'77*, Los Angeles, CA, 1977. ACM Press, pp. 238–252.

The Theory of Abstract Interpretation

- **Abstract interpretation** is a theory of conservative approximation of the semantics of computer systems.

  Approximation: observation of the behavior of a computer system at some level of abstraction, ignoring irrelevant details;

  Conservative: the approximation cannot lead to any erroneous conclusion.

Usefulness of Abstract Interpretation

- **Thinking tools**: the idea of abstraction is central to reasoning (in particular on computer systems);
- **Mechanical tools**: the idea of effective approximation leads to automatic semantics-based program manipulation tools.

Abstraction

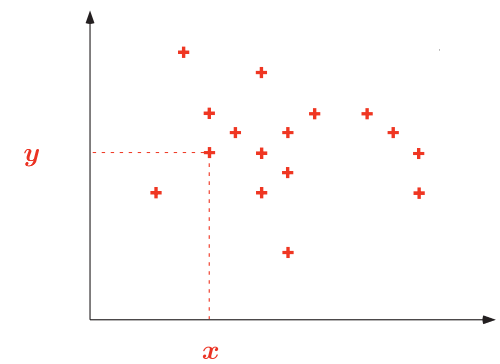
Abstraction: intuition

- Abstract interpretation formalizes the intuitive idea that a semantics is more or less precise according to the considered observation level of the program executions;

- Abstract interpretation theory formalizes this notion of approximation/abstraction in a mathematical setting which is independent of particular applications.

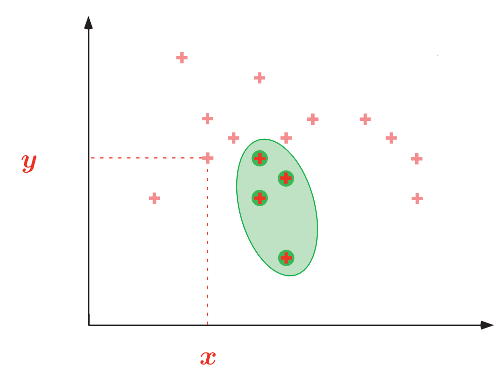
Intuition behind abstraction

Approximations of an [in]finite set of points;



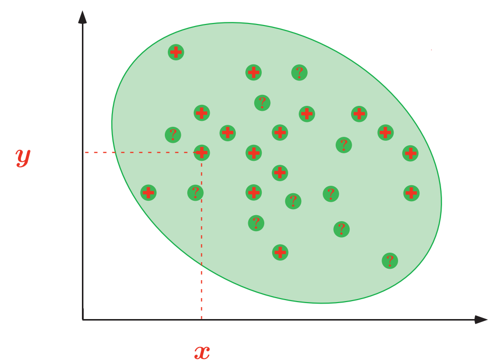
$\{\ldots, \langle 19,\ 78\rangle, \ldots,$
$\langle 20,\ 01\rangle, \ldots\}$

Approximations of an [in]finite set of points: From Below



$\{\ldots, \langle 19,\ 78\rangle, \ldots,$
$\ldots\}$

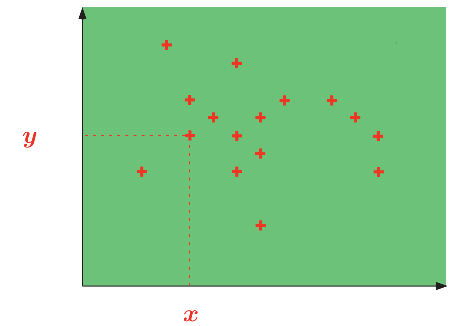
Approximations of an [in]finite set of points: From Above



$$\{\ldots, \langle 19,\ 78\rangle, \ldots,$$
$$\langle 20,\ 01\rangle, \langle ?,\ ?\rangle, \ldots\}$$

Intuition Behind Effective Computable Abstraction

Effective computable approximations of an [in]finite set of points; Signs [2]

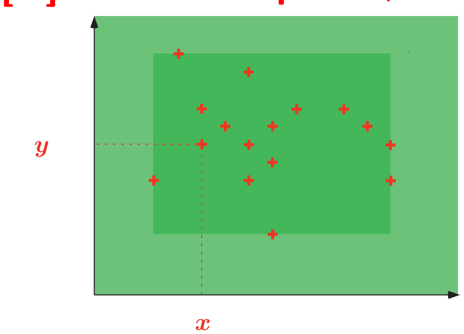


$$\begin{cases} x \geq 0 \\ y \geq 0 \end{cases}$$

Reference

[2] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *6th POPL*, pages 269–282, San Antonio, TX, 1979. ACM Press.

Effective computable approximations of an [in]finite set of points; Intervals [3]

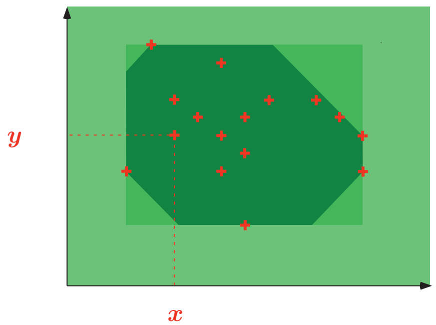


$$\begin{cases} x \in [19,\ 78] \\ y \in [20,\ 01] \end{cases}$$

Reference

[3] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *2nd Int. Symp. on Programming*, pages 106–130. Dunod, 1976.

Effective computable approximations of an [in]finite set of points; Octagons [4]

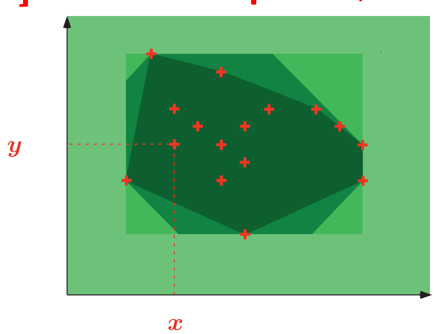


$$\begin{cases} 1 \le x \le 9 \\ x + y \le 78 \\ 1 \le y \le 9 \\ x - y \le 99 \end{cases}$$

Reference

[4] A. Miné. A New Numerical Abstract Domain Based on Difference-Bound Matrices. *Proc. 2nd Symp. on Programs as Data Objects PADO'2001*, 0. Danvy & A. Filinski (Eds.), LNCS 2053, Springer, 2001, pp. 155–172.

Effective computable approximations of an [in]finite set of points; Polyhedra [5]

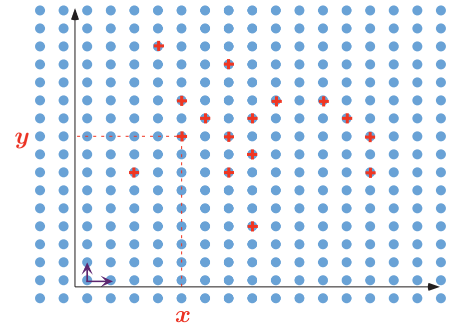


$$\begin{cases} 19x + 78y \le 2000 \\ 20x + 01y \ge 0 \end{cases}$$

Reference

[5] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *5th POPL*, pages 84–97, Tucson, AZ, 1978. ACM Press.

Effective computable approximations of an [in]finite set of points; Simple congruences [6]

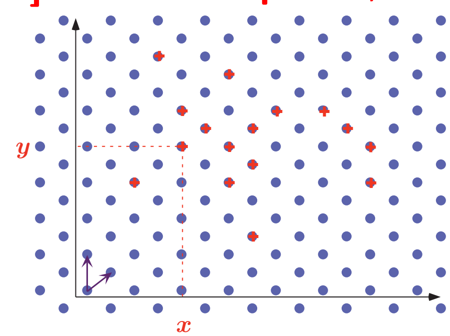


$$\begin{cases} x = 19 \bmod 78 \\ y = 20 \bmod 99 \end{cases}$$

Reference

[6] P. Granger. Static analysis of arithmetical congruences. *Int. J. Comput. Math.*, 30:165–190, 1989.

Effective computable approximations of an [in]finite set of points; Linear congruences [7]

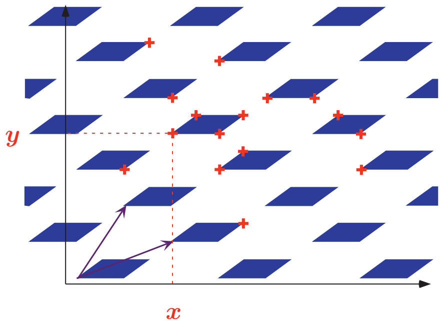


$$\begin{cases} 1x + 9y = 7 \bmod 8 \\ 2x - 1y = 9 \bmod 9 \end{cases}$$

Reference

[7] P. Granger. Static analysis of linear congruence equalities among variables of a program. *CAAP '91*, LNCS 493, pp. 169–192. Springer, 1991.

Effective computable approximations of an [in]finite set of points; Trapezoidal linear congruences [8]



$$\begin{cases} 1x + 9y \in [0, 78] \bmod 10 \\ 2x - 1y \in [0, 99] \bmod 11 \end{cases}$$

Reference

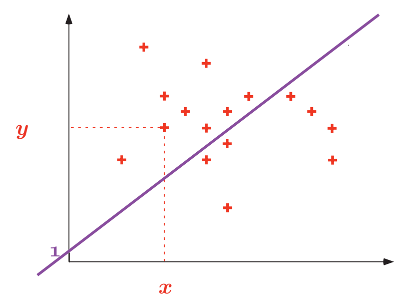
[8] F. Masdupuy. Array operations abstraction using semantic analysis of trapezoid congruences. In *ACM Int. Conf. on Supercomputing, ICS '92*, pages 226–235, 1992.

Conservative Approximation and Information Loss

Intuition Behind Sound/Conservative Approximation

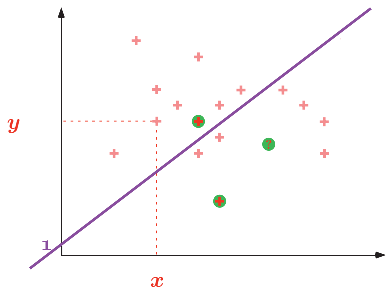
Conservative Approximation

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



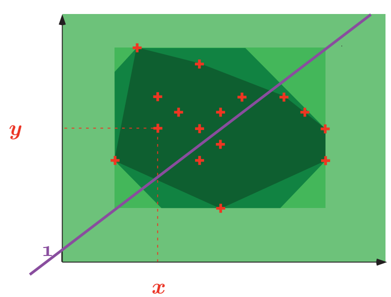
Conservative Approximation

- Is the operation `1/(x+1-y)` well defined at run-time?
- Testing : **You <u>never</u> know!**



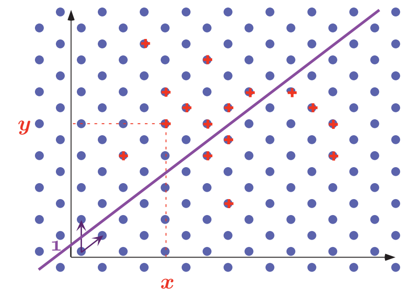
Conservative Approximation

- Is the operation `1/(x+1-y)` well defined at run-time?
- Abstract semantics 1: **I don't know**



Conservative Approximation

- Is the operation `1/(x+1-y)` well defined at run-time?
- Abstract semantics 2: **yes**



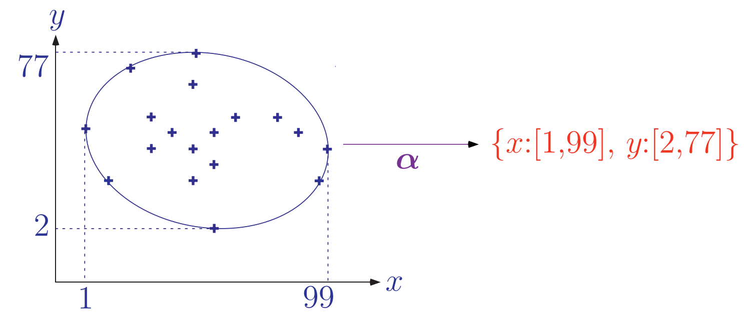
Intuition Behind Information Loss

Information Loss

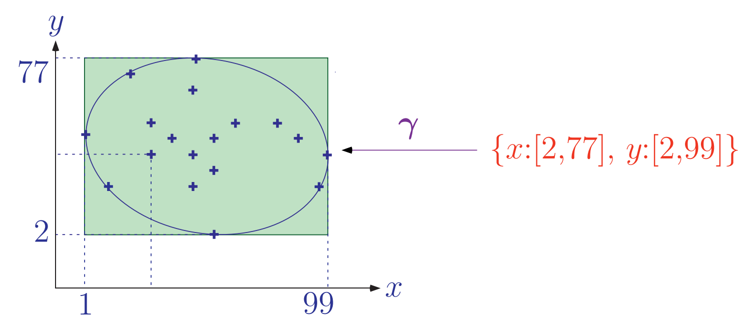
- All answers given by the abstract semantics are always correct with respect to the concrete semantics;
- Because of the information loss, not all questions can be definitely answered with the abstract semantics;
- The more concrete semantics can answer more questions;
- The more abstract semantics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

Basic Elements of Abstract Interpretation Theory

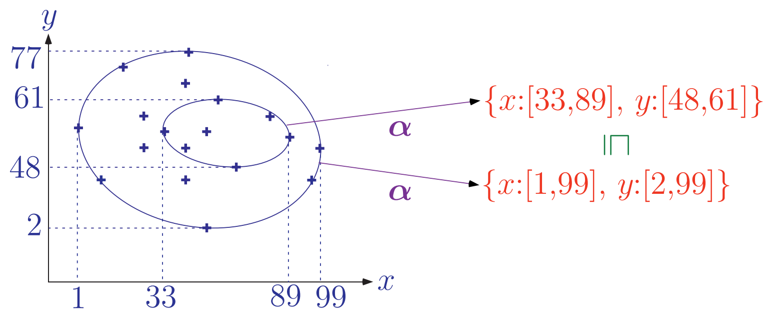
Abstraction α



Concretization γ

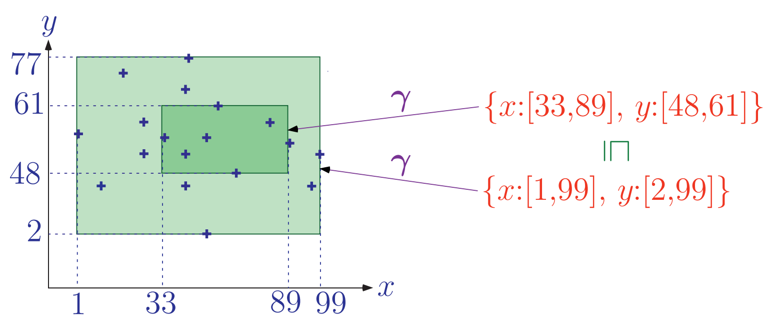


The Abstraction α is Monotone



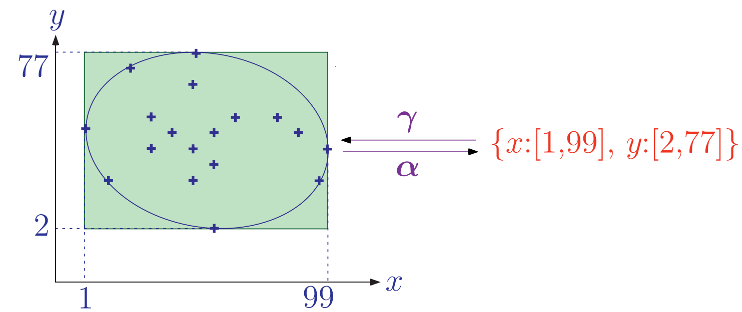
$$X \subseteq Y \Rightarrow \alpha(X) \sqsubseteq \alpha(Y)$$

The Concretization γ is Monotone



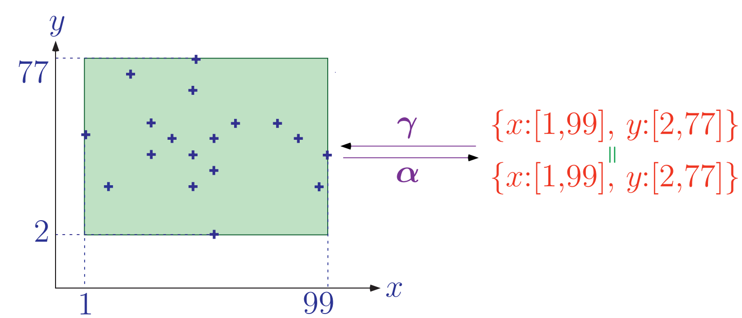
$$X \sqsubseteq Y \Rightarrow \gamma(X) \subseteq \gamma(Y)$$

The $\gamma \circ \alpha$ Composition



$$X \subseteq \gamma \circ \alpha(X)$$

The $\alpha \circ \gamma$ Composition



$$\alpha \circ \gamma(Y) = Y$$

Galois Connection [1]

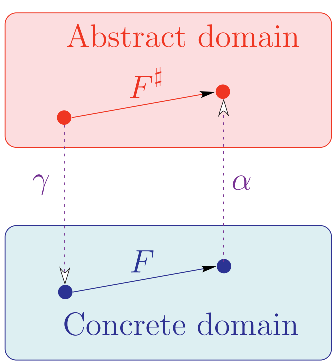
$$\langle P,\ \subseteq\rangle \xleftrightarrow[\alpha]{\gamma} \langle Q,\ \sqsubseteq\rangle$$

iff

- α is monotone
- γ is monotone
- $X \subseteq \gamma \circ \alpha(X)$
- $\alpha \circ \gamma(Y) \sqsubseteq Y$

[1] formalizations using closure operators, ideals, etc. are equivalent.

Function Abstraction

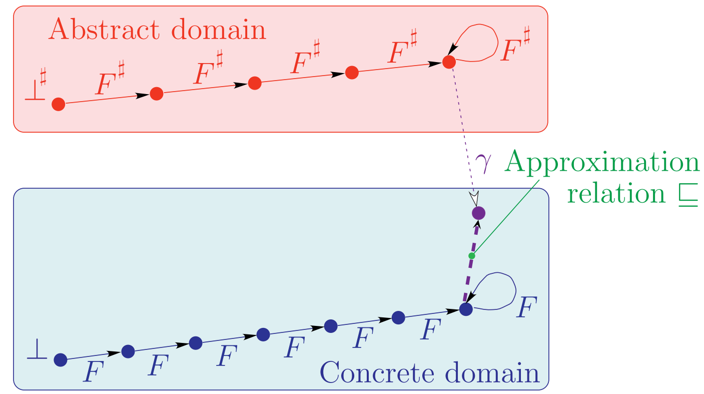


$$F^\sharp = \alpha \circ F \circ \gamma$$

$$\langle P,\ \subseteq\rangle \xleftrightarrow[\alpha]{\gamma} \langle Q,\ \sqsubseteq\rangle \Rightarrow$$

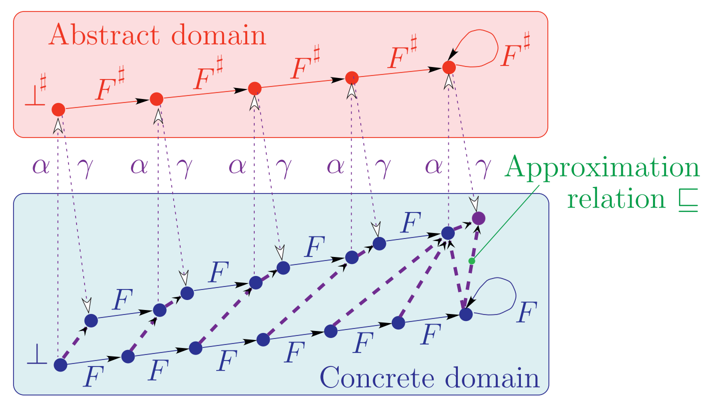
$$\langle P \stackrel{\text{mon}}{\longmapsto} P,\ \dot{\subseteq}\rangle \xleftrightarrow[\lambda F \cdot \alpha \circ F \circ \gamma]{\lambda F^\sharp \cdot \gamma \circ F^\sharp \circ \alpha} \langle Q \stackrel{\text{mon}}{\longmapsto} Q,\ \dot{\sqsubseteq}\rangle$$

Fixpoint Abstraction



$$\mathit{lfp}\, F \sqsubseteq \gamma(\mathit{lfp}\, F^\sharp)$$

Fixpoint Abstraction



$$F^\sharp = \alpha \circ F \circ \gamma \ \Rightarrow\ \mathit{lfp}\, F \sqsubseteq \gamma(\mathit{lfp}\, F^\sharp)$$

Exact/Approximate Fixpoint Abstraction

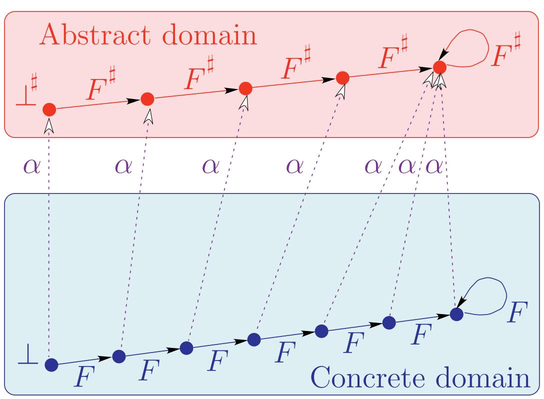
Exact Abstraction:

$$\alpha(\mathit{lfp}\, F) = \mathit{lfp}\, F^\sharp$$

Approximate Abstraction:

$$\alpha(\mathit{lfp}\, F) \sqsubseteq^\sharp \mathit{lfp}\, F^\sharp$$

Exact Fixpoint Abstraction



$$\alpha \circ F = F^\sharp \circ \alpha \;\Rightarrow\; \alpha(\mathit{lfp}\, F) = \mathit{lfp}\, F^\sharp$$

A Few References on Foundations

[9] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th POPL*, pages 238–252, Los Angeles, CA, 1977. ACM Press.

[10] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *6th POPL*, pages 269–282, San Antonio, TX, 1979. ACM Press.

[11] P. Cousot and R. Cousot. Abstract interpretation frameworks. *J. Logic and Comp.*, 2(4):511–547, 1992.

Applications of Abstract Interpretation

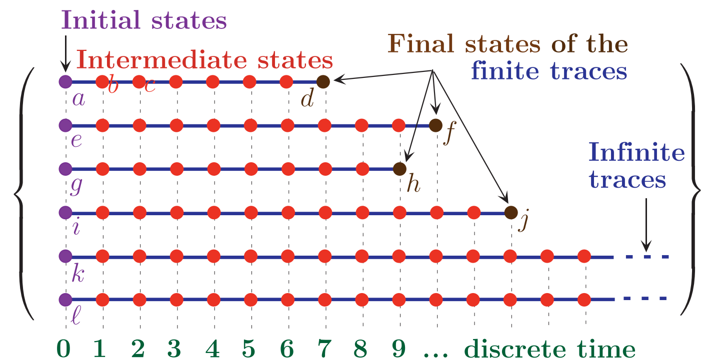
(1) Exact Abstractions

Abstractions of Semantics [12]

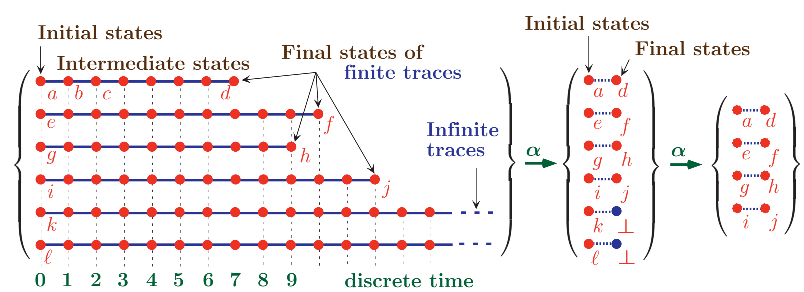
Reference

[12] P. Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. To appear in *Theoretical Computer Science*, (2001).

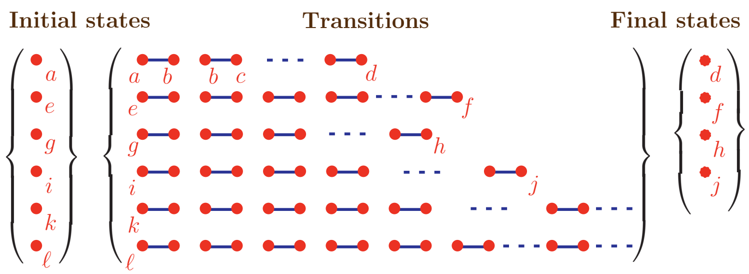
Trace Semantics (Once Again)



Example 1 of Semantics Abstraction

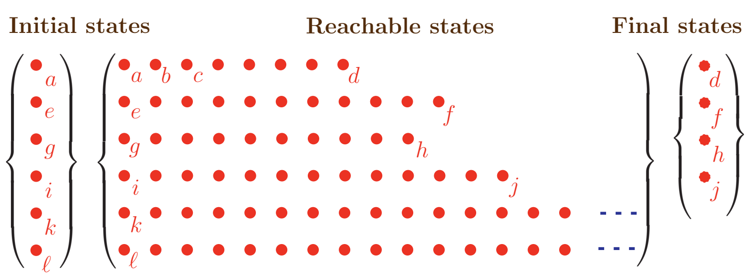


Example 2 of Semantics Abstraction



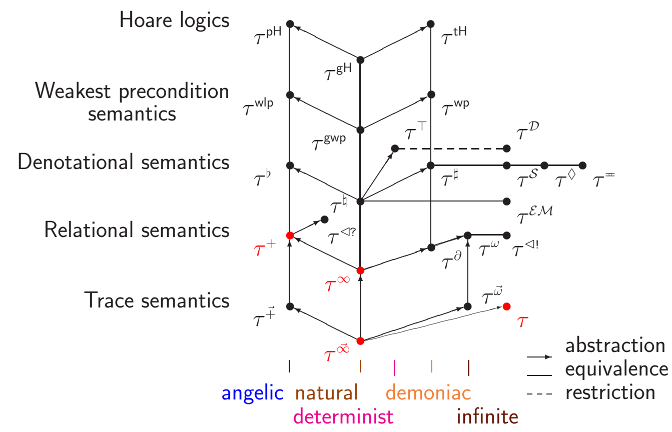
(Small-Step) Operational Semantics

Example 3 of Semantics Abstraction



Partial Correctness / Invariance Semantics

Lattice of Semantics



(2) Effective Approximate Abstractions

Reference

[13] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. *POPL'77*, ACM Press, pp. 238–252.

Effective Abstractions of Semantics

- 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.

Static Program Analysis

Objective of Static Program Analysis

- Program analysis is the automatic static determination of dynamic run-time properties of programs;
- The principle is to compute an approximate semantics of the program to check a given specification;
- *Abstract interpretation* is used to derive, from a standard semantics, the approximate and computable abstract semantics;
- This derivation is itself not (fully) mechanizable.

Basic Idea of Static Program Analysis

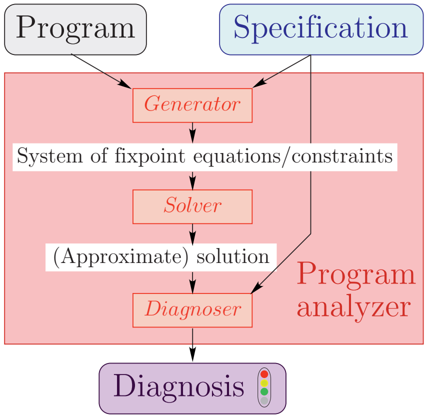
- **Basic idea:** use effective computable approximations of the program semantics;

  Advantage: fully automatic, no need for error-prone user designed model or costly user interaction;

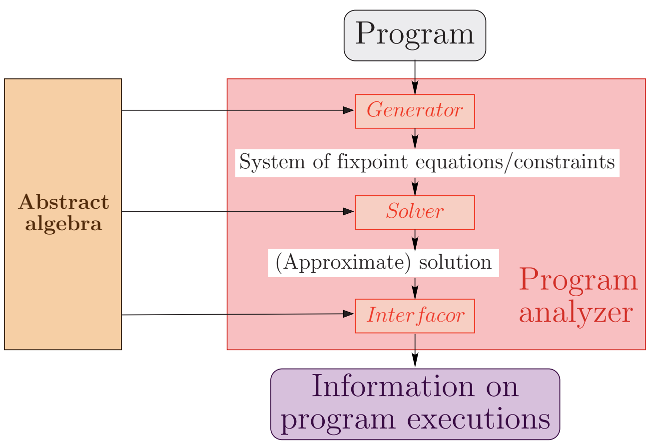
  Drawback: can only handle properties captured by the approximation;

  Remedy: ask the user to choose among a variety of possible approximations (abstract algebras) at various cost/precision ratio.

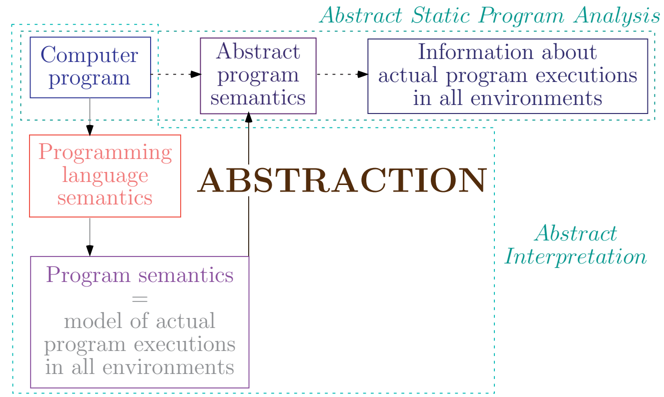
Principle of a Static Program Analyzer



Generic Static Program Analyzer



Design of a Static Program Analyzer by Abstract Interpretation



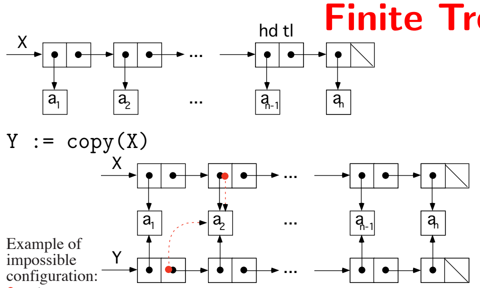
Effective Symbolic Abstractions

Effective 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 Finite Trees

- Program :



- Alias analysis:

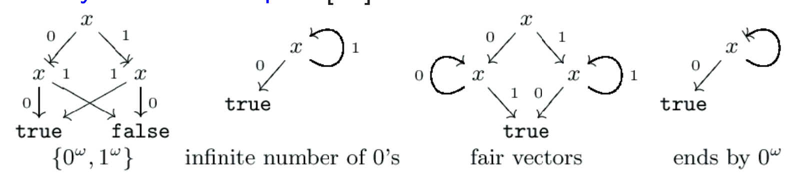
  $\emptyset$
  `Y := copy(X)`
  $\{(\mathtt{X} \mapsto (\mathtt{tl} \mapsto)^i \mapsto \mathtt{hd},\ \mathtt{Y} \mapsto (\mathtt{tl} \mapsto)^j \mapsto \mathtt{hd}) \mid i = j\}$

Reference

[14] A. Deutsch. Interprocedural may-alias analysis for pointers: beyond k-limiting. In *PLDI'94*, pp. 230–241, 1994.

Example of Abstractions of Infinite Sets of Infinite Trees

Binary Decision Graphs: [15]

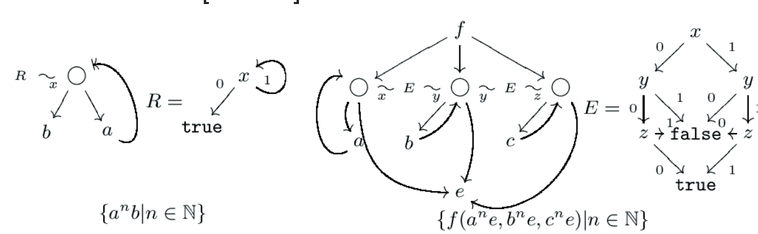


Reference

[15] L. Mauborgne. Binary decision graphs. *SAS '99*, LNCS 1694, pp. 101–116. Springer, 1999.

Example of Abstractions of Infinite Sets of Infinite Trees (Cont'd)

Tree Schemata: [16, 17]



Note that E is the equality relation.

Reference

[16] L. Mauborgne. Improving the representation of infinite trees to deal with sets of trees. *ESOP '2000*, LNCS 1782, pp. 275–289. Springer, 2000.

[17] L. Mauborgne. Tree schemata and fair termination. *SAS '2000*, LNCS 1824, pp. 302–321. Springer, 2000.

A Classical Example: Interval Analysis

Example: interval analysis (1975) [2]

Program to be analyzed:

```
   x := 1;
1:
   while x < 10000 do
2:
       x := x + 1
3:
   od;
4:
```

[2] P. Cousot & R. Cousot, ISOP'1976, POPL'77.

Example: interval analysis (1975) [2]

Equations (abstract interpretation of the semantics):

```
   x := 1;
1:
   while x < 10000 do
2:
       x := x + 1
3:
   od;
4:
```

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

[2] P. Cousot & R. Cousot, ISOP'1976, POPL'77.

Example: interval analysis (1975) [2]

Constraints (abstract interpretation of the semantics):

```
   x := 1;
1:
   while x < 10000 do
2:
       x := x + 1
3:
   od;
4:
```

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

[2] P. Cousot & R. Cousot, ISOP'1976, POPL'77.

Example: interval analysis (1975) [2]

Increasing chaotic iteration, initialization:

```
   x := 1;
1:
   while x < 10000 do
2:
       x := x + 1
3:
   od;
4:
```

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

$$\begin{cases} X_1 = \emptyset \\ X_2 = \emptyset \\ X_3 = \emptyset \\ X_4 = \emptyset \end{cases}$$

[2] P. Cousot & R. Cousot, ISOP'1976, POPL'77.

Example: interval analysis (1975) [2]

Increasing chaotic iteration:

```
   x := 1;
1:
   while x < 10000 do
2:
       x := x + 1
3:
   od;
4:
```

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

$$\begin{cases} X_1 = [1,1] \\ X_2 = [1,1] \\ X_3 = \emptyset \\ X_4 = \emptyset \end{cases}$$

[2] P. Cousot & R. Cousot, ISOP'1976, POPL'77.

Example: interval analysis (1975) [2]

Increasing chaotic iteration:

```
   x := 1;
1:
   while x < 10000 do
2:
       x := x + 1
3:
   od;
4:
```

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

$$\begin{cases} X_1 = [1,1] \\ X_2 = \emptyset \\ X_3 = \emptyset \\ X_4 = \emptyset \end{cases}$$

[2] P. Cousot & R. Cousot, ISOP'1976, POPL'77.

Example: interval analysis (1975) [2]

Increasing chaotic iteration:

```
   x := 1;
1:
   while x < 10000 do
2:
       x := x + 1
3:
   od;
4:
```

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

$$\begin{cases} X_1 = [1,1] \\ X_2 = [1,1] \\ X_3 = [2,2] \\ X_4 = \emptyset \end{cases}$$

[2] P. Cousot & R. Cousot, ISOP'1976, POPL'77.

Example: interval analysis (1975) [2]

Increasing chaotic iteration:

```
   x := 1;
1:
   while x < 10000 do
2:
       x := x + 1
3:
   od;
4:
```

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

$$\begin{cases} X_1 = [1,1] \\ X_2 = [1,2] \\ X_3 = [2,2] \\ X_4 = \emptyset \end{cases}$$

[2] P. Cousot & R. Cousot, ISOP'1976, POPL'77.

Example: interval analysis (1975) [2]

Increasing chaotic iteration: **convergence??**

```
   x := 1;
1:
   while x < 10000 do
2:
       x := x + 1
3:
   od;
4:
```

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

$$\begin{cases} X_1 = [1,1] \\ X_2 = [1,3] \\ X_3 = [2,3] \\ X_4 = \emptyset \end{cases}$$

[2] P. Cousot & R. Cousot, ISOP'1976, POPL'77.

Example: interval analysis (1975) [2]

Increasing chaotic iteration: **convergence?**

```
   x := 1;
1:
   while x < 10000 do
2:
       x := x + 1
3:
   od;
4:
```

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

$$\begin{cases} X_1 = [1,1] \\ X_2 = [1,2] \\ X_3 = [2,3] \\ X_4 = \emptyset \end{cases}$$

[2] P. Cousot & R. Cousot, ISOP'1976, POPL'77.

Example: interval analysis (1975) [2]

Increasing chaotic iteration: **convergence???**

```
   x := 1;
1:
   while x < 10000 do
2:
       x := x + 1
3:
   od;
4:
```

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

$$\begin{cases} X_1 = [1,1] \\ X_2 = [1,3] \\ X_3 = [2,4] \\ X_4 = \emptyset \end{cases}$$

[2] P. Cousot & R. Cousot, ISOP'1976, POPL'77.

Example: interval analysis (1975) [2]

Increasing chaotic iteration: **convergence????**

```
   x := 1;
1:
   while x < 10000 do
2:
       x := x + 1
3:
   od;
4:
```

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

$$\begin{cases} X_1 = [1,1] \\ X_2 = [1,4] \\ X_3 = [2,4] \\ X_4 = \emptyset \end{cases}$$

[2] P. Cousot & R. Cousot, ISOP'1976, POPL'77.

Example: interval analysis (1975) [2]

Increasing chaotic iteration: **convergence??????**

```
   x := 1;
1:
   while x < 10000 do
2:
       x := x + 1
3:
   od;
4:
```

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

$$\begin{cases} X_1 = [1,1] \\ X_2 = [1,5] \\ X_3 = [2,5] \\ X_4 = \emptyset \end{cases}$$

[2] P. Cousot & R. Cousot, ISOP'1976, POPL'77.

Example: interval analysis (1975) [2]

Increasing chaotic iteration: **convergence?????**

```
   x := 1;
1:
   while x < 10000 do
2:
       x := x + 1
3:
   od;
4:
```

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

$$\begin{cases} X_1 = [1,1] \\ X_2 = [1,4] \\ X_3 = [2,5] \\ X_4 = \emptyset \end{cases}$$

[2] P. Cousot & R. Cousot, ISOP'1976, POPL'77.

Example: interval analysis (1975) [2]

Increasing chaotic iteration: **convergence???????**

```
   x := 1;
1:
   while x < 10000 do
2:
       x := x + 1
3:
   od;
4:
```

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

$$\begin{cases} X_1 = [1,1] \\ X_2 = [1,5] \\ X_3 = [2,6] \\ X_4 = \emptyset \end{cases}$$

[2] P. Cousot & R. Cousot, ISOP'1976, POPL'77.

Example: interval analysis (1975) [2]

Convergence speed-up by extrapolation:

```
   x := 1;
1:
   while x < 10000 do
2:
       x := x + 1
3:
   od;
4:
```

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

$$\begin{cases} X_1 = [1,1] \\ X_2 = [1, +\infty] \quad \Leftarrow \text{widening} \\ X_3 = [2,6] \\ X_4 = \emptyset \end{cases}$$

[2] P. Cousot & R. Cousot, ISOP'1976, POPL'77.

Example: interval analysis (1975) [2]

Decreasing chaotic iteration:

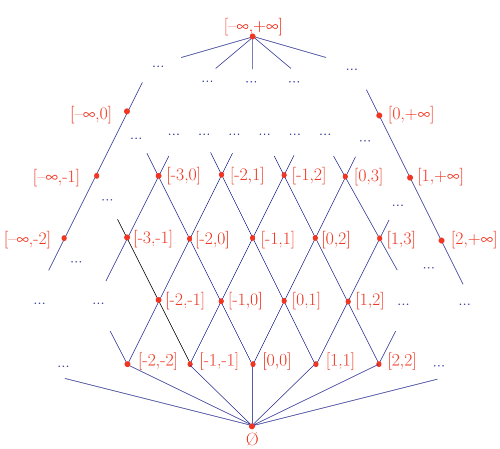
```
   x := 1;
1:
   while x < 10000 do
2:
       x := x + 1
3:
   od;
4:
```

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

$$\begin{cases} X_1 = [1,1] \\ X_2 = [1, +\infty] \\ X_3 = [2, +\infty] \\ X_4 = \emptyset \end{cases}$$

[2] P. Cousot & R. Cousot, ISOP'1976, POPL'77.

Widening



Example: interval analysis (1975) [2]

Decreasing chaotic iteration:

```
   x := 1;
1:
   while x < 10000 do
2:
       x := x + 1
3:
   od;
4:
```

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

$$\begin{cases} X_1 = [1,1] \\ X_2 = [1, 9999] \\ X_3 = [2, +\infty] \\ X_4 = \emptyset \end{cases}$$

[2] P. Cousot & R. Cousot, ISOP'1976, POPL'77.

Example: interval analysis (1975) [2]

Decreasing chaotic iteration:

```
   x := 1;
1:
   while x < 10000 do
2:
      x := x + 1
3:
   od;
4:
```

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

$$\begin{cases} X_1 = [1,1] \\ X_2 = [1, 9999] \\ X_3 = [2, 10000] \\ X_4 = \emptyset \end{cases}$$

[2] P. Cousot & R. Cousot, ISOP'1976, POPL'77.

Example: interval analysis (1975) [2]

Result of the interval analysis:

```
   x := 1;
1: {x = 1}
   while x < 10000 do
2: {x ∈ [1, 9999]}
      x := x + 1
3: {x ∈ [2, 10000]}
   od;
4: {x = 10000}
```

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

$$\begin{cases} X_1 = [1,1] \\ X_2 = [1, 9999] \\ X_3 = [2, 10000] \\ X_4 = [10000, 10000] \end{cases}$$

[2] P. Cousot & R. Cousot, ISOP'1976, POPL'77.

Example: interval analysis (1975) [2]

Final solution:

```
   x := 1;
1:
   while x < 10000 do
2:
      x := x + 1
3:
   od;
4:
```

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

$$\begin{cases} X_1 = [1,1] \\ X_2 = [1, 9999] \\ X_3 = [2, 10000] \\ X_4 = [10000, 10000] \end{cases}$$

[2] P. Cousot & R. Cousot, ISOP'1976, POPL'77.

A More Intriguing Example

```
program Variant_of_McCarthy_91_function;
   var X, Y : integer;
   function F(X : integer) : integer;
   begin
      if X > 100 then F := X - 10
      else F := F(F(F(F(F(F(F(F(F(F(X + 90))))))))));
   end;
begin
   readln(X);
   Y := F(X);
   { Y ∈ [91, +∞] }
end.
```

Reference

[18] F. Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Proc. FMPA*, LNCS 735, pages 128–141. Springer, 1993.

Probabilistic Program Analysis [4]

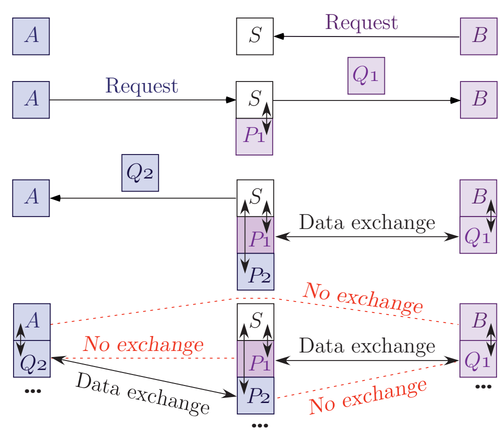
```
double x, i;
assume (-1.0 < x < 0.0);
i = 0.0;
while (i < 3.0) {
  x += uniform();
  i += 1.0;
};
assert (x < 1.0);
```

With 99% safety:

- the probability of the outcome $(x < 1)$ is less than 0.859,
- assuming:
  - worst-case nondeterministic choices of the precondition $(-1.0 < x < 0.0)$,
  - random choices `uniform()` chosen in $[0, 1]$ with the Lebesgue uniform distribution.

[2] D. Monniaux, SAS'00, POPL'01

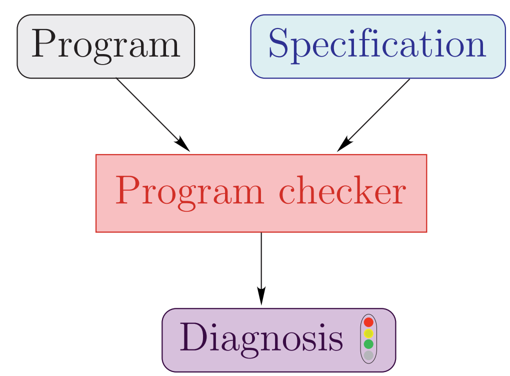
Communication Topology of Mobile Processes [3]



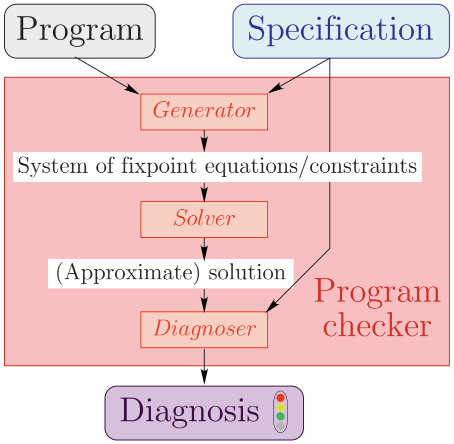
[3] J. Feret, SAS'00, ENTCS Vol. 39

Static Program Checking

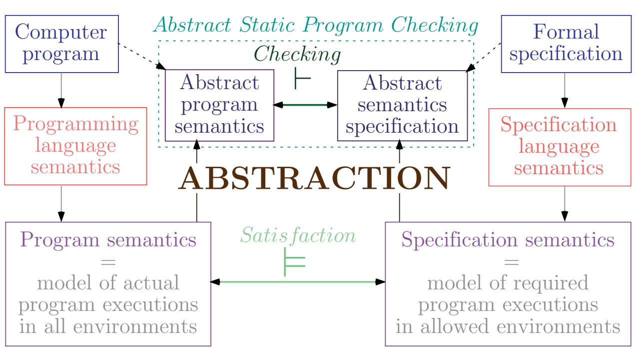
Objective of Static Program Checking



Principle of a Static Program Checker



Design of a Static Program Checker by Abstract Interpretation



Example: interval analysis (1975) [2]

Exploitation of the result of the interval analysis:

```
   x := 1;
1: {x = 1}
   while x < 10000 do
2: {x ∈ [1, 9999]}
      x := x + 1
3: {x ∈ [2, 10000]}          ⟵ no overflow
   od;
4: {x = 10000}
```

[2] P. Cousot & R. Cousot, ISOP'1976, POPL'77.

Other Examples of Faultless Execution Checks

- Absence of runtime errors (array bounds violations, arithmetic overflow, erroneous data accesses, etc.),
- Absence of memory leaks (dangling pointers, uninitialized variables, etc.),
- Handling of all possible runtime exceptions (failures of I/O and system calls, etc.),
- No resource contention and race conditions in concurrent programs (deadlocks & livelocks),
- Termination / non termination conditions,
- Etc.

Static Program Testing

Combining Empirical and Formal Methods

- The user provides local formal abstractions of the program specifications using predefined abstractions [4];
- The program is evaluated by abstract interpretation of the formal semantics of the program [5];
- If the local abstract specification cannot be proved correct, a more precise abstract domain must be considered [6];
- The process is repeated until appropriate coverage of the specification.

[4] thus replacing infinitely many test data.
[5] thus replacing program execution on the test data.
[6] similarly to different test data.

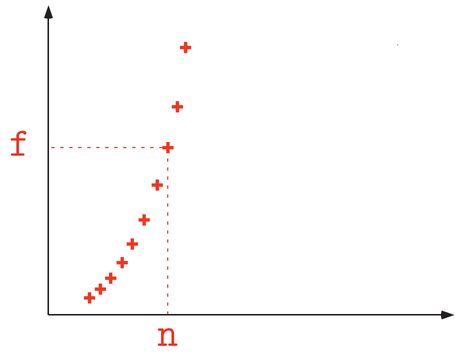
Abstract checking versus Abstract Testing

- Abstract checking: specification derived automatically from the program (e.g. using the language specification for run-time errors);
- Abstract testing: specification provided by the programmer.

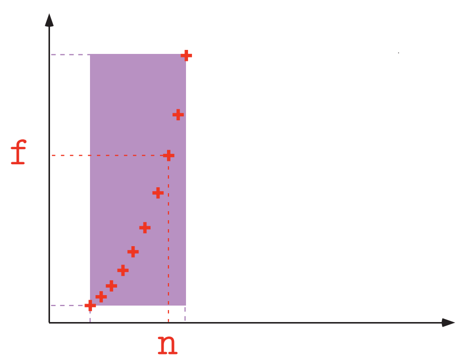
Abstract Program Testing

Debugging	Abstract testing
Run the program	Compute the abstract semantics
On test data	Choosing a predefined abstraction
Checking if all right	Checking user-provided abstract assertions
Providing more tests	With more refined abstractions
Until coverage	Until enough assertions proved or no predefined abstraction can do.

Example of predefined abstraction



Example of predefined abstraction: intervals



A Tiny Example

```
0: { n:[−∞,+∞]?; f:[−∞,+∞]? }
  read(n);
1: { n:[0,+∞]; f:[−∞,+∞]? }
  f := 1;
2: { n:[0,+∞]; f:[1,+∞] }
  while (n <> 0) do
    3: { n:[1,+∞]; f:[1,+∞] }
    f := (f * n);
    4: { n:[1,+∞]; f:[1,+∞] }
    n := (n - 1)
    5: { n:[0,+∞]; f:[1,+∞] }
  od;
6: { n:[0,0]; f:[1,+∞] }
sometime true;;
```

- static analyzer inference
- diagnosis: definite error
- no error
- potential error
- user program
- user specification

A Tiny Example (Cont'd)

```
0: { n:⊥; f:⊥ }
  initial (n < 0);
1: { n:[−∞,-1]; f:[−∞,+∞]? }
  f := 1;
2: { n:[−∞,-1]; f:[−∞,+∞] }
  while (n <> 0) do
    3: { n:[−∞,-1]; f:[−∞,+∞] }
    f := (f * n);
    4: { n:[−∞,-1]; f:[−∞,+∞] }
    n := (n - 1)
    5: { n:[−∞,-2]; f:[−∞,+∞] }
  od
6: { n:⊥; f:⊥ }
```

- static analyzer inference
- user specification
- user program
- diagnosis: no error
- potential error
- potential error
- ⊥ unreachable code

A More Intriguing Example

```
program Variant_of_McCarthy_91_function;
    var X, Y : integer;
    function F(X : integer) : integer;
    begin
        if X > 100 then F := X - 10    91
        else F := F(F(F(F(F(F(F(F(F(F(X + ~~90~~))))))))));
    end;
begin
    readln(X);
    {% X > 100 %}
    Y := F(X);
    {% sometime true %}
end.
```

Example of cycle: F(100) → F(190) → F(180) → F(170) → F(160) → F(150) → F(140) → F(130) → F(120) → F(110) → F(100) → ...

Comparing with program debugging

- Similarity: user interaction, on the source code;
- Essential differences:
  - user provided test data are replaced by abstract specifications;
  - evaluation of an abstract semantics instead of program execution/simulation;
  - one can prove the absence of (some categories of) bugs, not only their presence;
  - abstract evaluation can be forward and/or backward (reverse execution).

Examples of Functional Specifications for Abstract Testing

- Worst-case execution/response time in real-time systems running on a computer with pipelines and caches;
- Periodicity of some action over time/with respect to some clock;
- Possible reactions to real-time event/message sequences;
- Compatibility with state/transition/sequence diagrams/charts;
- Absence of deadlock/livelock with different scheduling policies;

Conclusion

Concluding Remarks

- Program debugging is still the prominent industrial program "verification" method. Complementary program verification methods are needed;
- Fully mechanized program verification by formal methods is either impossible (e.g. typing/program analysis) or extremely costly since it ultimately requires user interaction (e.g. abstract model checking/deductive methods for large programs);
- For program verification, semantic abstraction is mandatory but difficult whence hardly automatizable, even with the help of programmers;

Concluding Suggestions

- Abstract interpretation introduces the idea of safe approximation within formal methods;
- So you might think to use it for partial verification of the source specification/program code:
  - Abstract checking (fully automatic and exhaustive diagnosis on run-time safety properties),
  - Abstract testing (interactive/planned diagnosis on functional, behavioural and resources-usage requirements),

  using tools providing predefined abstractions. .../...

- Does apply to any computer-related language with a well-specified semantics describing computations (e.g. specification languages, data base languages, sequential, concurrent, distributed, mobile, logical, functional, object oriented, ... programming languages, etc.);
- Does apply to any property and combinations of properties (such as safety, liveness, timing, event preconditions, ...);
- Can follow up program modifications over time;
- Very cost effective, especially in early phases of program development.

Industrialization of Static Analysis/Checking by Abstract Interpretation

- Connected Components Corporation (U.S.A.), L. Harrison, 1993[7];
- AbsInt Angewandte Informatik GmbH (Germany), R. Wilhelm & C. Ferdinand, 1998;
- PolySpace Polyspace Technologies (France), A. Deutsch & D. Pilaud, 1999.

[7] Internal use for compiler design.

DAEDALUS European project on the verification of critical real-time avionic software (oct. 2000 — sep. 2002):

- P. Cousot (ENS, France), scientific coordinator;
- R. Cousot (École polytechnique, France);
- A. Deutsch & D. Pilaud (Polyspace Technologies, France);
- C. Ferdinand (AbsInt, Germany);
- É. Goubault (CEA, France);
- N. Jones (DIKU, Denmark);
- F. Randimbivololona & J. Souyris (EADS Airbus, France), coord.;
- M. Sagiv (Univ. Tel Aviv, Israel);
- H. Seidel (Univ. Trier, Germany);
- R. Wilhelm (Univ. Sarrebrücken, Germany);

A reference (with a large bibliography)

P. Cousot.
Abstract interpretation based formal methods and future challenges.

In R. Wilhelm (editor), *« Informatics — 10 Years Back, 10 Years Ahead »*.
Volume 2000 of *Lecture Notes in Computer Science*, pages 138–156. Springer-Verlag, 2001.

An extended electroning version is also available on Springer-Verlag web site together with a very long electroning version with a complete bibliography.