# Progress on Abstract Interpretation Based Formal Methods and Future Challenges


**Patrick COUSOT**

École Normale Supérieure

45 rue d'Ulm, 75230 Paris cedex 05, France

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



Dagstuhl's 10th Anniversary Conf., Saarbr cken, Aug. 28–31, 2000

# Abstract

In order to contribute to the software reliability problem, tools have been designed in order to analyze statically the run-time behavior of programs. Because the correctness problem is undecidable, some form of approximation is needed. The whole purpose of abstract interpretation is to formalize this idea of approximation. We illustrate informally the application of abstraction to the semantics of programming languages as well as to program static analysis. The main point is that in order to reason or compute about a complex system, some information must be lost, that is the observation of executions must be at a high level of abstraction.

In the second part of the talk, we compare program static analysis with deductive methods, model-checking and type inference. Their foundational ideas are shortly reviewed, and the shortcomings of these four tools are discussed, including when they are combined. Alternatively, since program debugging is still the main program verification method used in industry, we suggest to combine formal with informal methods.

Finally, the grand challenge for all formal methods and tools is to solve the software reliability, trustworthiness or robustness problems. Few challenges more specific to program analysis by abstract interpretation are shortly discussed.

The published slides slightly extend those of the presentation and include a shortened bibliography, mainly restricted to result obtained in our research group.

# Motivations and Overview

# 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 softwares 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;

# 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 head (Thinking/intellectual tools, this morning session);
- Use our computer (Mechanical tools, this afternoon session).

# Computer Aided Program Verification

- Empirical program verification methods (execute/simulate the program in enough representative possible environments):
  - Debugging ,
  - Simulation;
- Formal program verification methods (mecanically prove that program execution is correct in all specified environments):
  - Deductive methods ,
  - Model checking ,
  - Program typing ,
  - Program analysis.

# Undecidability and Approximation

- Since program verification is undecidable, computer aided program verification methods are all partial/incomplete;
- They all involve some form of approximation:
  - practical complexity limitations,
  - required user interaction,
  - semi-algorithms or finiteness hypotheses,
  - restricted specifications or programs;
- Most of these approximations are formalized by Abstract Interpretation.

# Abstract Interpretation

- Abstract Interpretation is a theory of approximation of the behavior of dynamic discrete systems (such as the formal semantics of programs);
- Since such behaviors can be characterized by fixpoints, the theory essentially provides constructive and effective methods for fixpoint approximation and checking by abstraction.

**Seminal reference**

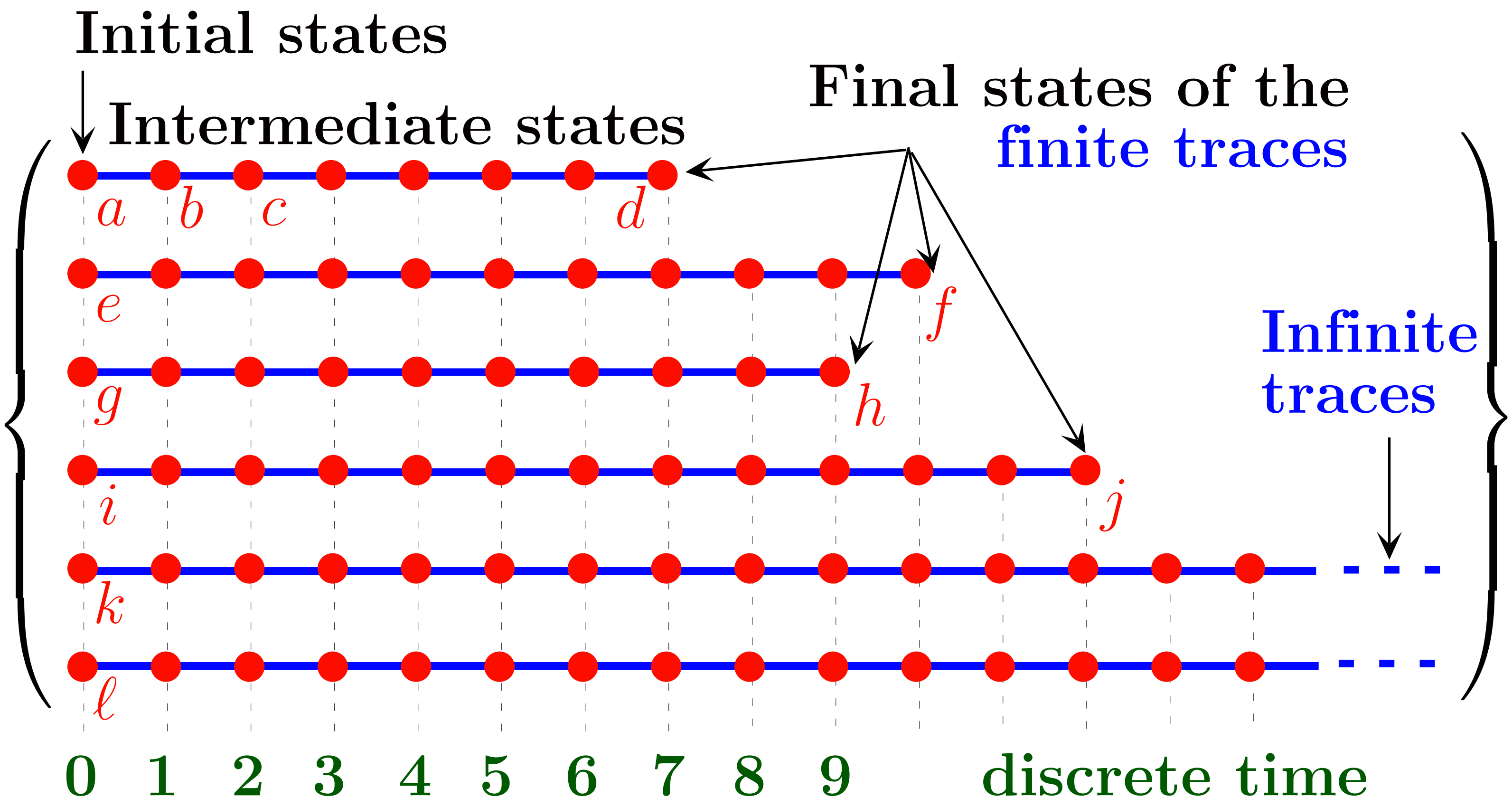

– P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conf. Record of the 4th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages POPL'77*, Los Angeles, CA, 1977. ACM Press, pp.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

# Semantics

# Semantics: Intuition

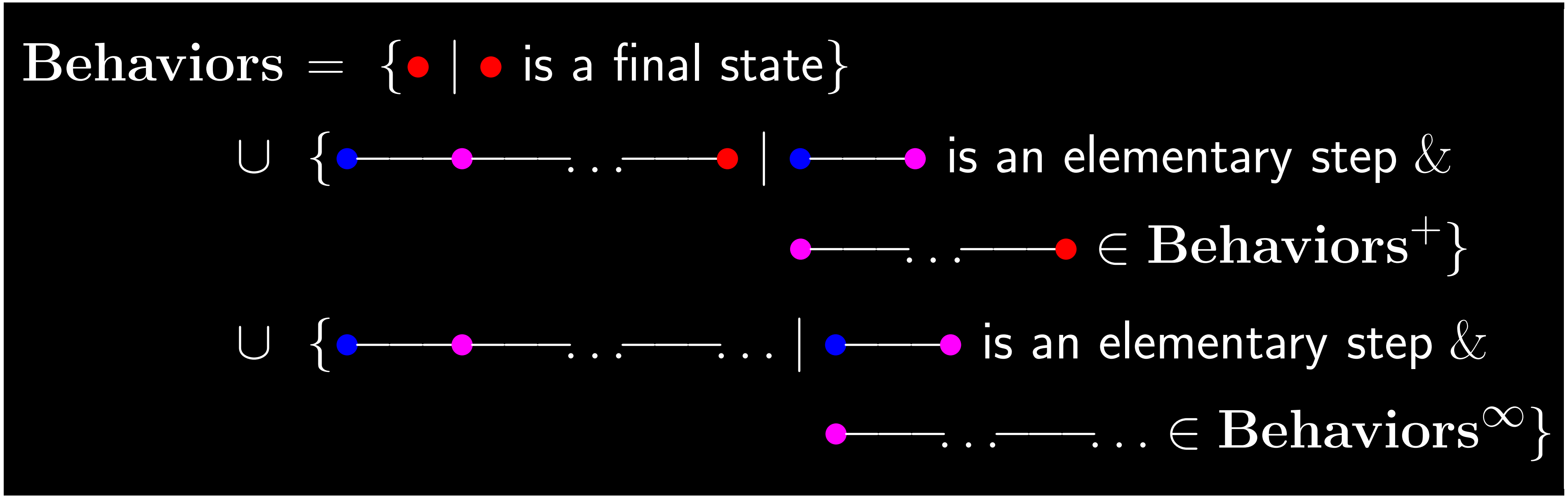
- The semantics of a language defines the semantics of any program written in this language;
- The semantics of a program provides a formal mathematical model of all possible behaviors of a computer system executing this program (interacting with any possible environment);
- Any semantics of a program 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 [7, 9]



# Fixpoints

# Least Fixpoints: Intuition [7, 9]




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

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

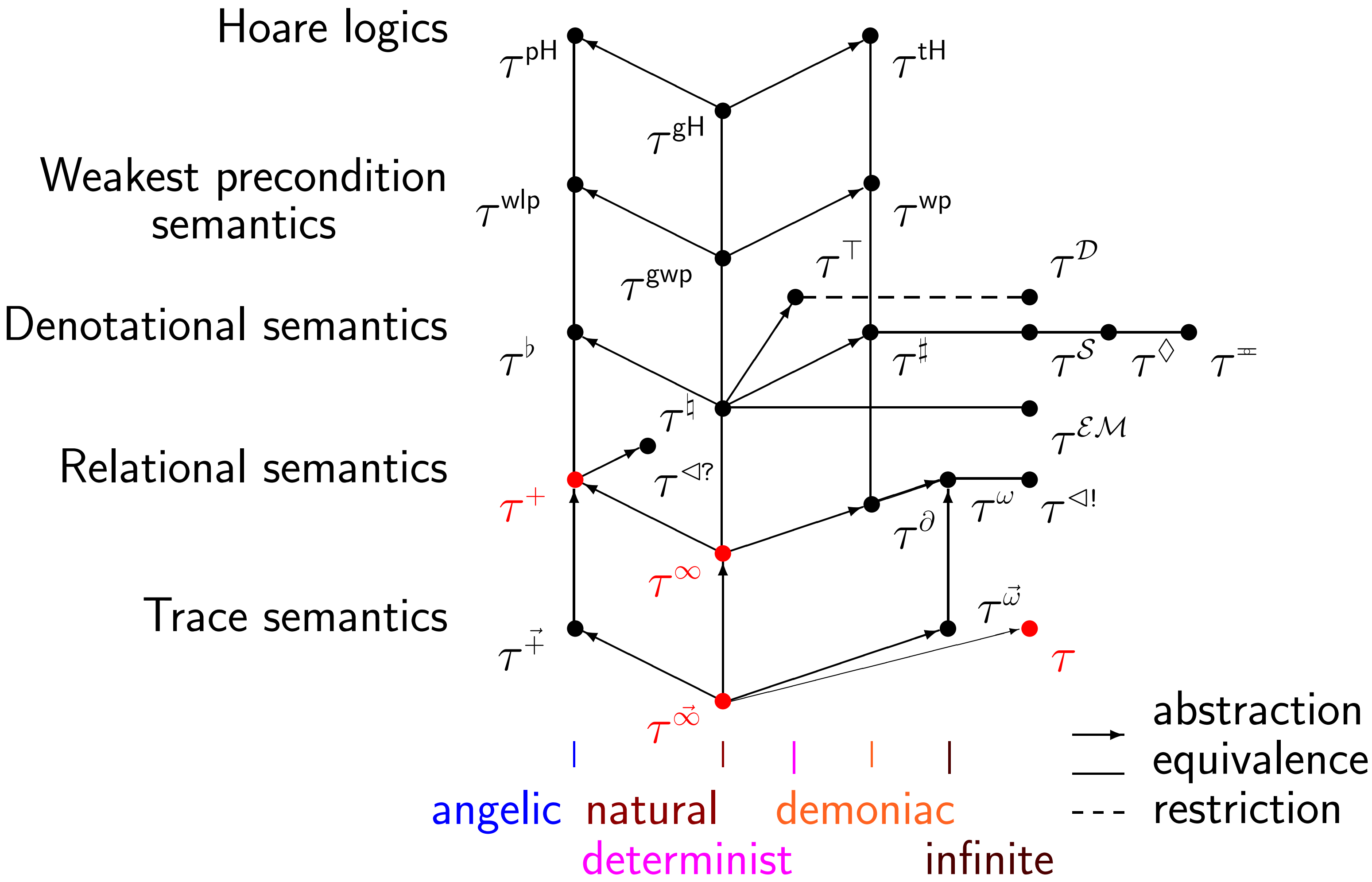
# Abstraction

# Abstraction: Intuition

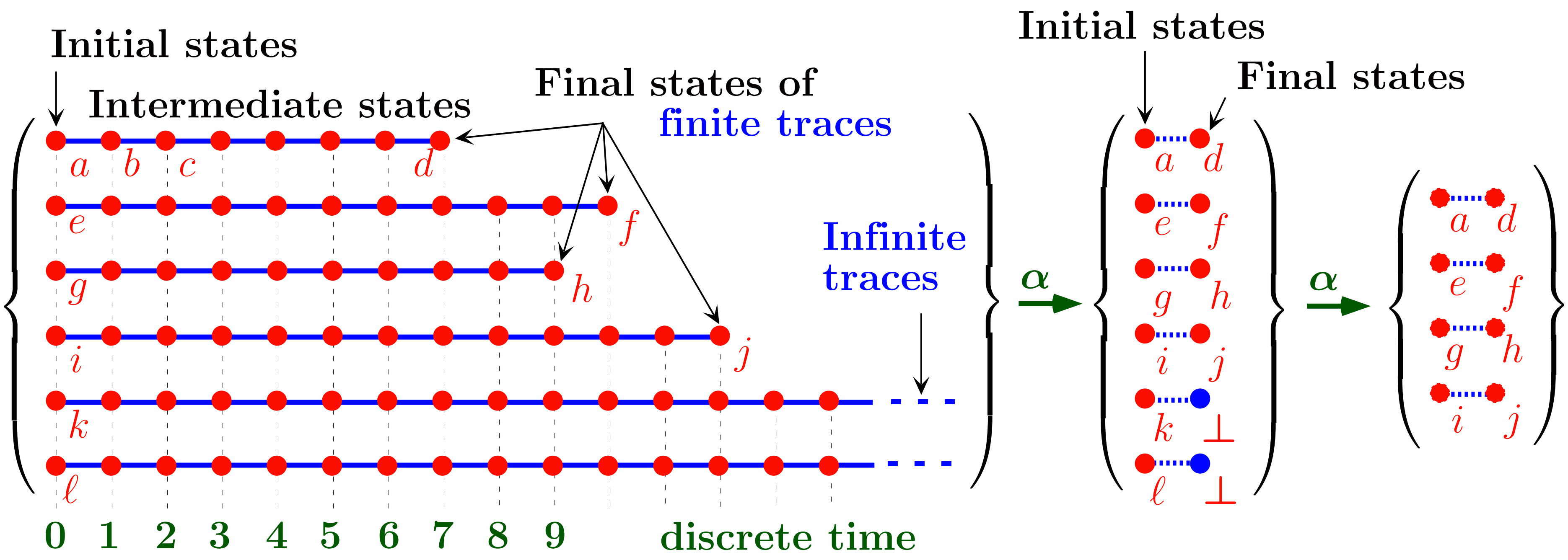
- Abstract interpretation formalizes the intuitive idea that a semantics is more or less precise according to the considered observation level of the program executions;

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

# Lattice of Semantics [9]



# Example 1 of Abstraction [1]




Trace semantics

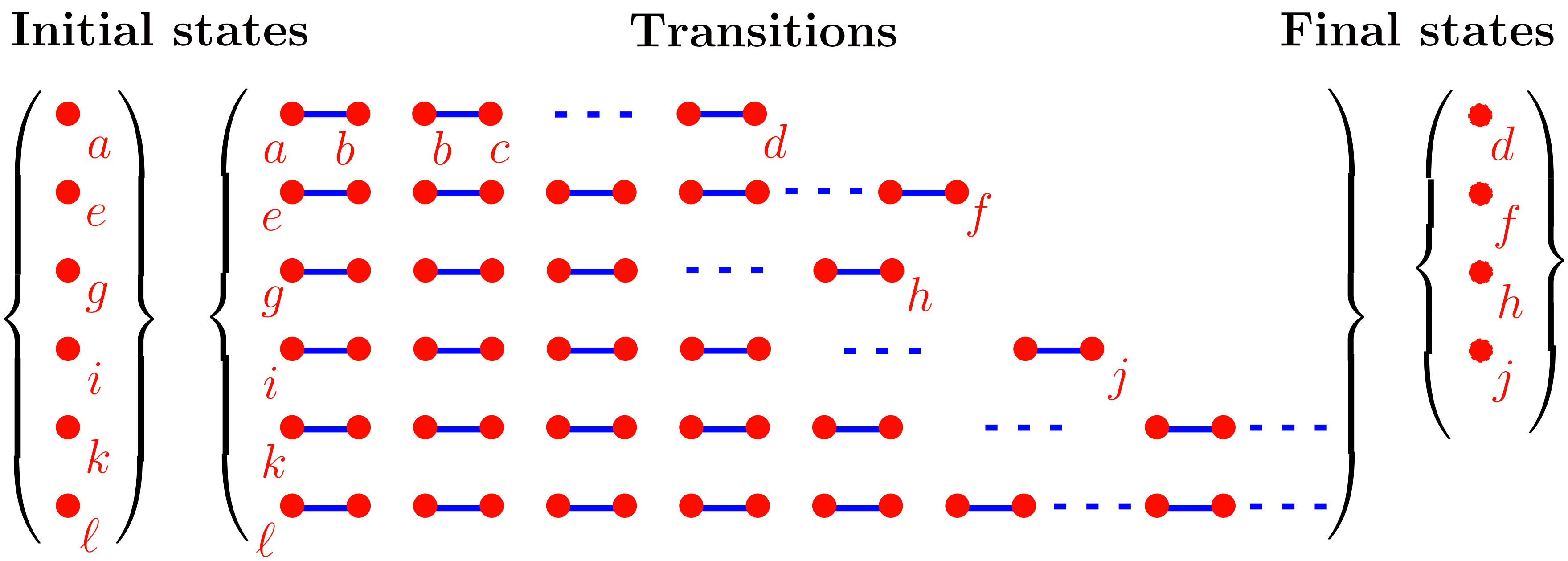
Denotational semantics

Natural semantics

---

[1] P. Cousot. *Constructive design of a hierarchy of semantics of a transition system by abstract interpretation*. To appear in TCS (2000).

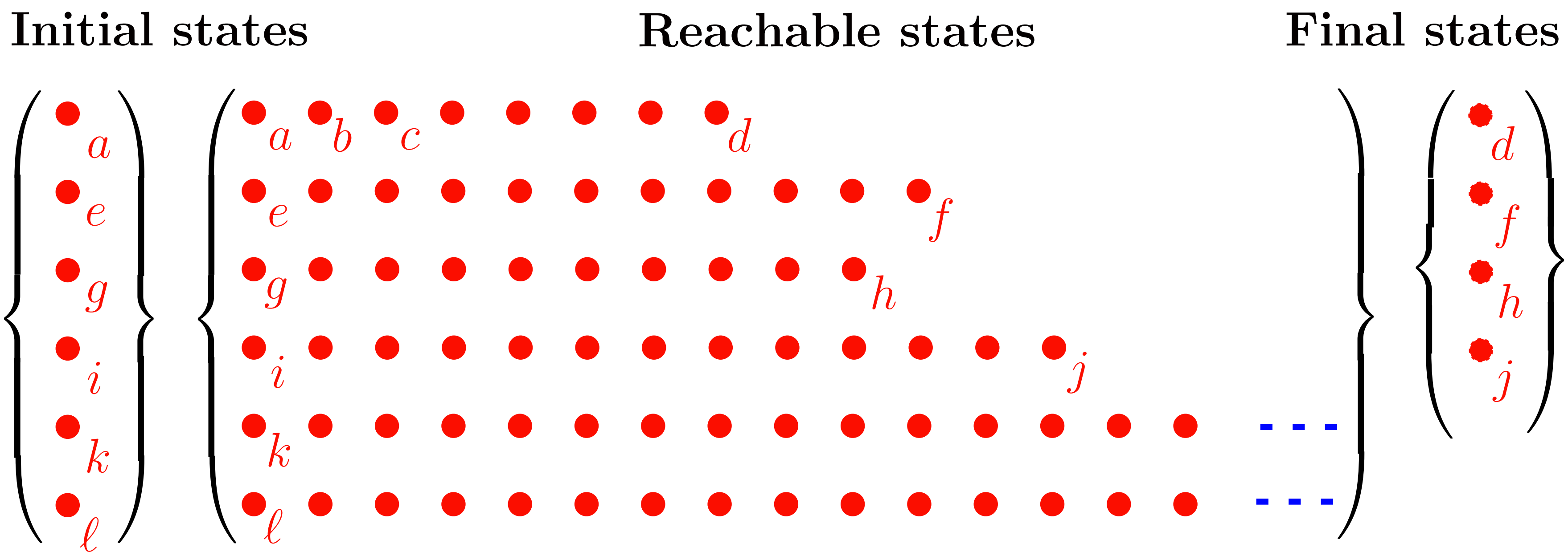
# Example 2 of Abstraction [2]




## (Small-Step) Operational Semantics

[2] P. Cousot. *Constructive design of a hierarchy of semantics of a transition system by abstract interpretation*. To appear in TCS (2000).

# Example 3 of Abstraction [3]




**Partial Correctness / Invariance Semantics**

---

[3] P. Cousot. *Constructive design of a hierarchy of semantics of a transition system by abstract interpretation*. To appear in TCS (2000).

# Effective Abstractions

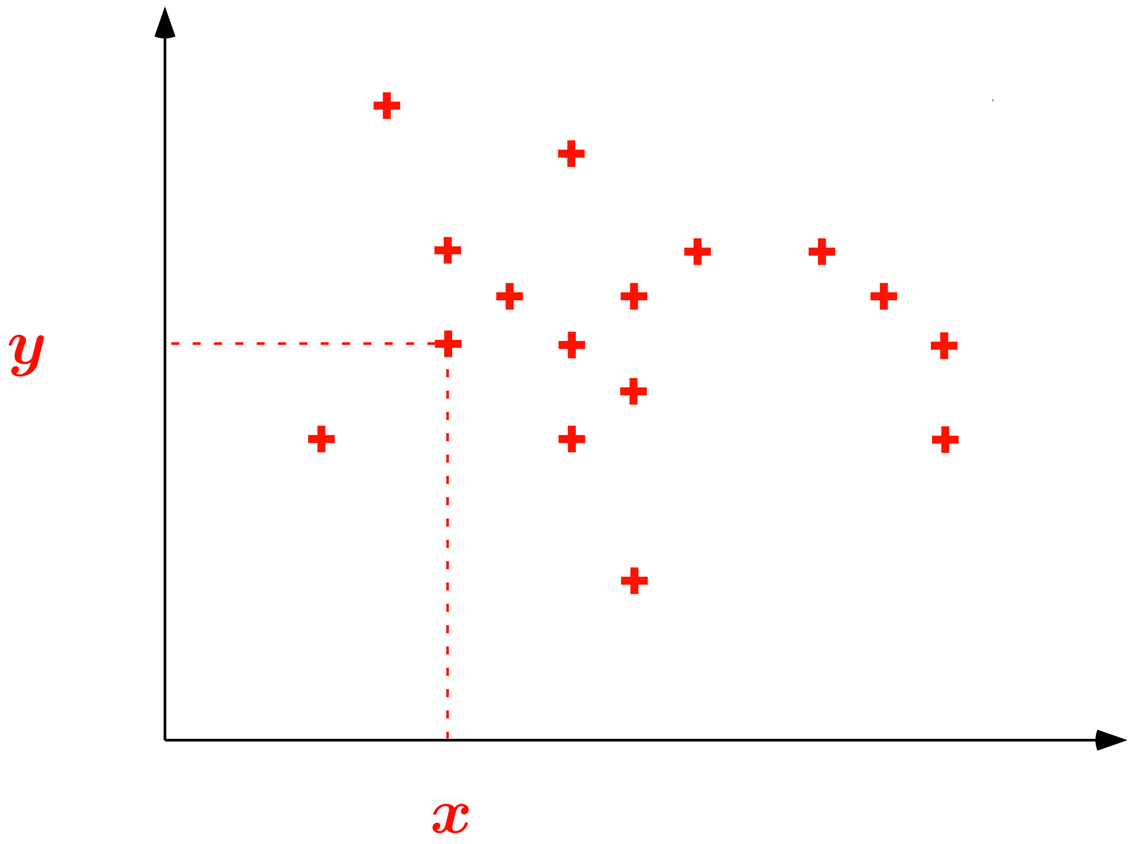
# Effective Abstractions

- 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;

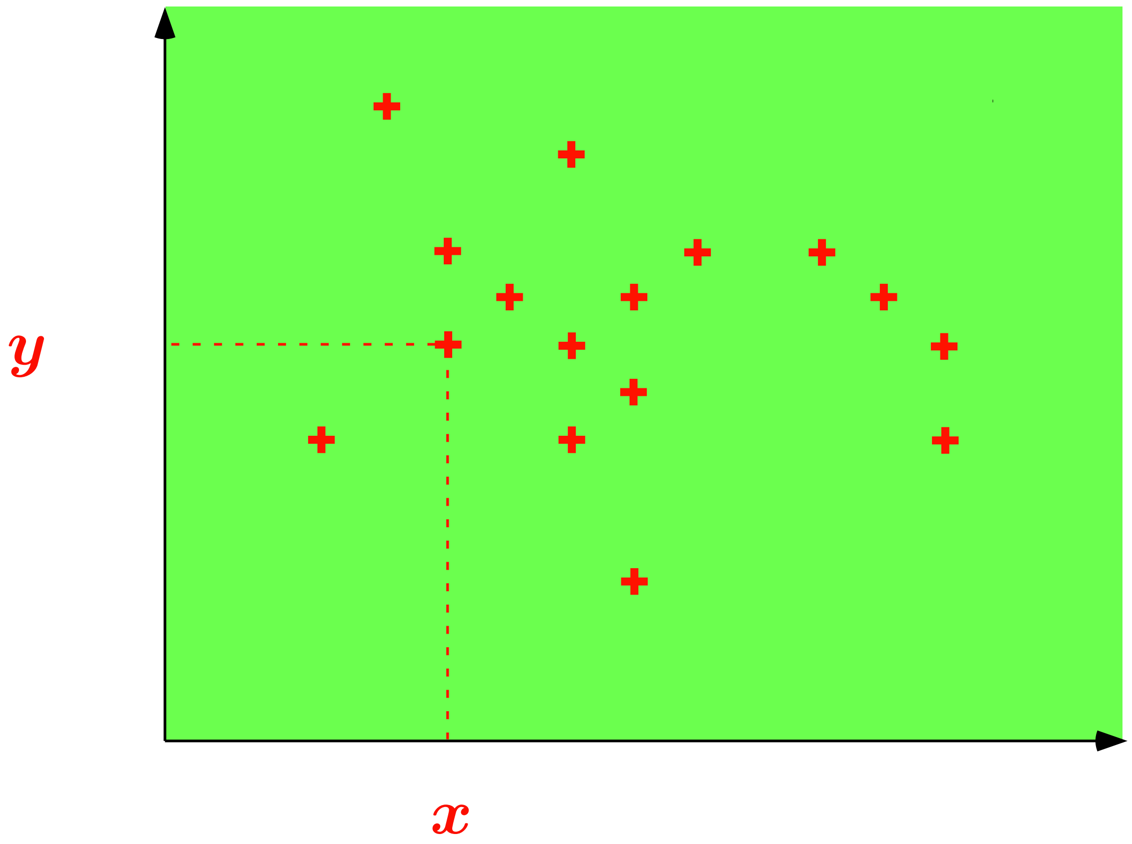
- By effective computation of the abstract semantics , the computer is able to analyze the behavior of programs and of software before and without executing them [10].

# Effective Abstractions of an [In]finite Set of Points;



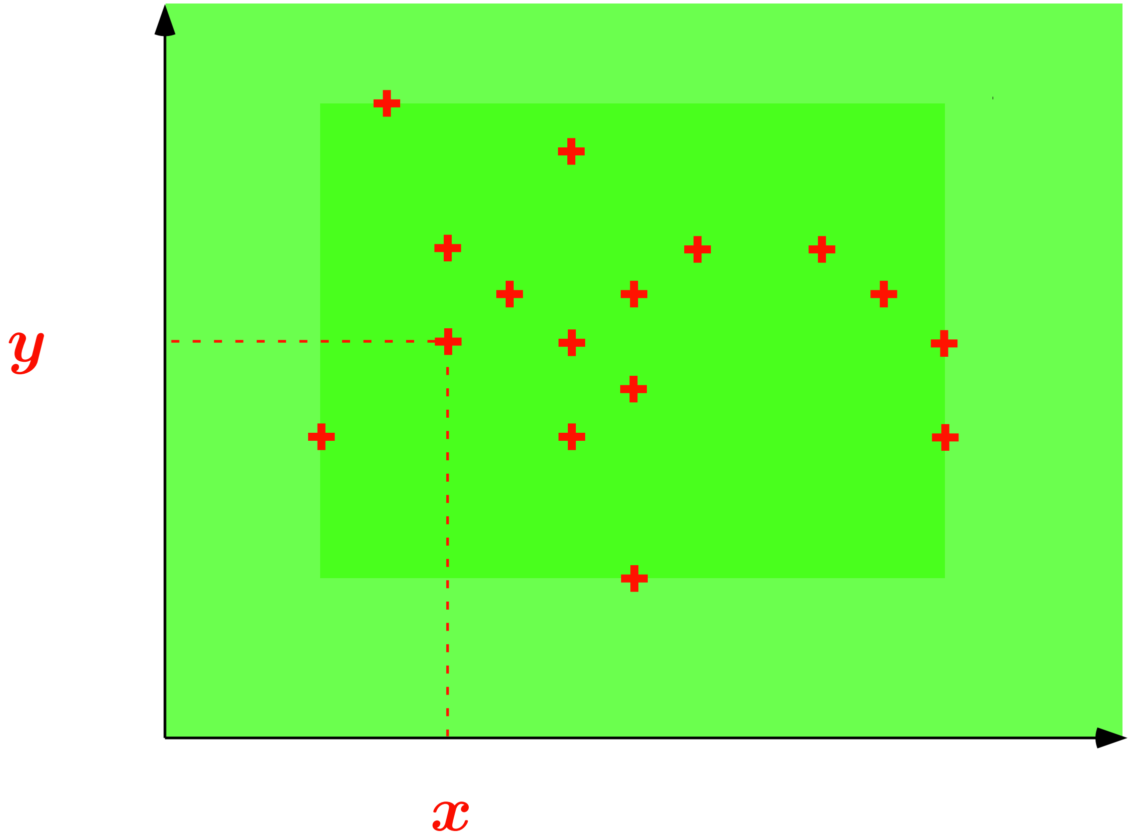

$$\{\ldots, \langle 19,\ 88\rangle, \ldots, \langle 19,\ 99\rangle, \ldots\}$$

# Effective Abstractions of an [In]finite Set of Points; Example 1: Signs [12]



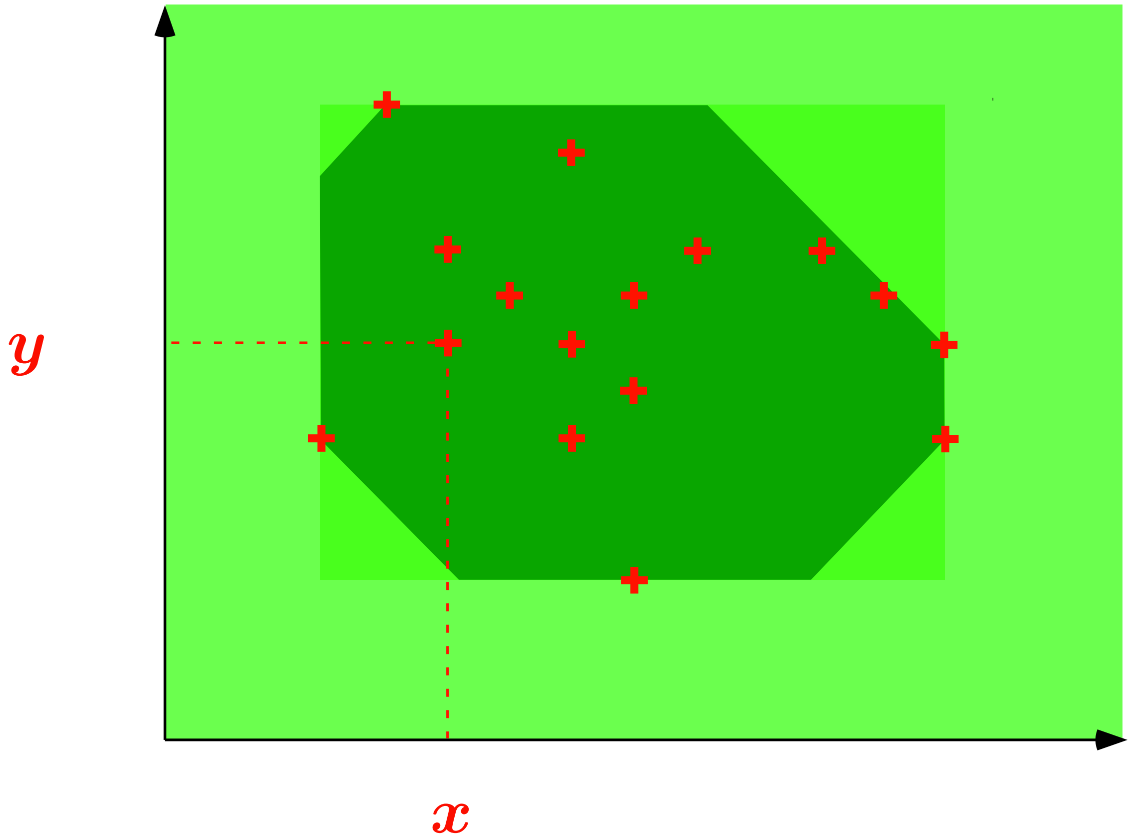

$$\begin{cases} x \geq 0 \\ y \geq 0 \end{cases}$$

# Effective Abstractions of an [In]finite Set of Points; Example 2: Intervals [10 , 11]



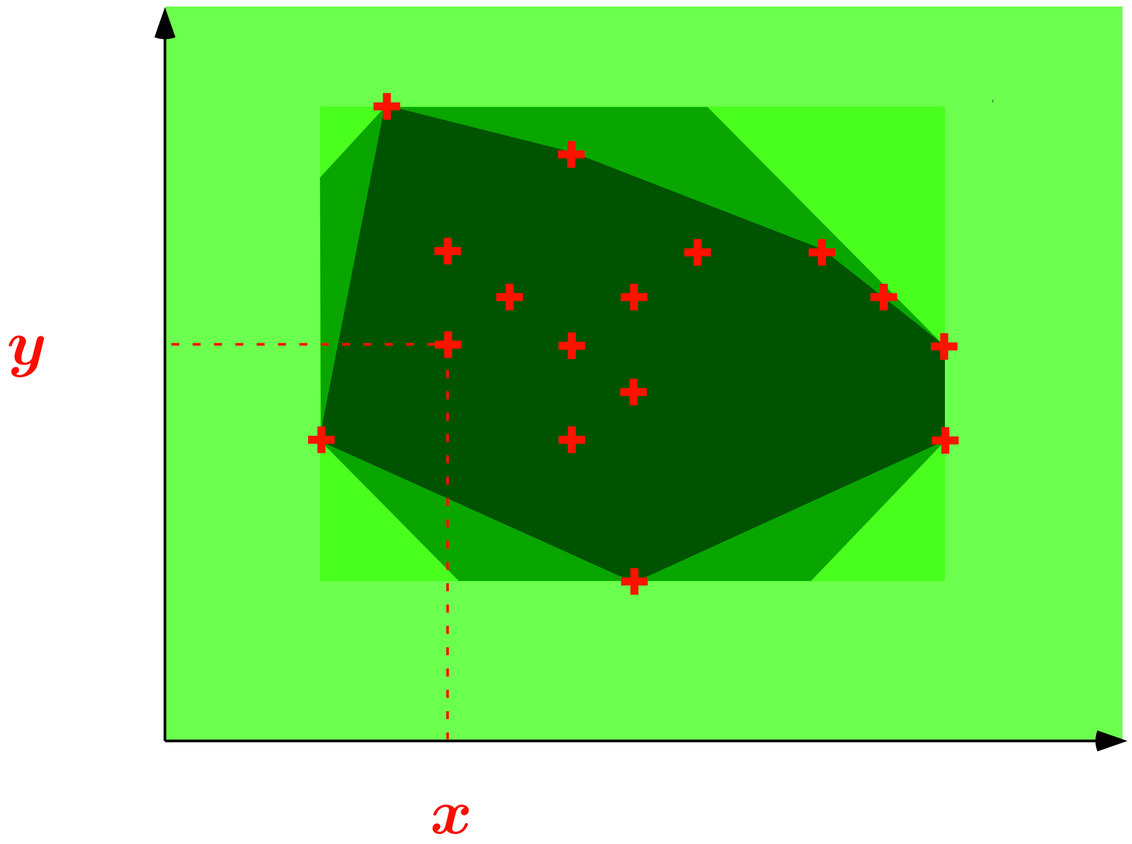

$$\begin{cases} x \in [19,\ 88] \\ y \in [19,\ 99] \end{cases}$$

# Effective Abstractions of an [In]finite Set of Points; Example 3: Octagons



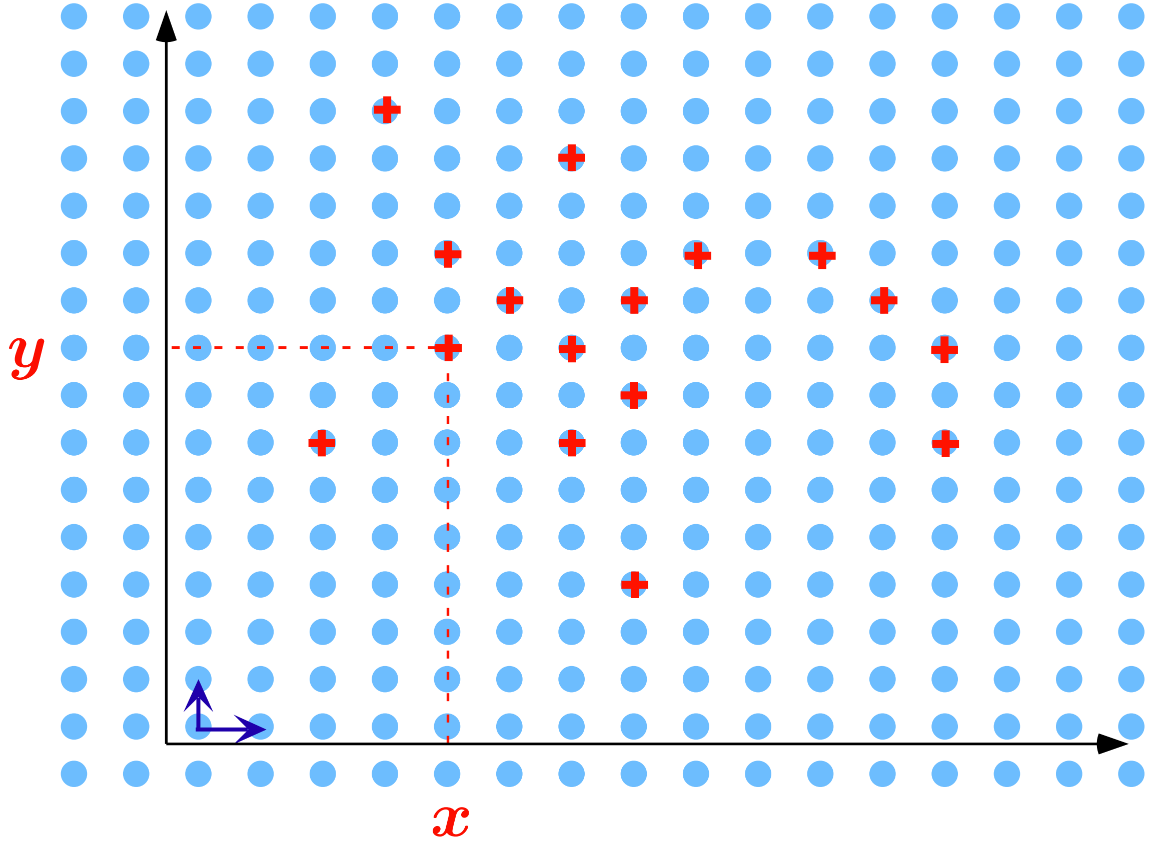

$$\begin{cases} 1 \leq x \leq 9 \\ x + y \leq 88 \\ 1 \leq y \leq 9 \\ x - y \leq 99 \end{cases}$$

# Effective Abstractions of an [In]finite Set of Points; Example 4: Polyhedra [15]



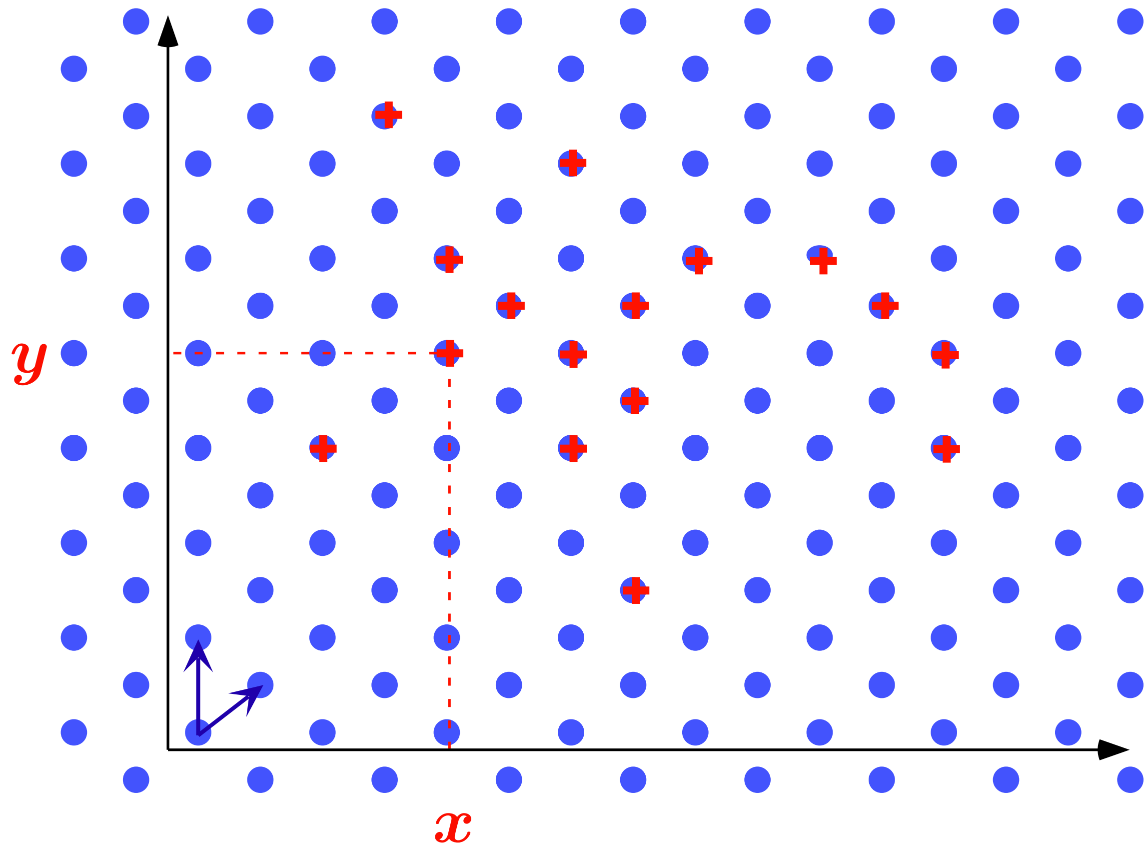

$$\begin{cases} 19x + 88y \leq 2000 \\ 19x + 99y \geq 0 \end{cases}$$

# Effective Abstractions of an [In]finite Set of Points; Example 5: Simple Congruences [17]



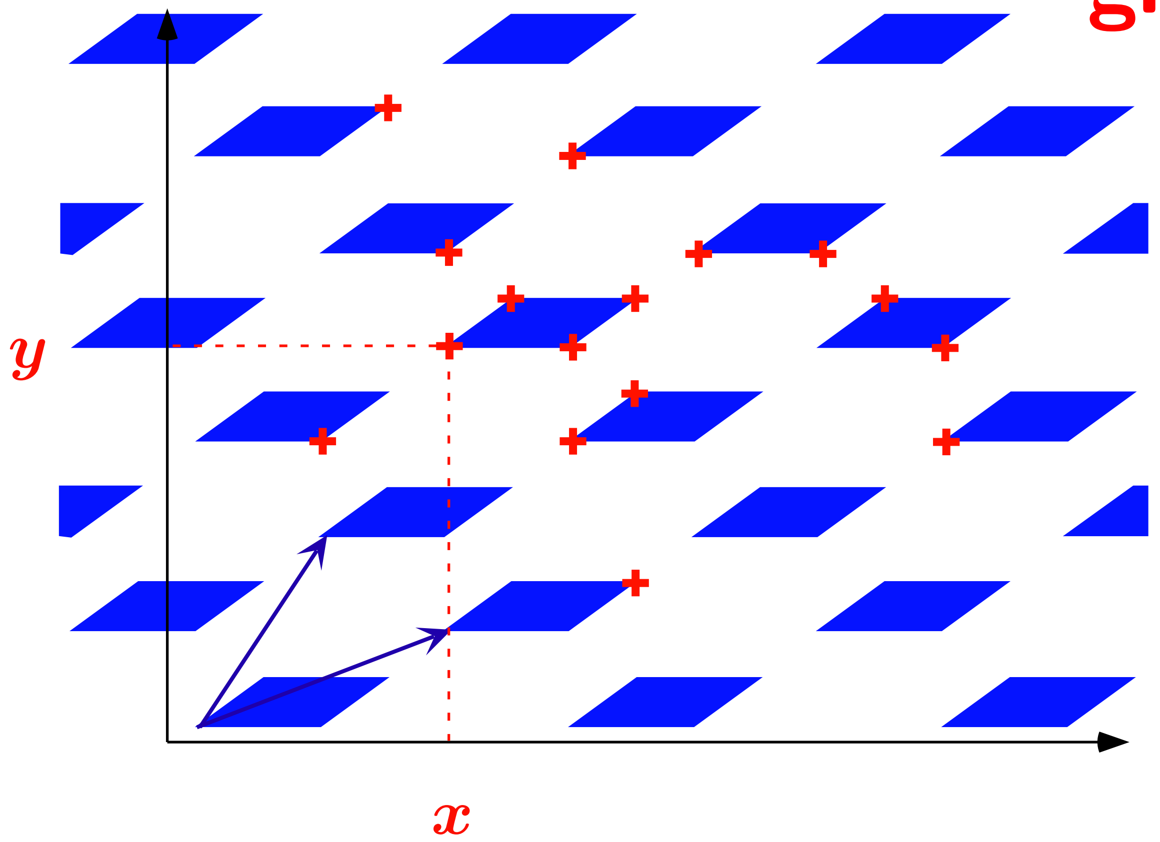

$$\begin{cases} x = 19 \bmod 88 \\ y = 19 \bmod 99 \end{cases}$$

# Effective Abstractions of an [In]finite Set of Points; Example 6: Linear Congruences [18]




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

# Effective Abstractions of an [In]finite Set of Points; Example 7: Trapezoidal Linear Congruences [20, 21]



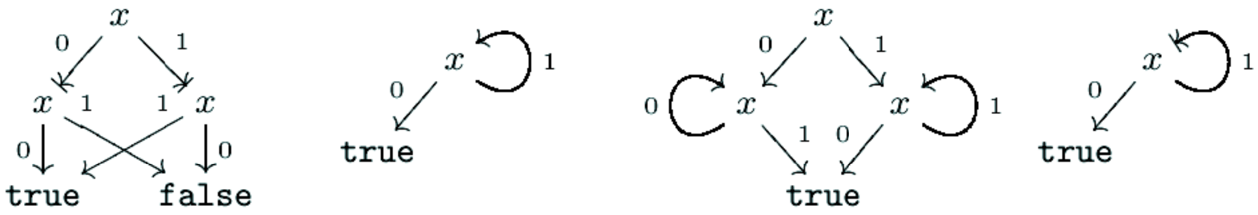

$$\begin{cases} 1x + 9y \in [0, 88] \bmod 10 \\ 1x - 9y \in [0, 99] \bmod 11 \end{cases}$$

# 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.).

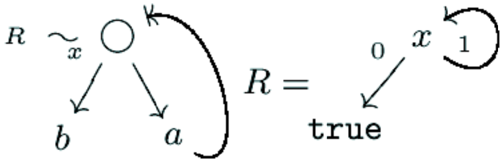
# Example of Abstractions of Infinite Sets of Infinite Trees

## Binary Decision Graphs: [22]

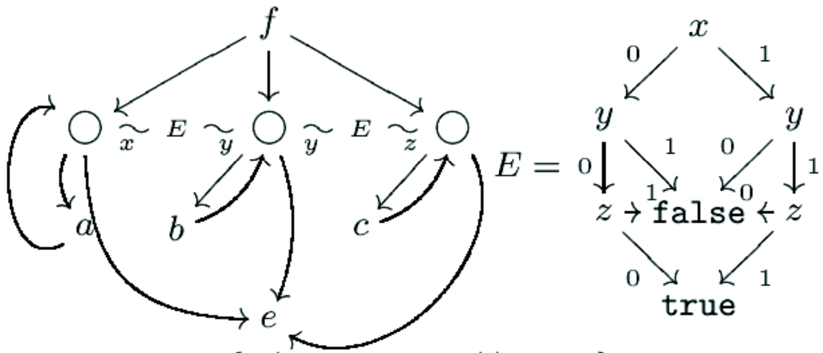



$\{0^\omega, 1^\omega\}$ — infinite number of 0's — fair vectors — ends by $0^\omega$

## Tree schemata: [24, 23]




$\{a^n b \mid n \in \mathbb{N}\}$




$\{f(a^n e, b^n e, c^n e) \mid n \in \mathbb{N}\}$

Note that $E$ is the equality relation.

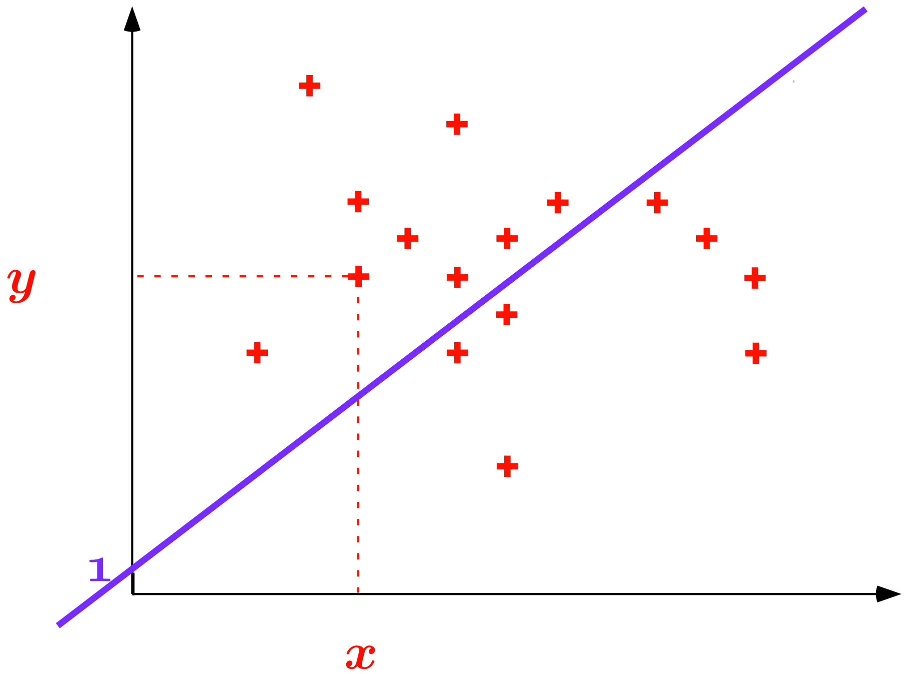
# Information Loss

# 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.

# Example of Information Loss

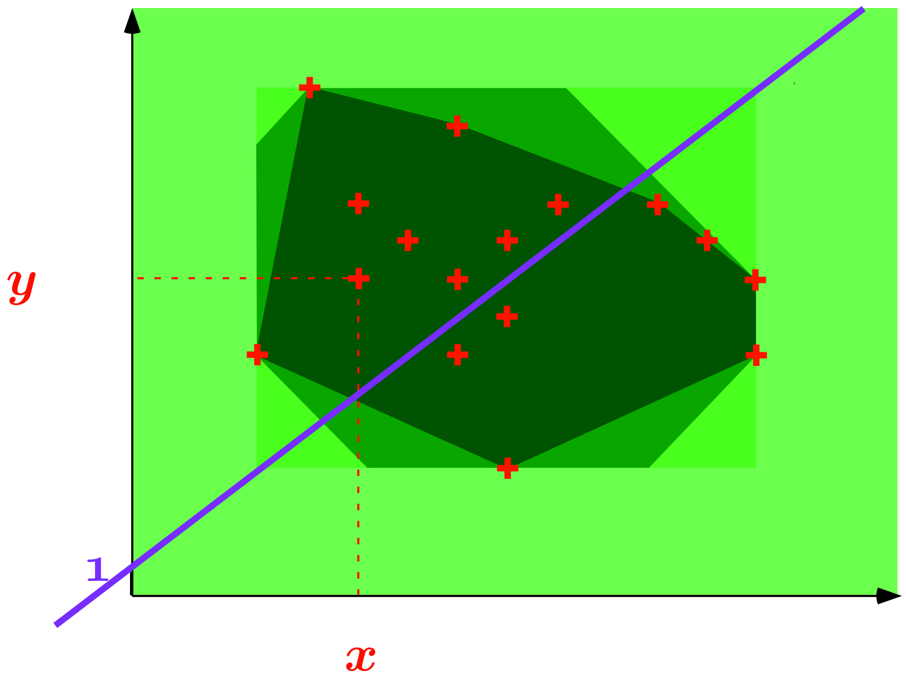
- Is the operation `1/(x+1-y)` well defined at run-time?
- Concrete semantics: **yes**



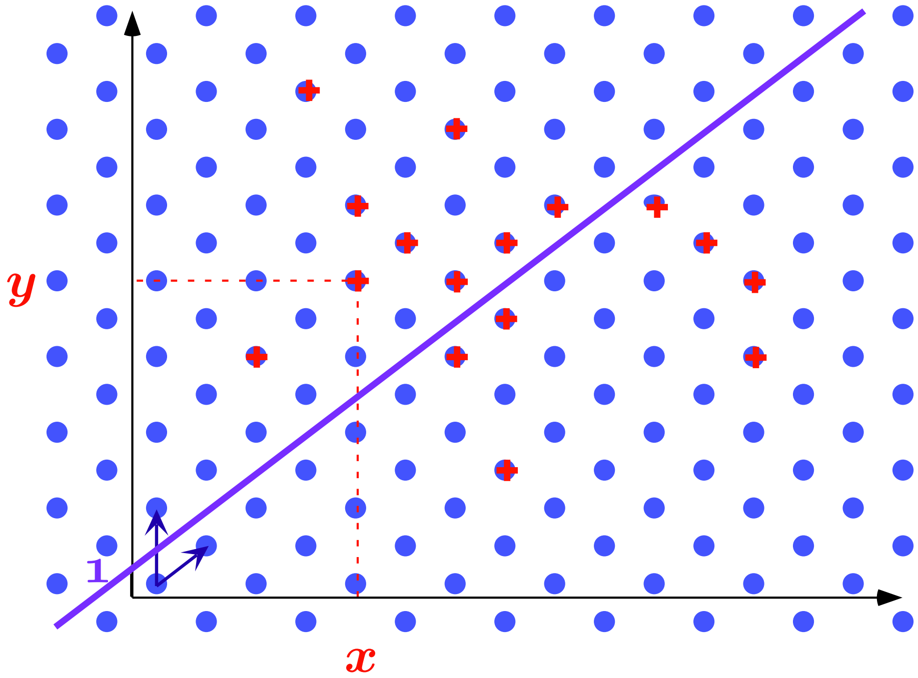
# Example of Information Loss

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



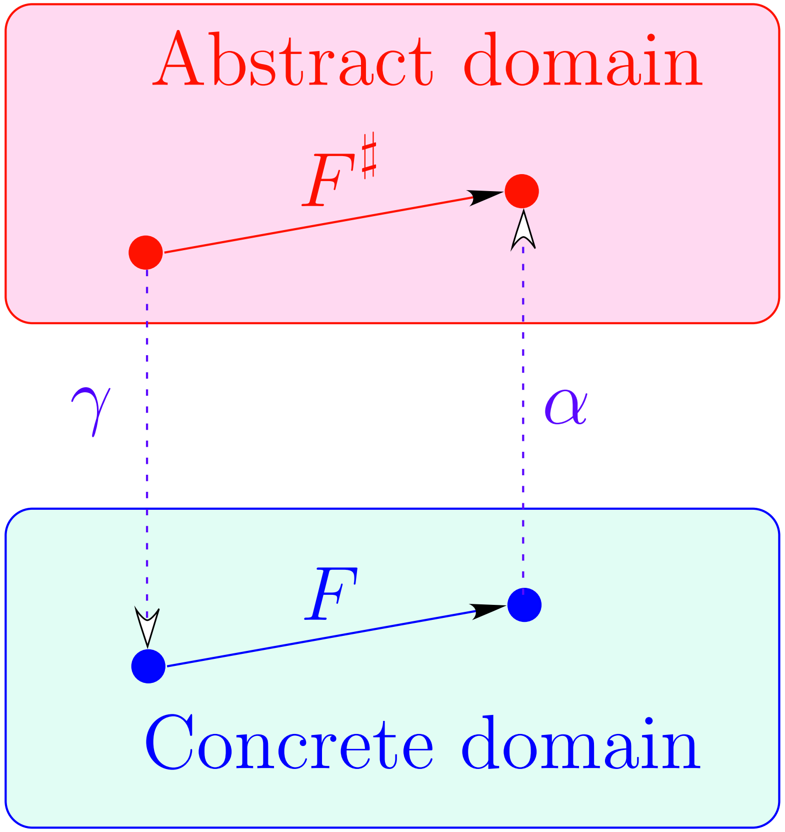
# Example of Information Loss

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



# Fixpoint Abstraction

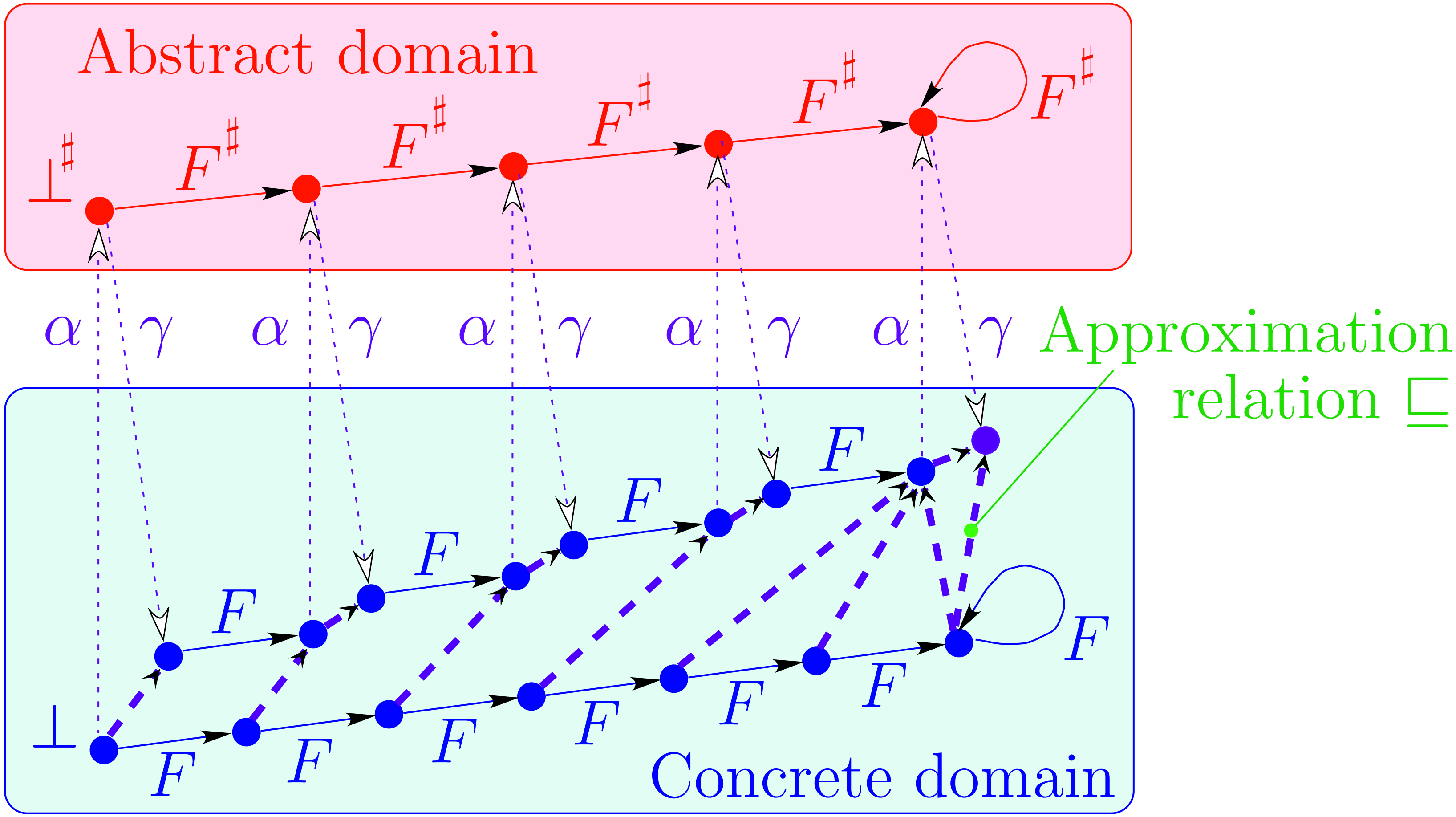
# Function Abstraction




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

# Fixpoint Abstraction



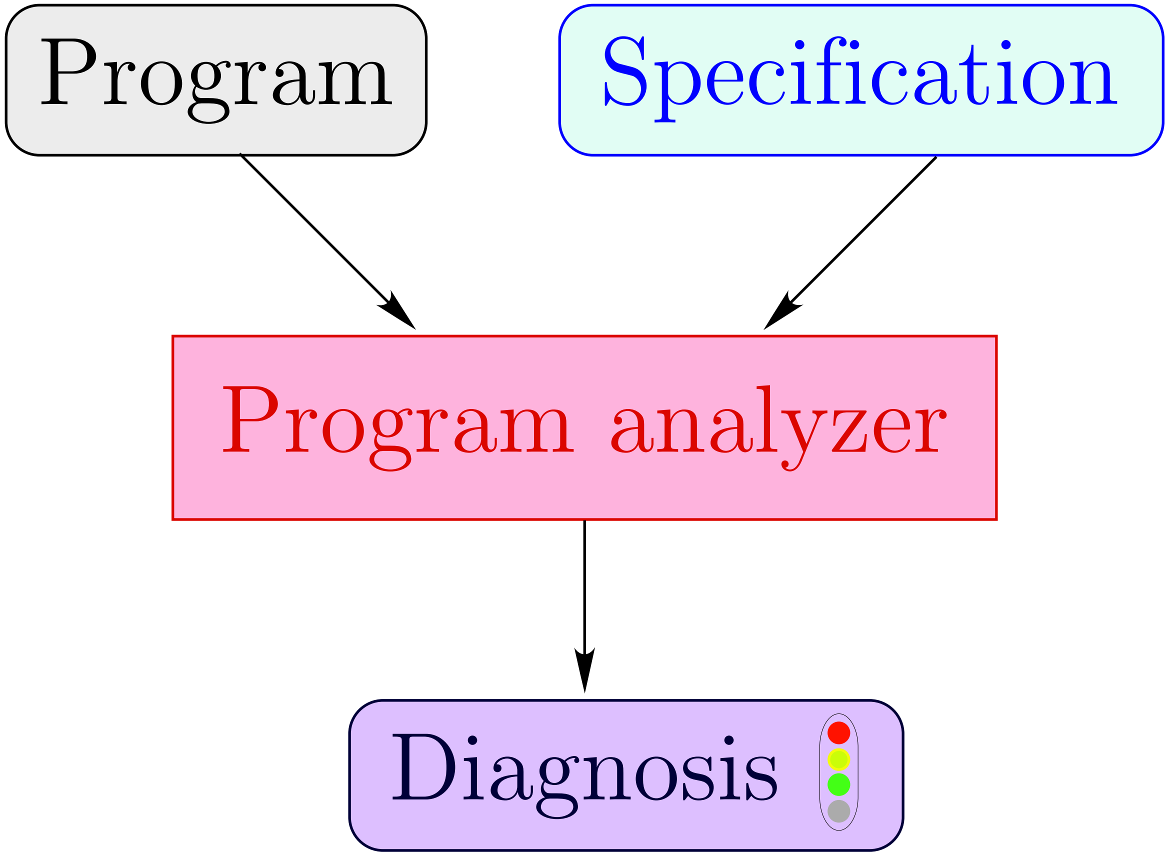

$$\mathit{lfp}\, F \sqsubseteq \gamma(\mathit{lfp}\, F^\sharp)$$

# Program Analysis

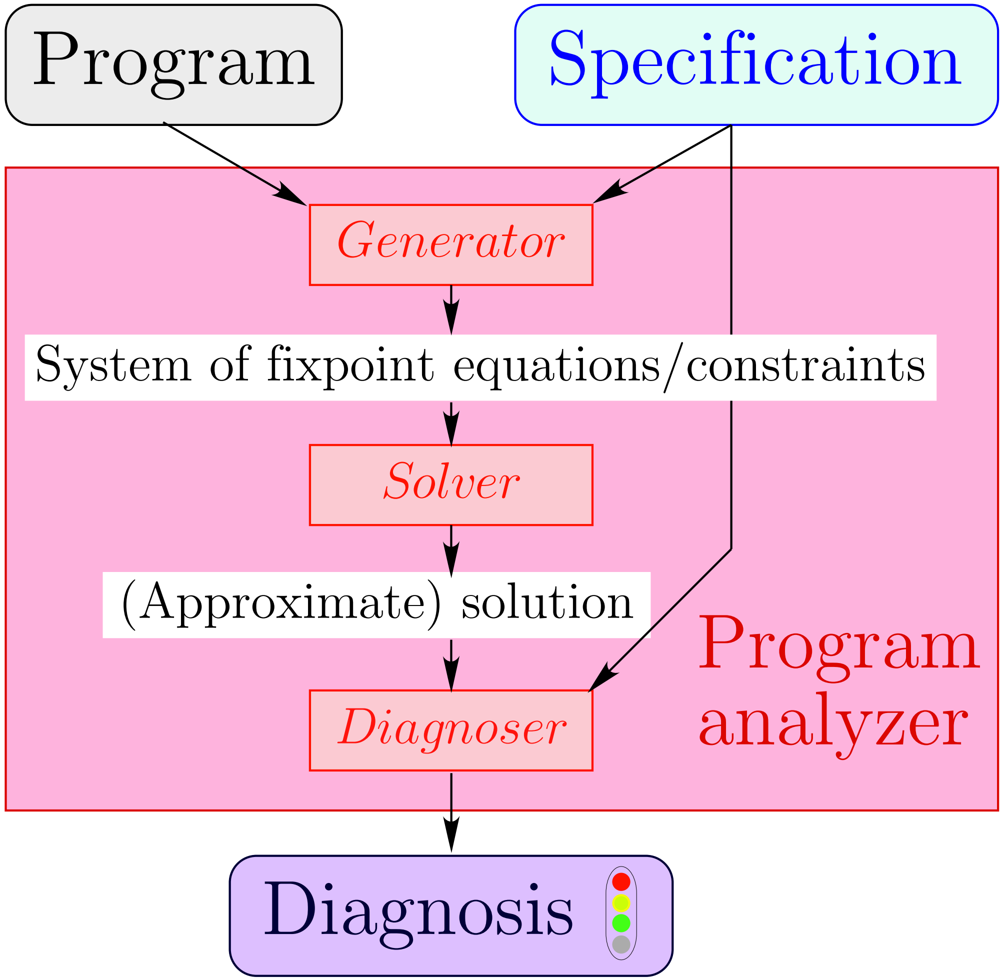
# Objective of Program Analysis

- 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.

# Objective of Program Analysis



# Principle of Program Analysis



# A Few Applications ...

- Data flow and set-based analysis for program optimization & transformation (including partial evaluation) [12 , 14];
- Type inference (including undecidable systems)/soft typing [8];
- Abstract model-checking of infinite systems [13 , 14];
- Abstract debugging & testing [5 , 2];
- Probabilistic analysis [26];
- Communication topology analysis for mobile/distributed code [28];
- Automatic differentiation of numerical programs;
- Semantic tattooing/watermarking of software; ...;

# 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 [4]; [19]
- Automatic detection of the definiteness ●, potentiality ●, impossibility ● or inaccessibility ● of run-time errors [5];
- Automatic discovery of the 501 flight error;
- Success for the 502 & 503 flights and the ARD [6].

---

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, uninitialized variables, data races on shared data structures, etc.

6 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;
- Polyspace Technologies (France), A. Deutsch & D. Pilaud, 1999.

# Abstract Formal Methods

# The Ultimate Verification Problem

- Find the last error in a software system;
- Can abstract formal methods solve the ultimate verification problem?

# Program Analysis: Shortcomings

- Can analyze large programs (220 000 lines of C) without user interaction but specifications are simple;
- Programming language semantics is very complex whence so is their abstraction;
- The abstraction hence the design of the analyzer is manual (and beyond the hability of casual programmers);
- Errors can be explained by abstract counter-examples (but hardly concrete ones);
- The 5 to 10 % cases of uncertainty must be handle with other empirical or formal methods.

# Deductive Methods: Foundational Ideas

- Use a (manually designed abstraction of the) program semantics to obtain minimal verification conditions to prove program correctness;
- Use a theorem prover or proof assistant to check the verification conditions.

# Deductive Methods: Shortcomings

- An inductive argument (e.g. invariant, variant function) has to be discovered, generally by the user;
- Only the proof verification can be (partially) automatized;
- Verification conditions sometimes unsound , essentially to make verifier simpler (e.g. modular airthmetic);
- The size of the proof is often exponential in the size of the program;
- Debugging an unsuccessful proof is as complex as (if not much more complex than) debugging the program; .../...

# Deductive Methods: Shortcomings (Cont'd)

- Interaction with the prover is hard if not despairing;
- Theorem provers are unstable over time (e.g. proof strategies get changed so that old proof no longer work);
- Uniform encoding of properties as syntactical terms/formulæ (so that e.g. BBDs are hardly efficiently encodable);
- Not good at fixpoint computation (only checking);
- No tool for mechanizing abstraction.

# Model Checking: Foundational Ideas [3, 4, 27]

- Use a model of the program (i.e. manually designed abstraction of the program semantics);
- Use a user-provided specification of the program (in a very expressive temporal logic);
- Check the specification by exhaustive search/exploration of the state space;
- Success by designing clever data structures (e.g. BDDs) and algorithms (e.g. SAT) for representing very large sets of booleans and their transformations.

# Model Checking: Shortcomings

- Does not scale up (gained only a factor of 100 in 10 years);
- The abstraction of the program semantics into a model is often manual and/or left informal;
- The model is ultimately finite (to allow for exhaustive search);
- The method is complete but the program specific abstraction is not reusable;
- Most often used as debugging rather than a verification tool;

# Typing: Foundational Ideas [16, 25]

- Consider decidable analyses only, by restricting both on specifications (allowed types) and on programs;
- Clean presentation of the type analysis (inference algorithm) through an equivalent logical formal system (type verification);
- Extended to complex data structures, polymorphism, exceptions and separate modules in a way that scales up for large programs;
- Integrated in the compiler , the certification can go down to the generated code (proof-carrying code, certified compiler);

# Typing: Shortcomings

- Type system (e.g. with subtle subtyping) can be very complex to understand for the casual user;
- Compositional but not fully abstract (same polymorphic code types differently in different contexts);
- Crude interaction with the user (no hint is given to understand why wrong programs do not type well, difficult for the user to provide hints to help the typing process);
- Considered programs are both complex (higher-order) and too restricted (mainly functional languages);

# Typing: Shortcomings (Cont'd)

- Severe restrictions on considered properties (arithmetic, out of range, null pointer dereferencing, ... errors are checked at run-time, all liveness properties are ignored);
- Encoding of types as terms/formulæ and one iterate fixpoint approximation make generalization to more expressive properties very difficult;
- The logical specification of the type system is often inexistent in the reference manual, not equivalent to the type inference algorithm or so inextricable that it is useless both to the programmer and compiler designer.

**No single formal method can ultimately solve the verification problem.**

# Current Trend: Combine Formal Methods

- **User designed abstraction:** derive a program finite abtract model by abstract interpretation , prove the correctness of the abstraction by deductive methods , later verify the abstract model by model-checking;
- **Fundamental limitation [1]:** 1°) abstraction discovery and 2°) abstract semantics derivation is as difficult as doing the proof! (resp. 1°) invariant discovery & 2°) invariant verification)

**Reference**

[1] P. Cousot. Partial completeness of abstract fixpoint checking, invited paper. In B.Y. Choueiry and T. Walsh, eds, *Proc. 4th Int. Symp. on Abstraction, Reformulations and Approximation, SARA '2000*, Horseshoe Bay, TX, USA, LNAI 1864, pp. 1–25. Springer-Verlag, 26–29 July 2000.

**No combination of formal methods can ultimately solve the verification problem either.**

# Possible Alternative: Combine Empirical and Formal Methods

# Example: 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. |

# Conclusions and Challenges

# Conclusions

- Full program verification by formal methods (model checking/deductive methods) is very costly since it ultimately requires user interaction hence is not widely applicable;
- Abstraction is mandatory for program verification but difficult, hardly automatizable and beyond the common capabilities of most programmers;
- Program analysis is cost-effective [7] since no user intervention is mandatory and universal abstractions are reusable hence commercializable;

[7] Less than 0.25$ per program line costing 50 to 80$.

# Conclusion (Cont'd)

- For large and complex programs, complete verification by formal methods is not viable at low cost;
- Program debugging is still the prominent industrial program "verification" method;
- In this context, abstract interpretation based program static analysis can be extended to abstract program testing;
- Abstract interpretation methods offer powerful techniques which, in the presence of approximation , can be viable alternatives to both the exhaustive search of model-checking and the partial exploration methods of classical debugging.

# Grand Challenge for Computer Scientists

**Software reliability** [8]

8 other suggestions were "trustworthiness" (C. Jones) and "robustness" (R. Leino).

# Challenges for Abstract Interpretation

- Large scale industrialization;
- Fundamental research:
  - Cost-effective & expressive abstractions:
    * Floating point numbers,
    * Dependence analyses,
    * Liveness properties with fairness (extending finite-state model-checking),
    * Probabilistic analyses,
    * ...;

# Challenges for Abstract Interpretation (Cont'd)

- Fundamental research (cont'd):
  - Higher-order compositional modular analyses;
  - (Automatic) combination/refinement of abstractions;
  - Interaction with users, other (in)formal methods, ...;
  - New programming paradigms (threads, mobile/network programming);
  - Integrate analysis by abstract interpretation in the full software development process.

# Short Bibliography

[2] F. Bourdoncle. Abstract debugging of higher-order imperative languages. In *Proc. PLDI*, pages 46–55. ACM Press, 1993.

[3] E.M. Clarke and E.A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *IBM Workshop on Logics of Programs*, LNCS 131. Springer-Verlag, May 1981.

[4] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite state concurrent systems using temporal logic specifications: A practical approach. In *10$^{th}$ POPL*, pages 117–126. ACM Press, Jan. 1983.

[5] P. Cousot. Semantic foundations of program analysis. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 10, pages 303–342. Prentice-Hall, 1981.

[6] P. Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *ENTCS*, 6, 1997. URL: http://www.elsevier.nl/locate/entcs/volume6.html, 25 pages.

[7] P. Cousot. Design of semantics by abstract interpretation, invited address. In *Mathematical Foundations of Programming Semantics, 30$^{th}$ Annual Conf. (MFPS XIII)*, Carnegie Mellon University, Pittsburgh, PA, US, 23–26 Mar. 1997.

[8] P. Cousot. Types as abstract interpretations, invited paper. In *24$^{th}$ POPL*, pages 316–331, Paris, FR, Jan. 1997. ACM Press.

[9] P. Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Theoret. Comput. Sci.*, To appear (Preliminary version in [6]).

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

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

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

[13] P. Cousot and R. Cousot. Refining model checking by abstract interpretation. *Aut. Soft. Eng.*, 6:69–95, 1999.

[14] P. Cousot and R. Cousot. Temporal abstract interpretation. In *27$^{th}$ POPL*, pages 12–25, Boston, MA, Jan. 2000. ACM Press.

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

[16] L. Damas and R. Milner. Principal type-schemes for functional programs. In *$9^{th}$ POPL*, pages 207–212, Albuquerque, NM, Jan. 1982. ACM Press.

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

[18] P. Granger. Static analysis of linear congruence equalities among variables of a program. In S. Abramsky and T.S.E. Maibaum, editors, *Proc. Int. J. Conf. TAPSOFT '91, Volume 1 (CAAP '91)*, Brighton, GB, LNCS 493, pages 169–192. Springer-Verlag, 1991.

[19] P. Lacan, J.N. Monfort, L.V.Q. Ribal, A. Deutsch, and G. Gonthier. The software reliability verification process: The ARIANE 5 example. In *Proceedings DASIA 98 – DAta Systems In Aerospace*, Athens, GR. ESA Publications, SP-422, 25–28 May 1998.

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

[21] F. Masdupuy. Semantic analysis of interval congruences. In D. Bjørner, M. Broy, and I.V. Pottosin, editors, *Proc. FMPA*, Academgorodok, Novosibirsk, RU, LNCS 735, pages 142–155. Springer-Verlag, 28 June – 2 Jul. 1993.

[22] L. Mauborgne. Binary decision graphs. In A. Cortesi and G. Fil , editors, *Proc. 6$^{th}$ Int. Symp. SAS '99*, Venice, IT, 22–24 Sep. 1999, LNCS 1694, pages 101–116. Springer-Verlag, 1999.

[23] L. Mauborgne. Tree schemata and fair termination. In J. Palsberg, editor, *Proc. 7$^{th}$ Int. Symp. SAS '2000*, Santa Barbara, CA, US, LNCS 1824, pages 302–321. Springer-Verlag, 29 June – 1 Jul. 2000.

[24] L. Mauborgne. Improving the representation of infinite trees to deal with sets of trees. In G. Smolka, editor, *Programming Languages and Systems, Proc.* $9^{th}$ *ESOP '2000*, Berlin, DE, LNCS 1782, pages 275–289. Springer-Verlag, Mar. – Apr. 2000.

[25] R. Milner. A theory of polymorphism in programming. *J. Comput. System Sci.*, 17(3):348–375, Dec. 1978.

[26] D. Monniaux. Abstract interpretation of probabilistic semantics. In J. Palsberg, editor, *Proc.* $7^{th}$ *Int. Symp. SAS '2000*, Santa Barbara, CA, US, LNCS 1824, pages 322–339. Springer-Verlag, 29 June – 1 Jul. 2000.

[27] J.-P. Queille and J. Sifakis. Verification of concurrent systems in CESAR. In *Proc. Int. Symp. on Programming*, LNCS 137, pages 337–351. Springer-Verlag, 1982.

[28] A. Venet. Automatic determination of communication topologies in mobile systems. In G. Levi, editor, *Proc. 5$^{th}$ Int. Symp. SAS '98*, Pisa, IT, 14–16 Sep. 1998, LNCS 1503, pages 152–167. Springer-Verlag, 1998.

# THE END