Abstract Interpretation

SAVE 2016,
Changsha, 10 December 2016

Patrick Cousot
pcousot@cs.nyu.edu cs.nyu.edu/~pcousot

This is an abstract interpretation

Scientific research

Scientific research

- In Mathematics/Physics:

  trend towards unification and synthesis through universal principles

- In Computer science:

  trend towards dispersion and parcellation through a ever-growing collection of local ad-hoc techniques for specific applications

  An exponential process, will stop!

Example: reasoning on computational structures

WCET
Axiomatic semantics
Confidentiality analysis
Program synthesis
Grammar analysis
Statistical model-checking
Invariance proof
Probabilistic verification
Parsing
Security protocole verification
Partial evaluation
Effect systems
Trace semantics
Symbolic execution
Quantum entanglement detection
Type theory
Dataflow analysis
Model checking
Obfuscation
Denotational semantics
Theories combination
Code contracts
Steganography
Systems biology analysis
Database query
Dependence analysis
CEGAR
Program transformation
Interpolants
Integrity analysis
Abstract model checking
Bisimulation
SMT solvers
Tautology testers
Operational semantics
Abstraction refinement
Type inference
Separation logic
Termination proof
Shape analysis
Malware detection
Code refactoring

Example: reasoning on computational structures

WCET
Axiomatic semantics
Confidentiality analysis
Program synthesis
Grammar analysis
Statistical model-checking
Invariance proof
Probabilistic verification
Parsing
Security protocole verification
Partial evaluation
Effect systems
Trace semantics
Symbolic execution
Quantum entanglement detection
Type theory
Dataflow analysis
Model checking
Obfuscation
Denotational semantics
Theories combination
Code contracts
Steganography
Systems biology analysis
Database query
Dependence analysis
CEGAR
Program transformation
Interpolants
Integrity analysis
Abstract model checking
Bisimulation
SMT solvers
Tautology testers
Operational semantics
Abstraction refinement
Type inference
Separation logic
Termination proof
Shape analysis
Malware detection
Code refactoring

Example: reasoning on computational structures

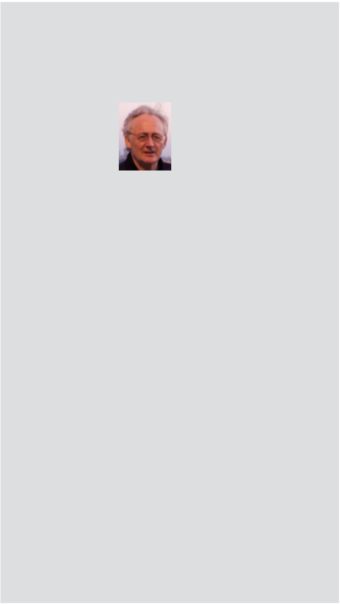
Abstract interpretation

WCET
Axiomatic semantics
Confidentiality analysis
Program synthesis
Grammar analysis
Statistical model-checking
Invariance proof
Probabilistic verification
Parsing
Security protocole verification
Partial evaluation
Effect systems
Trace semantics
Symbolic execution
Quantum entanglement detection
Type theory
Dataflow analysis
Model checking
Obfuscation
Denotational semantics
Theories combination
Code contracts
Steganography
Systems biology analysis
Database query
Dependence analysis
CEGAR
Program transformation
Interpolants
Integrity analysis
Abstract model checking
Bisimulation
SMT solvers
Tautology testers
Operational semantics
Abstraction refinement
Type inference
Separation logic
Termination proof
Shape analysis
Malware detection
Code refactoring

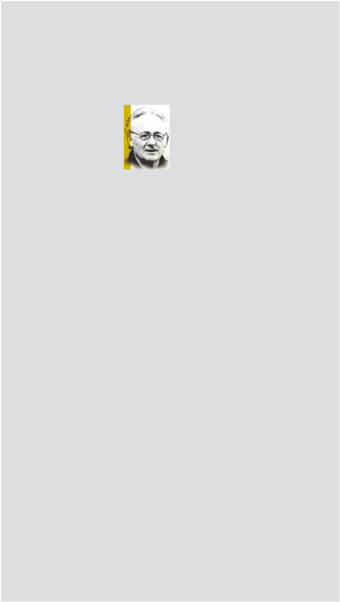
Intuition 1

Concrete

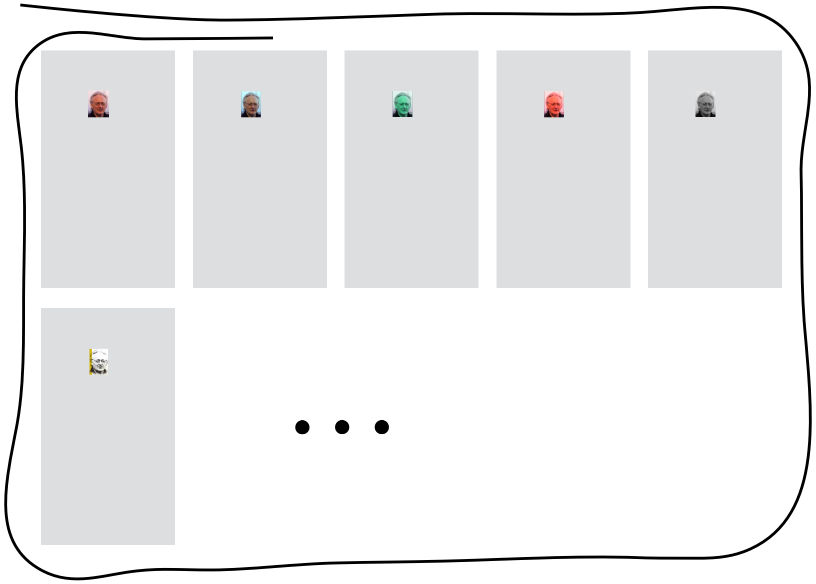
Abstraction 1



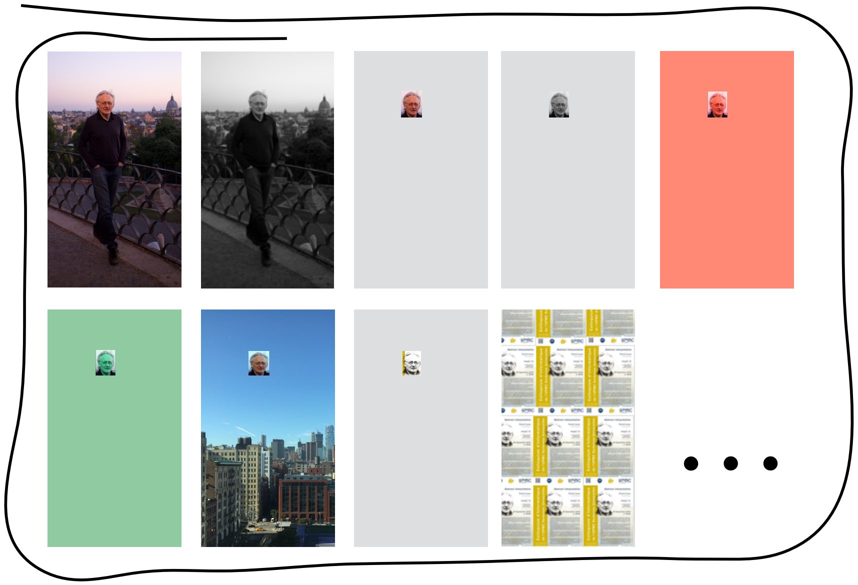
Abstraction 2



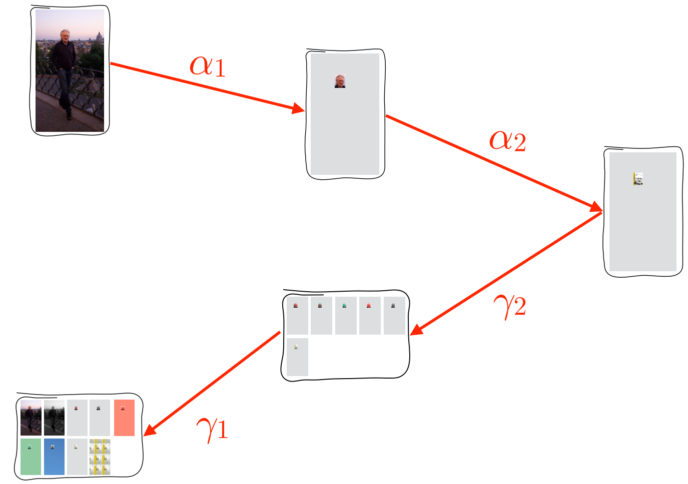
Concretization 2



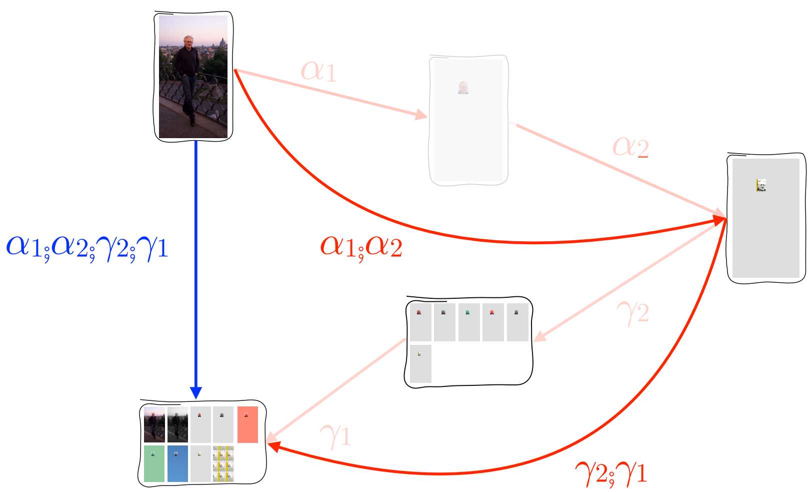
Concretization 1



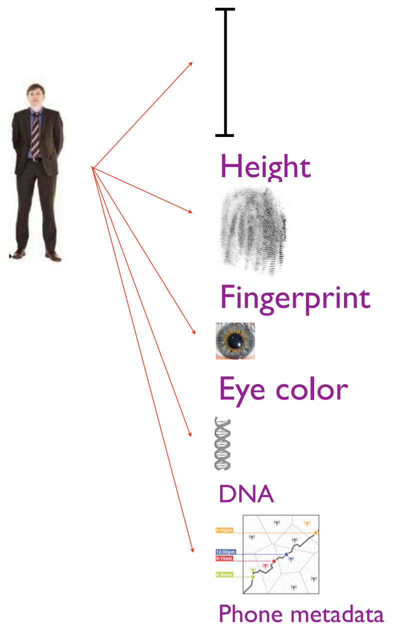
Abstract interpretations

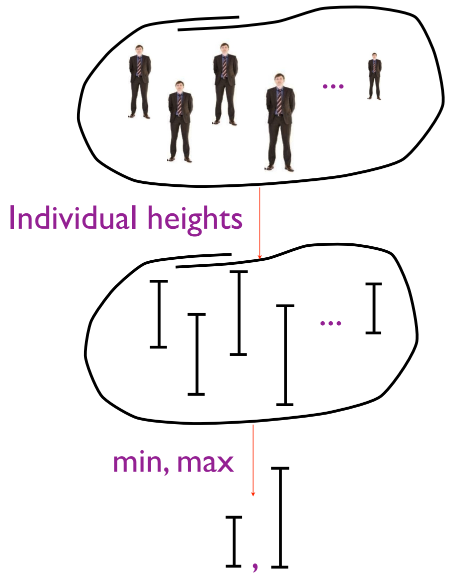


Abstract interpretations



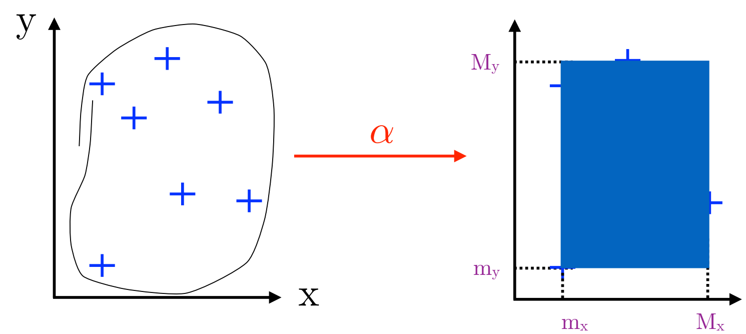
Intuition 2





Interval abstraction

- Example: interval abstraction (also called *box abstraction*)



Set of points

Interval abstraction $[m_x,M_x]\times[m_y,M_y]$

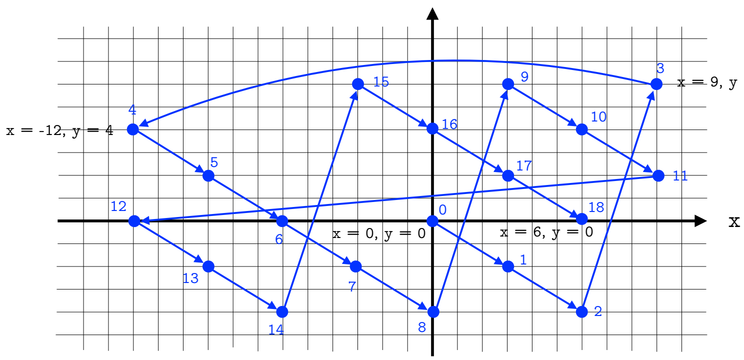
Intuition 3

A C program and one of its executions

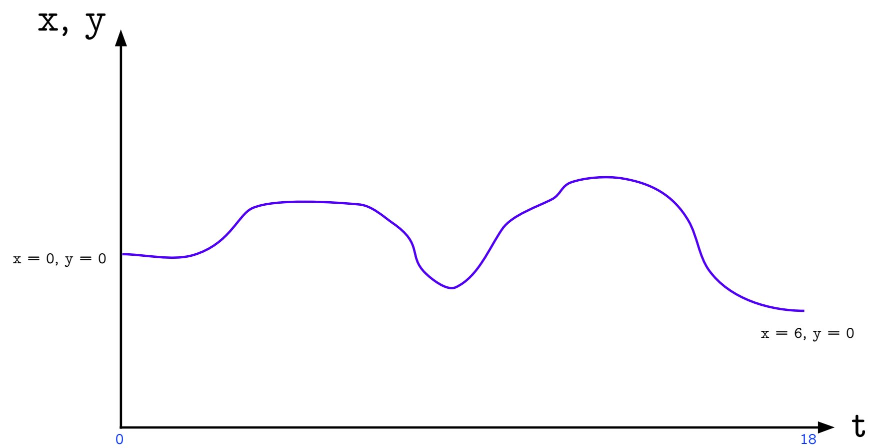
```
#include <stdio.h>
int main()
{
          int x, y;
          printf("Enter two integers: ");
          scanf("%d %d",&x, &y);
/* 1: */  while ((x != 6) || ( y != 0)) {
             printf("x = %d, y = %d\n",x,y);
/* 2: */     x = x + 3;
/* 3: */     if (x > 10) x = -x;
/* 4: */     y = y - 2;
/* 5: */     if (y < -5) y = -y;
          }
/* 6: */  printf("x = %d, y = %d\n",x,y);
}
```

```
Enter two integers: x = 0, y = 0
x = 3, y = -2
x = 6, y = -4
x = 9, y = 6
x = -12, y = 4
x = -9, y = 2
x = -6, y = 0
x = -3, y = -2
x = 0, y = -4
x = 3, y = 6
x = 6, y = 4
x = 9, y = 2
x = -12, y = 0
x = -9, y = -2
x = -6, y = -4
x = -3, y = 6
x = 0, y = 4
x = 3, y = 2
x = 6, y = 0
```

Graphical representation of the execution (I)

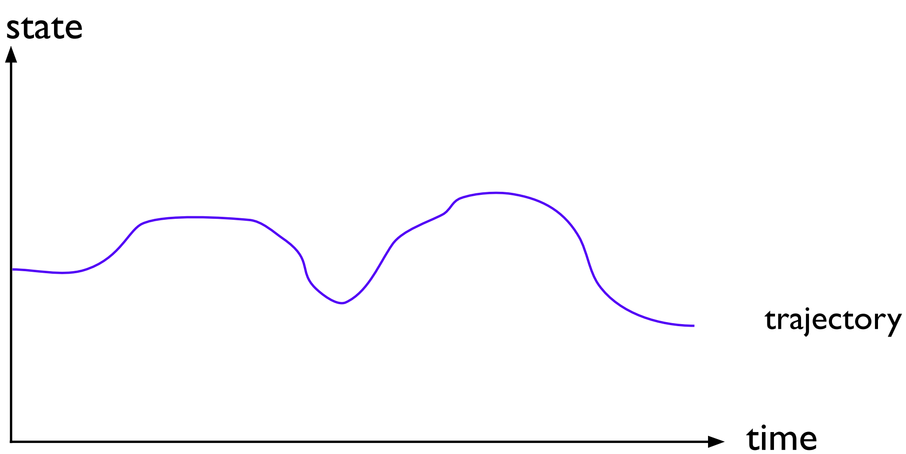


Graphical representation of the execution (2)



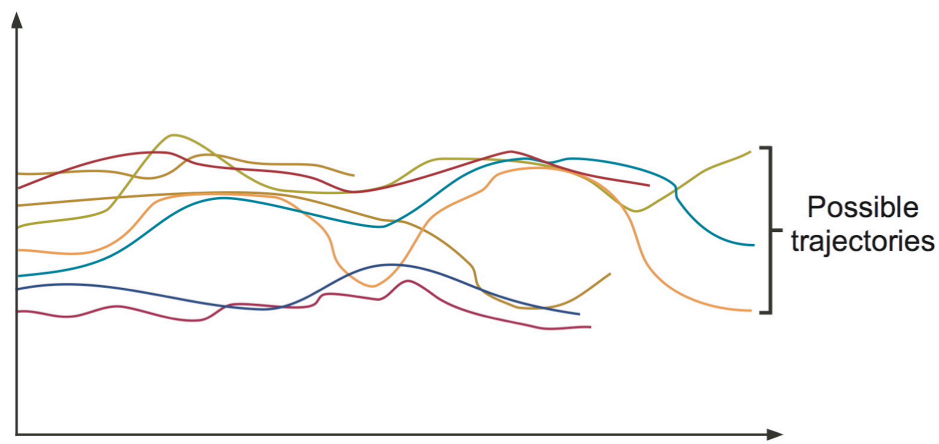
Semantics

Formalize what it means to run a program



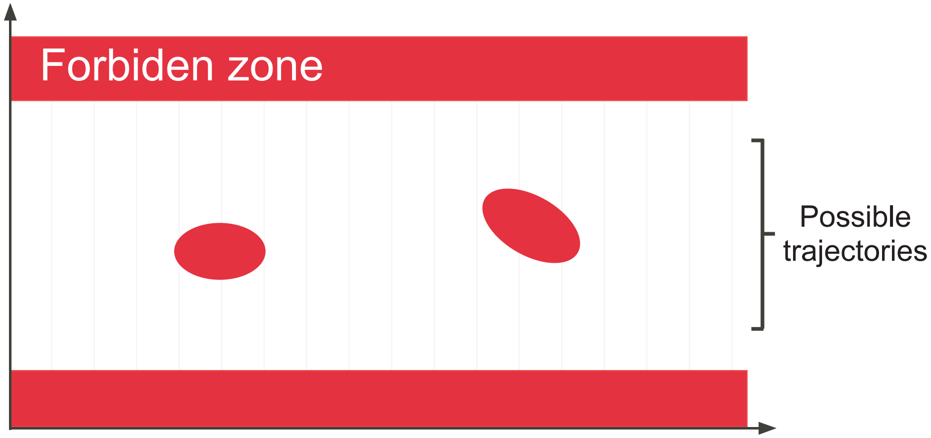
Properties (Collecting semantics)

Formalize what you are interested to ***know*** *about program behaviors*



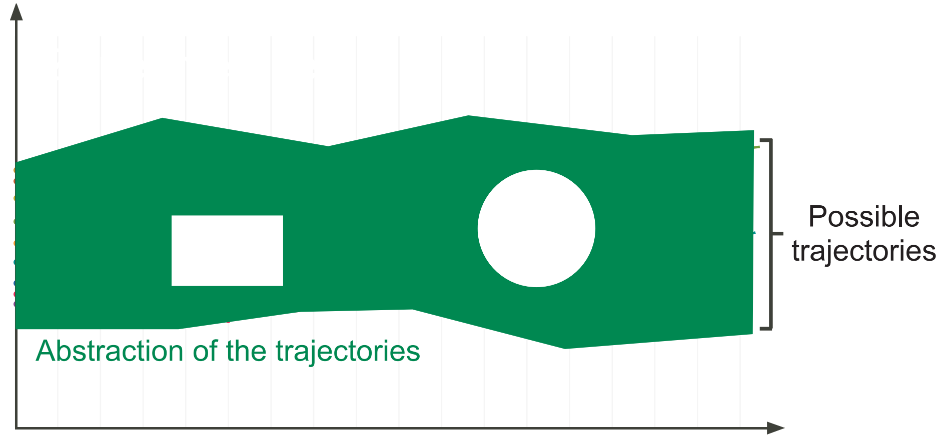
Specification

*Formalize what you are interested to **prove** about program behaviors*



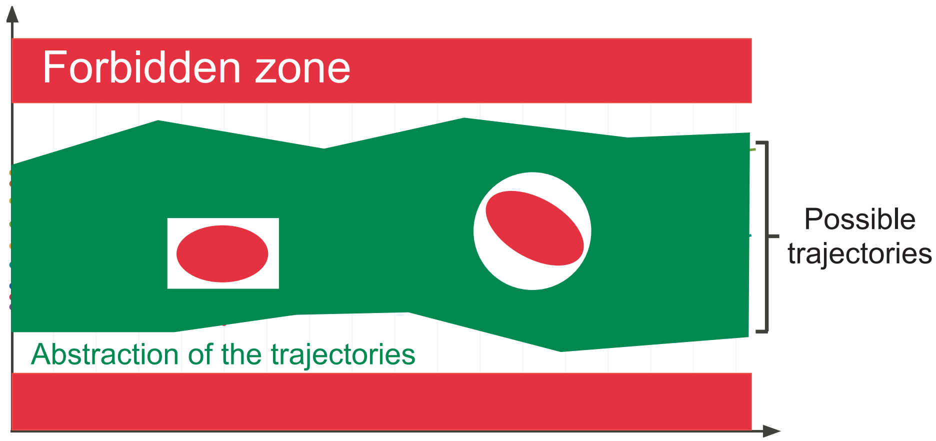
Abstraction

***Abstract** away all information on program behaviors irrelevant to the proof*



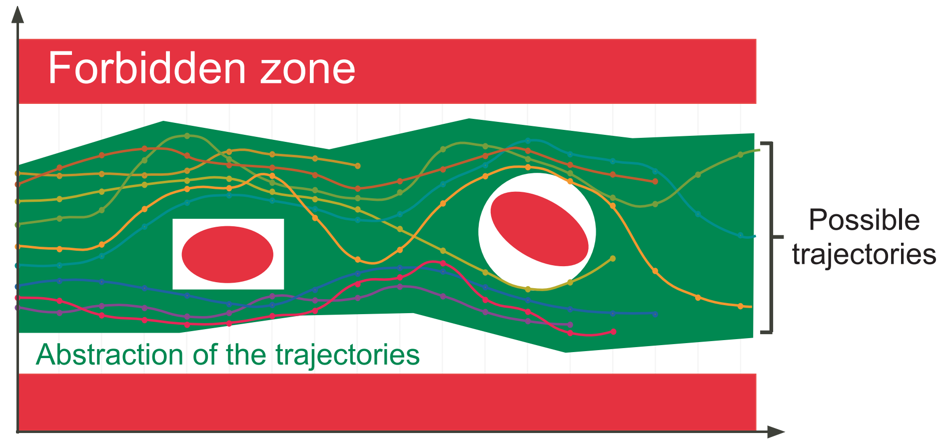
Verification

*The proof is fully **automatic***



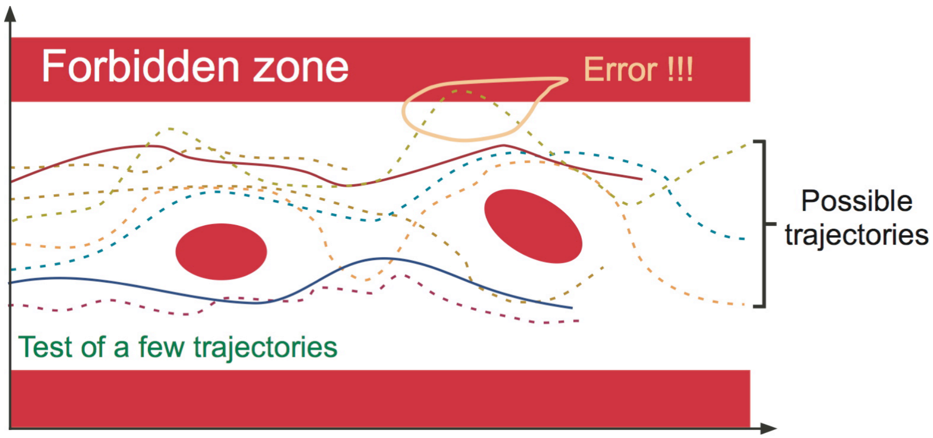
Soundness

*Never forget any possible case so the **abstract proof is correct in the concrete***



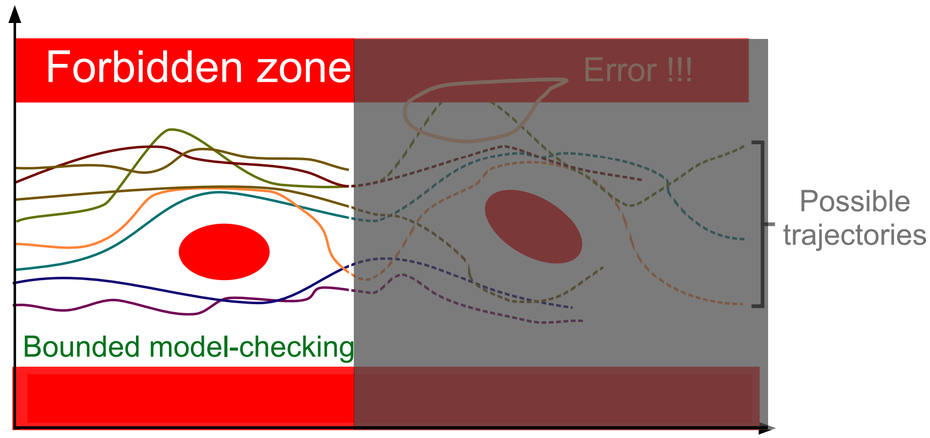
Unsound methods: testing

Try a few cases



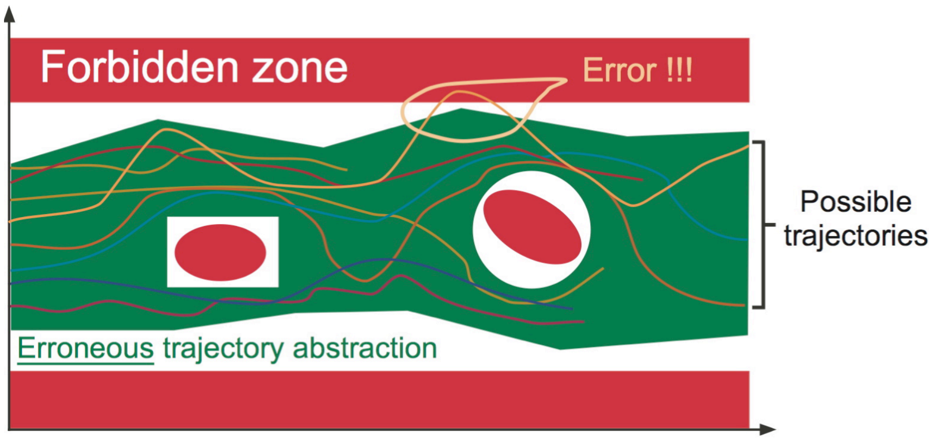
Unsound methods: bounded model checking

Simulate the beginning of all executions (so called bounded model-checking)



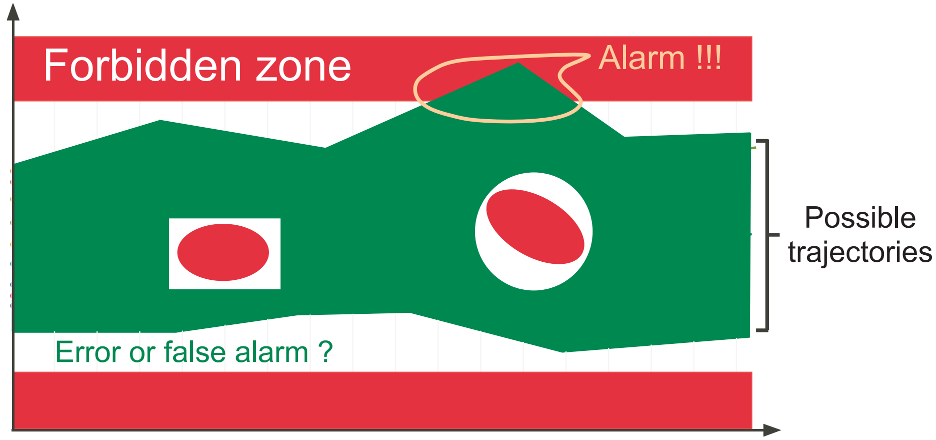
Unsound methods: soundiness

Many static analysis tools are ***unsound*** *(e.g. Coverity, etc.) so inconclusive*



Alarms

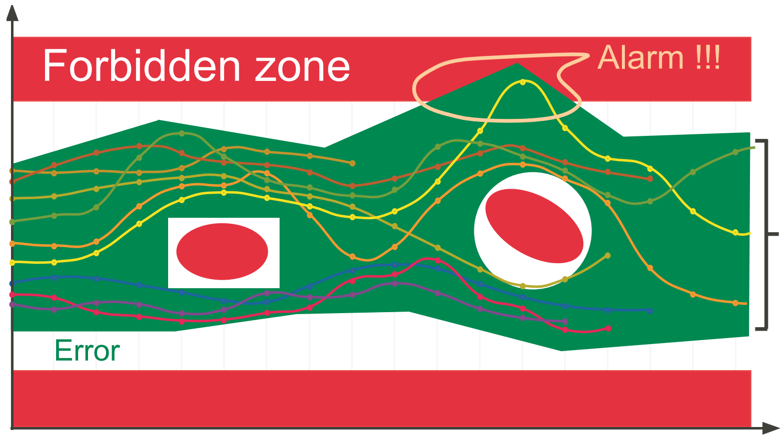
When abstract proofs may fail while concrete proofs would succeed



By soundness an alarm must be raised for this over-approximation!

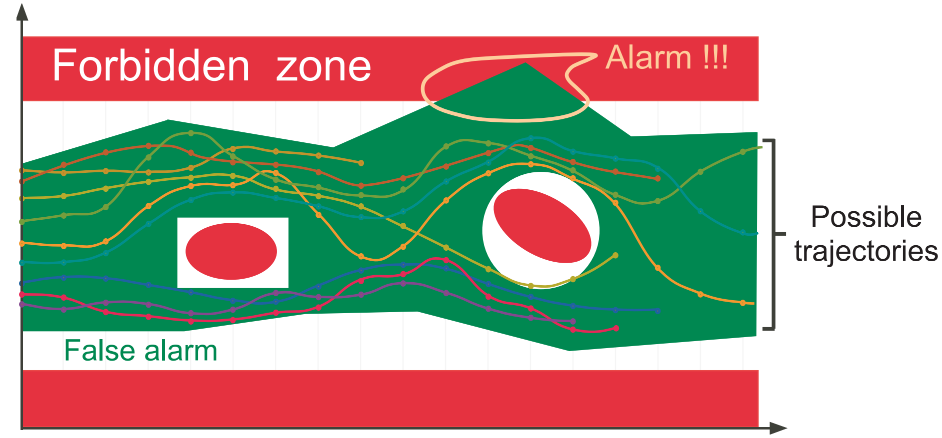
True alarm

The abstract alarm may correspond to a concrete error



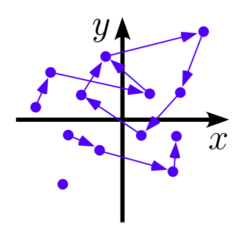
False alarm

The abstract alarm may correspond to no concrete error (false negative)

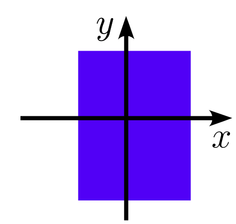


What to do in presence of false alarms

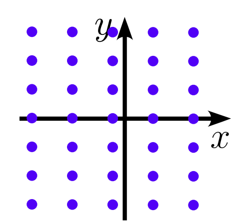
- False alarms are ultimately unavoidable (Gödel's incompleteness)
- Consider finite cases or decidable cases only (model-checking, *does not scale*)
- Ask for human help by providing information on the program behavior (theorem provers, SMT solvers), *program specific and labor costly*
- Have specialists refine the abstract interpretation (e.g. Astrée, http://www.absint.com/astree/index.htm), *shared cost*



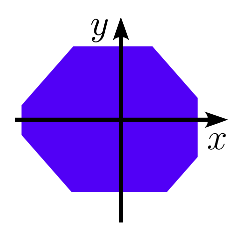
Collecting semantics: partial traces



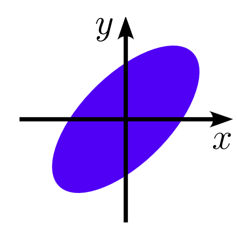
Intervals: $x \in [a, b]$



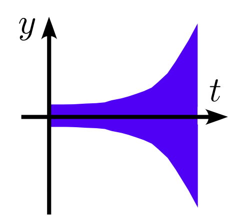
Simple congruences: $x \equiv a[b]$



Octagons: $\pm x \pm y \leqslant a$



Ellipses: $x^2 + by^2 - axy \leqslant d$



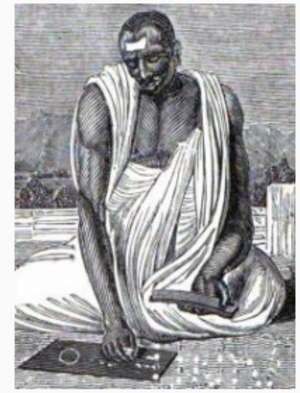
Exponentials: $-a^{bt} \leqslant y(t) \leqslant a^{bt}$

The very first static analysis

Brahmagupta

Brahmagupta (Sanskrit: ब्रह्मगुप्त; (598–c.670 CE) was an Indian mathematician and astronomer who wrote two important works on Mathematics and Astronomy: the *Brāhmasphuṭasiddhānta* (Extensive Treatise of Brahma) (628), a theoretical treatise, and the *Khaṇḍakhādyaka*, a more practical text.

Brahmagupta



Born	598 CE
Died	c.670 CE
Fields	Mathematics, Astronomy
Known for	Zero, modern Number system

The rule of signs by Brahmagupta (628)

18.30. [The sum] of two positives is positives, of two negatives negative;

The rule of signs by Brahmagupta (628)

18.30. [The sum] of two positives is positives, of two negatives negative;

- The abstraction is that you do not (always) need to known the absolute value of the arguments to know the sign of the result;

The rule of signs by Brahmagupta (628)

18.30. [The sum] of two positives is positives, of two negatives negative;

- The abstraction is that you do not (always) need to known the absolute value of the arguments to know the sign of the result;
- Sometimes imprecise (don't know the sign of the sum of a positive and a negative)

The rule of signs by Brahmagupta (628)

18.30. [The sum] of two positives is positives, of two negatives negative;

- The abstraction is that you do not (always) need to known the absolute value of the arguments to know the sign of the result;
- Sometimes imprecise (don't know the sign of the sum of a positive and a negative)
- Useful in practice (if you know what to do when you don't know the sign)

The rule of signs by Brahmagupta (628)

18.30. [The sum] of two positives is positives, of two negatives negative;

- The abstraction is that you do not (always) need to known the absolute value of the arguments to know the sign of the result;
- Sometimes imprecise (don't know the sign of the sum of a positive and a negative)
- Useful in practice (if you know what to do when you don't know the sign)
- e.g. in compilation: do not optimize (a division by 2 into a shift when positive(*))

(*) Unless processor uses 2's complement and can shift the sign.

The rule of signs by Brahmagupta (628)

18.30. [The sum] of two positives is positives, of two negatives negative;

[...]

18.32. A negative minus zero is negative, a positive [minus zero] positive; zero [minus zero] is zero. When a positive is to be subtracted from a negative or a negative from a positive, then it is to be added.

The rule of signs by Brahmagupta (628)

18.30. [The sum] of two positives is positives, of two negatives negative; [...]

18.32. A negative minus zero is negative, a positive [minus zero] positive; zero [minus zero] is zero. When a positive is to be subtracted from a negative or a negative from a positive, then it is to be added.

18.33. The product of a negative and a positive is negative, of two negatives positive, and of positives positive; the product of zero and a negative, of zero and a positive, or of two zeros is zero.

The rule of signs by Brahmagupta (628)

18.30. [The sum] of two positives is positives, of two negatives negative; [...]

18.32. A negative minus zero is negative, a positive [minus zero] positive; zero [minus zero] is zero. When a positive is to be subtracted from a negative or a negative from a positive, then it is to be added.

18.33. The product of a negative and a positive is negative, of two negatives positive, and of positives positive; the product of zero and a negative, of zero and a positive, or of two zeros is zero.

18.34. A positive divided by a positive or a negative divided by a negative is positive; a zero divided by a zero is zero; a positive divided by a negative is negative; a negative divided by a positive is [also] negative.

wrong

The rule of signs by Michel Sintzoff (1972)

For example, a×a+b×b yields the value 25 when a is 3 and b is -4, and when + and × are the arithmetic multiplication and addition. But a×a+b×b yields always the object "pos" when a and b are the objects "pos" or "neg", and when the valuation is defined as follows :

```
pos+pos=pos              pos×pos=pos
pos+neg=pos,neg          pos×neg=neg
neg+pos=pos,neg          neg×pos=neg
neg+neg=neg              neg×neg=pos
V(p+q)=V(p)+V(q)         V(p×q)=V(p)×V(q)
V(0)=V(1)=...=pos
V(-1)=V(-2)=...=neg
```

The valuation of a×a+b×b yields "pos" by the following computations :

```
V(a)=pos,neg                V(b)=pos,neg
V(a×a)=pos×pos,neg×neg      V(b×b)=pos×pos,neg×neg
      =pos,pos=pos                =pos,pos=pos
V(a×a+b×b)=V(a×a)+V(b×b)=pos+pos=pos
```

This valuation proves that the result of a×a+b×b is always positive and hence allows to compute its square root without any preliminary dynamic test on its sign. On the other hand, the

The rule of signs by Michel Sintzoff (1972)

For example, a×a+b×b yields the value 25 when a is 3 and b is -4, and when + and × are the arithmetic multiplication and addition. But a×a+b×b yields always the object "pos" when a and b are the objects "pos" or "neg", and when the valuation is defined as follows :

```
pos+pos=pos              pos×pos=pos
pos+neg=pos,neg          pos×neg=neg
neg+pos=pos,neg          neg×pos=neg
neg+neg=neg              neg×neg=pos
V(p+q)=V(p)+V(q)         V(p×q)=V(p)×V(q)
V(0)=V(1)=...=pos
V(-1)=V(-2)=...=neg
```

The valuation of a×a+b×b yields "pos" by the following computations :

```
V(a)=pos,neg                V(b)=pos,neg
V(a×a)=pos×pos,neg×neg      V(b×b)=pos×pos,neg×neg
      =pos,pos=pos                =pos,pos=pos
V(a×a+b×b)=V(a×a)+V(b×b)=pos+pos=pos
```

This valuation proves that the result of a×a+b×b is always positive and hence allows to compute its square root without any preliminary dynamic test on its sign. On the other hand, the

The rule of signs by Michel Sintzoff (1972)

For example, a×a+b×b yields the value 25 when a is 3 and b is -4, and when + and × are the arithmetic multiplication and addition. But a×a+b×b yields always the object "pos" when a and b are the objects "pos" or "neg", and when the valuation is defined as follows :

```
pos+pos=pos                 pos×pos=pos
pos+neg=pos,neg             pos×neg=neg
neg+pos=pos,neg             neg×pos=neg
neg+neg=neg                 neg×neg=pos
V(p+q)=V(p)+V(q)            V(p×q)=V(p)×V(q)
V(0)=V(1)=...=pos
V(-1)=V(-2)=...=neg
```

The valuation of a×a+b×b yields "pos" by the following computations :

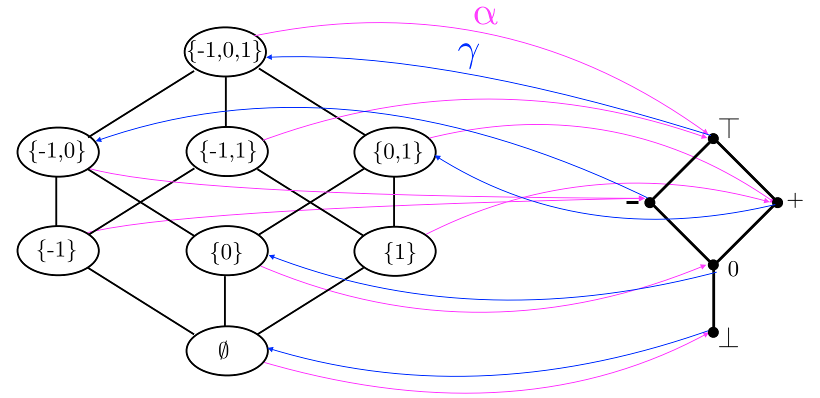
```
V(a)=pos,neg                V(b)=pos,neg
V(a×a)=pos×pos,neg×neg      V(b×b)=pos×pos,neg×neg
     =pos,pos=pos                =pos,pos=pos
V(a×a+b×b)=V(a×a)+V(b×b)=pos+pos=pos
```

This valuation proves that the result of a×a+b×b is always positive and hence allows to compute its square root without any preliminary dynamic test on its sign. On the other hand, the

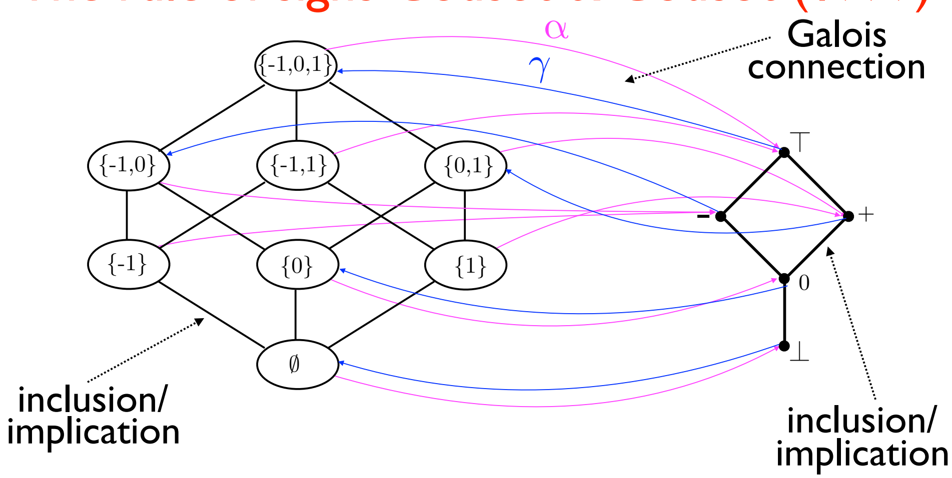
wrong

$0 \in pos \times -1 \in neg$
$= 0 \notin neg$

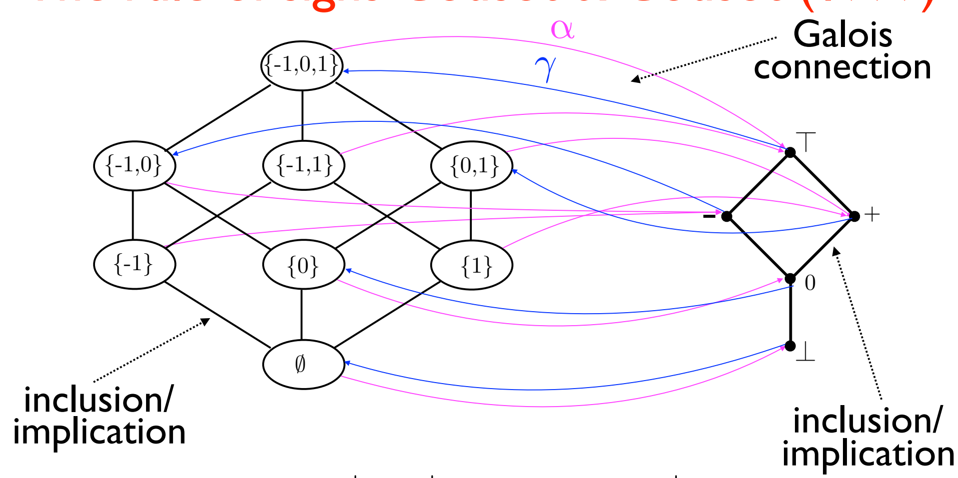
The rule of signs Cousot & Cousot (1979)



The rule of signs Cousot & Cousot (1979)



The rule of signs Cousot & Cousot (1979)



calculational design method

$+ \; + \; + \quad = \quad +$

$\gamma \downarrow \quad \downarrow \gamma \quad \uparrow \alpha$

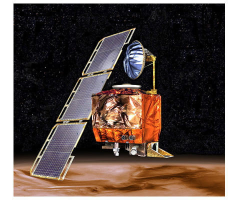
$\{0,1\} + \{0,1\} = \{0,1,2[2]\} = \{0,1\}$

Application of abstract interpretation to static analysis

All computer scientists have experienced bugs

Ariane 5.01 failure (overflow)

Patriot failure (float rounding)



Mars orbiter loss (unit error)

Heartbleed (buffer overrun)

- Checking the presence of bugs by debugging is great
- Proving their absence by static analysis is even better!

Static analysis

- Check program properties (automatically, using the program text only, without running the program)
- Difficulties:
  - Undecidability / complexity:
    - Precision
    - Scalability
  - Soundness (correctness)
  - Induction: widening/narrowing

Fixpoint

Fixpoint equation

$\{y \geqslant 0\}$ ← hypothesis
```
x = y
```
$\{I(x,y)\}$ ← loop invariant
```
while (x > 0) {
  x = x - 1;
}
```

Floyd-Naur-Hoare verification conditions:

$$(y \geqslant 0 \wedge x = y) \Longrightarrow I(x,y) \quad \textit{initialisation}$$
$$(I(x,y) \wedge x > 0 \wedge x' = x - 1) \Longrightarrow I(x',y) \quad \textit{iteration}$$

Equivalent fixpoint equation:

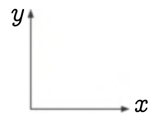
$$I(x,y) = x \geqslant 0 \wedge (x = y \vee I(x+1,y)) \qquad (\text{i.e. } I = F(I)^{(5)})$$

(5) We look for the most precise invariant I, implying all others, that is $\mathsf{lfp}^{\Rightarrow} F$.

Iterates

$$\text{Iterates } I = \lim_{n \to \infty} F^n(\text{false})$$

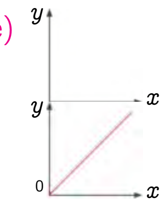
$$I^0(x, y) = \text{false}$$



Iterates

$$\text{Iterates } I = \lim_{n \to \infty} F^n(\text{false})$$

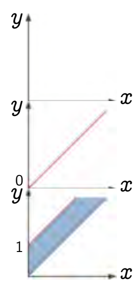
$$\begin{aligned} I^0(x, y) &= \text{false} \\ I^1(x, y) &= x \geqslant 0 \wedge (x = y \vee I^0(x+1, y)) \\ &= 0 \leqslant x = y \end{aligned}$$



Iterates

$$\text{Iterates } I = \lim_{n \to \infty} F^n(\text{false})$$

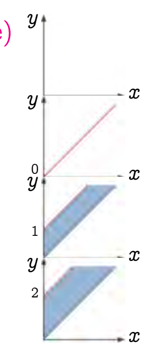
$$\begin{aligned} I^0(x, y) &= \text{false} \\ I^1(x, y) &= x \geqslant 0 \wedge (x = y \vee I^0(x+1, y)) \\ &= 0 \leqslant x = y \\ I^2(x, y) &= x \geqslant 0 \wedge (x = y \vee I^1(x+1, y)) \\ &= 0 \leqslant x \leqslant y \leqslant x+1 \end{aligned}$$



Iterates

$$\text{Iterates } I = \lim_{n \to \infty} F^n(\text{false})$$

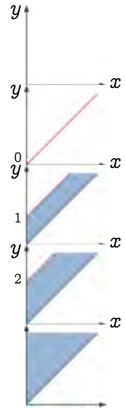
$$\begin{aligned} I^0(x, y) &= \text{false} \\ I^1(x, y) &= x \geqslant 0 \wedge (x = y \vee I^0(x+1, y)) \\ &= 0 \leqslant x = y \\ I^2(x, y) &= x \geqslant 0 \wedge (x = y \vee I^1(x+1, y)) \\ &= 0 \leqslant x \leqslant y \leqslant x+1 \\ I^3(x, y) &= x \geqslant 0 \wedge (x = y \vee I^2(x+1, y)) \\ &= 0 \leqslant x \leqslant y \leqslant x+2 \end{aligned}$$



Convergence acceleration: widening

Accelerated Iterates $I = \lim_{n \to \infty} F^n(\text{false})$

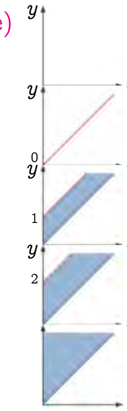
$$\begin{aligned}
I^0(x,y) &= \text{false} \\
I^1(x,y) &= x \geqslant 0 \wedge (x = y \vee I^0(x+1,y)) \\
&= 0 \leqslant x = y \\
I^2(x,y) &= x \geqslant 0 \wedge (x = y \vee I^1(x+1,y)) \\
&= 0 \leqslant x \leqslant y \leqslant x+1 \\
I^3(x,y) &= x \geqslant 0 \wedge (x = y \vee I^2(x+1,y)) \\
&= 0 \leqslant x \leqslant y \leqslant x+2 \\
I^4(x,y) &= I^2(x,y) \mathbin{\nabla} I^3(x,y) \leftarrow \text{widening} \\
&= 0 \leqslant x \leqslant y
\end{aligned}$$



Fixed point

Accelerated Iterates $I = \lim_{n \to \infty} F^n(\text{false})$

$$\begin{aligned}
I^0(x,y) &= \text{false} \\
I^1(x,y) &= x \geqslant 0 \wedge (x = y \vee I^0(x+1,y)) \\
&= 0 \leqslant x = y \\
I^2(x,y) &= x \geqslant 0 \wedge (x = y \vee I^1(x+1,y)) \\
&= 0 \leqslant x \leqslant y \leqslant x+1 \\
I^3(x,y) &= x \geqslant 0 \wedge (x = y \vee I^2(x+1,y)) \\
&= 0 \leqslant x \leqslant y \leqslant x+2 \\
I^4(x,y) &= I^2(x,y) \mathbin{\nabla} I^3(x,y) \leftarrow \text{widening} \\
&= 0 \leqslant x \leqslant y \\
I^5(x,y) &= x \geqslant 0 \wedge (x = y \vee I^4(x+1,y)) \\
&= I^4(x,y) \quad \text{fixed point!}
\end{aligned}$$

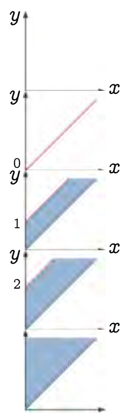


Octagons

Accelerated Iterates $I = \lim_{n \to \infty} F^n(\text{false})$

$$\begin{aligned}
I^0(x,y) &= \text{false} \\
I^1(x,y) &= x \geqslant 0 \wedge (x = y \vee I^0(x+1,y)) \\
&= 0 \leqslant x = y \\
I^2(x,y) &= x \geqslant 0 \wedge (x = y \vee I^1(x+1,y)) \\
&= 0 \leqslant x \leqslant y \leqslant x+1 \\
I^3(x,y) &= x \geqslant 0 \wedge (x = y \vee I^2(x+1,y)) \\
&= 0 \leqslant x \leqslant y \leqslant x+2 \\
I^4(x,y) &= I^2(x,y) \mathbin{\nabla} I^3(x,y) \leftarrow \text{widening} \\
&= 0 \leqslant x \leqslant y \\
I^5(x,y) &= x \geqslant 0 \wedge (x = y \vee I^4(x+1,y)) \\
&= I^4(x,y) \quad \text{fixed point!}
\end{aligned}$$

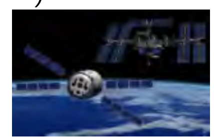
The invariants are computer representable with octagons!



Industrialisation: Development in cooperation with Airbus France

- Automatic proofs of absence of runtime errors in Electric Flight Control Software:
  - A340/600: 132.000 lines of C, 40mn on a PC 2.8 GHz, 300 Mb (Nov. 2003)
  - A380: 1.000.000 lines of C, 34h, 8 Gb (Nov. 2005)

  no false alarm, World premières !



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

(2) the Jules Vernes Automated Transfer Vehicle (ATV) enabling ESA to transport payloads to the International Space Station.

Application of abstract interpretation to program proof methods

Maximal execution trace

```
#include <stdio.h>
int main() {
          int x,y;
          printf("Enter an integer: ");
          scanf("%d",&x); y = x;
/* 1: */  while (x != 0) {
             printf("x = %d, y = %d\n",x,y);
/* 2: */     x = x - 1;
/* 3: */     y = y + 2;
          }
/* 4: */  printf("x = %d, y = %d\n",x,y); }
```

```
Enter an integer: 3
x = 3, y = 3
x = 2, y = 5
x = 1, y = 7
x = 0, y = 9
```

```
Enter an integer: -1
x = -1, y = -1
x = -2, y = 1
x = -3, y = 3
x = -4, y = 5
…
x = -738245, y = 1476487
…
```

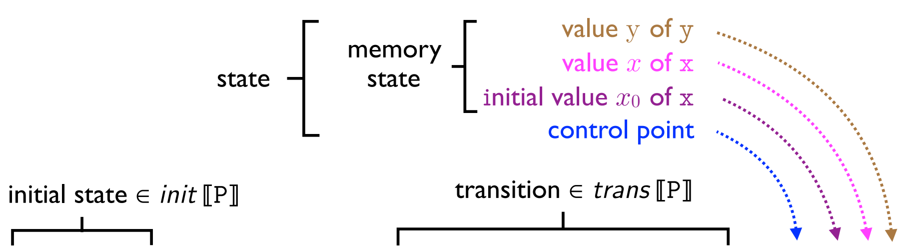
⟨1:,3,3,3⟩ ⟶ ⟨2:,3,3,3⟩ ⟶ ⟨3:,3,2,3⟩ ⟶ ⟨1:,3,2,5⟩ ⟶ ⟨2:,3,2,5⟩ ⟶ ⟨3:,3,1,5⟩ ⟶ ⟨1:,3,1,7⟩ ⟶ ⟨2:,3,1,7⟩ ⟶ ⟨3:,3,0,7⟩ ⟶ ⟨1:,3,0,9⟩ ⟶ ⟨6:,3,0,9⟩

Maximal execution trace

```
#include <stdio.h>
int main() {
          int x,y;
          printf("Enter an integer: ");
          scanf("%d",&x); y = x;
/* 1: */  while (x != 0) {
             printf("x = %d, y = %d\n",x,y);
/* 2: */     x = x - 1;
/* 3: */     y = y + 2;
          }
/* 4: */  printf("x = %d, y = %d\n",x,y); }
```

```
Enter an integer: 3
x = 3, y = 3
x = 2, y = 5
x = 1, y = 7
x = 0, y = 9
```

```
Enter an integer: -1
x = -1, y = -1
x = -2, y = 1
x = -3, y = 3
x = -4, y = 5
…
x = -738245, y = 1476487
```



⟨1:,3,3,3⟩ ⟶ ⟨2:,3,3,3⟩ ⟶ ⟨3:,3,2,3⟩ ⟶ ⟨1:,3,2,5⟩ ⟶ ⟨2:,3,2,5⟩ ⟶ ⟨3:,3,1,5⟩ ⟶ ⟨1:,3,1,7⟩ ⟶ ⟨2:,3,1,7⟩ ⟶ ⟨3:,3,0,7⟩ ⟶ ⟨1:,3,0,9⟩ ⟶ ⟨6:,3,0,9⟩

Maximal trace semantics

- The trace semantics of a program is the set of all possible maximal finite or infinite execution traces for that program
- The trace semantics of a programing language maps programs to their trace semantics

Inductive definition

- Partial traces:
  - A trace with one initial state is a partial trace
  - A partial trace extended by a transition is a partial trace
- Maximal traces:
  - Finite traces with no extension by a transition
  - Infinite traces which prefixes are all partial traces

Fixpoint partial trace semantics

- initial states of program P: $\mathit{init}[\![\mathrm{P}]\!]$
- transitions of programs P: $\mathit{trans}[\![\mathrm{P}]\!]$
- $\mathrm{F}^{\mathrm{t}}[\![\mathrm{P}]\!]X = \{ \mathrm{s} \mid \mathrm{s} \in \mathit{init}[\![\mathrm{P}]\!] \} \cup \{ \sigma \mathrm{ss}' \mid \sigma \mathrm{s} \in X \wedge \mathrm{ss}' \in \mathit{trans}[\![\mathrm{P}]\!] \}$
- $\mathrm{S}^{\mathrm{t}}[\![\mathrm{P}]\!] = \mathsf{lfp}^{\subseteq}\, \mathrm{F}^{\mathrm{t}}[\![\mathrm{P}]\!]$

Invariance abstraction

- Collect at each control point the possible values of variables when execution reaches that control point
- $\alpha(X)\mathrm{c} = \{\mathrm{m} \mid \exists \sigma,\sigma'.\ \sigma\langle \mathrm{c},\mathrm{m}\rangle\sigma' \in X\}$
- Invariance semantics: $\mathrm{S}^{\mathrm{i}}[\![\mathrm{P}]\!] = \alpha(\mathrm{S}^{\mathrm{t}}[\![\mathrm{P}]\!])$

Invariance abstraction

- Collect at each control point the possible values of variables when execution reaches that control point
- $\mathrm{S}^{\mathrm{i}}[\![\mathrm{P}]\!] = \alpha(\mathrm{S}^{\mathrm{t}}[\![\mathrm{P}]\!])\mathrm{c} = \{\mathrm{m} \mid \exists \sigma,\sigma'.\ \sigma\langle \mathrm{c},\mathrm{m}\rangle\sigma' \in \mathrm{S}^{\mathrm{t}}[\![\mathrm{P}]\!]\}$

```
#include <stdio.h>
int main() {
         int x,y;
         printf("Enter an integer: ");
         scanf("%d",&x); y = x;
/* 1: */ while (x != 0) {
            printf("x = %d, y = %d\n",x,y);
/* 2: */    x = x - 1;
/* 3: */    y = y + 2;
         }
/* 4: */ printf("x = %d, y = %d\n",x,y); }
```

$\{\langle x_0, x, \mathrm{y}\rangle \mid \mathrm{y} = 3x_0 - 2x\}$ ← /* 1: */

$\{\langle x_0, x, \mathrm{y}\rangle \mid \mathrm{y} = 3x_0 - 2x\}$ ← /* 2: */

$\{\langle x_0, x, \mathrm{y}\rangle \mid \mathrm{y} = 3x_0 - 2x - 2\}$ ← /* 3: */

$\{\langle x_0, x, \mathrm{y}\rangle \mid \mathrm{y} = 3x_0 \wedge x = 0\}$ ← /* 4: */

Calculations design of the verification conditions

- $\alpha(\mathrm{F}^{\mathrm{t}}[\![\mathrm{P}]\!]X)$
  $= \boldsymbol{\lambda}\mathrm{c}.\{\mathrm{m} \mid \exists\sigma,\sigma'.\ \sigma\langle\mathrm{c,m}\rangle\sigma' \in X\}$
  $= \ldots$
  $= \mathrm{F}^{\mathrm{i}}[\![\mathrm{P}]\!](\alpha(X))$
  where $\mathrm{F}^{\mathrm{i}}[\![\mathrm{P}]\!]$ are the Turing/Floyd/Naur/Hoare verification conditions
- It follows that $\mathrm{S}^{\mathrm{i}}[\![\mathrm{P}]\!] = \mathsf{lfp}^{\dot{\subseteq}}\ \mathrm{F}^{\mathrm{i}}[\![\mathrm{P}]\!]$
- The proof method is then by fixpoint induction (Tarski 1955)

Application to the semantics of programming languages

General idea

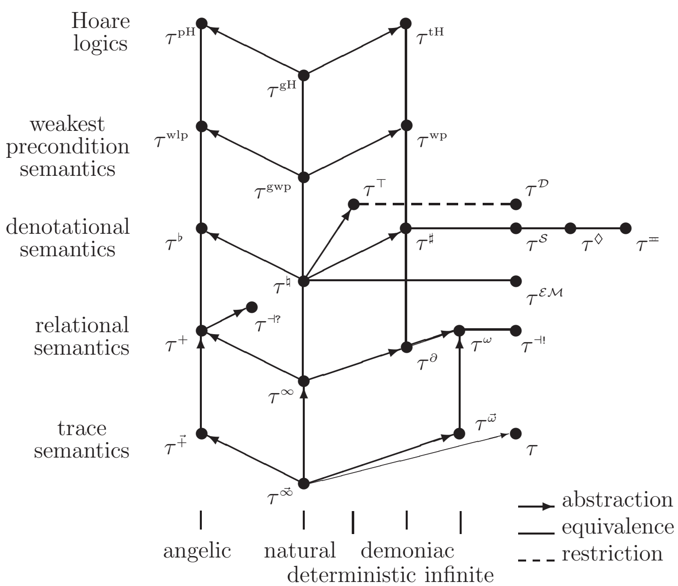
- All known semantics are abstractions of a most precise semantics

Abstraction to denotational semantics

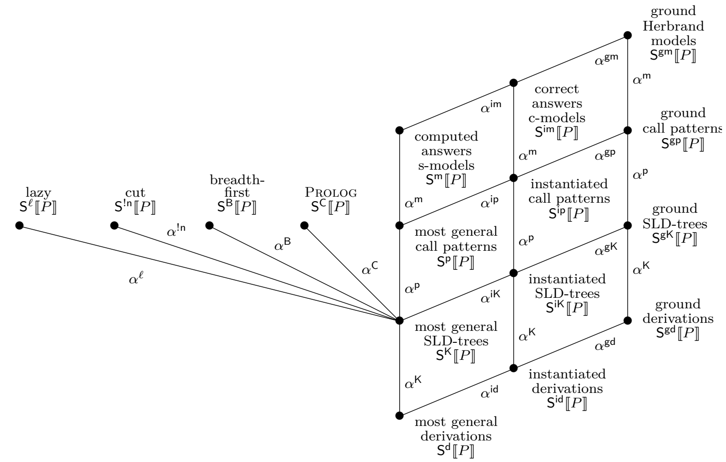
- The maximal trace semantics $\mathrm{S}^{\mathrm{m}}[\![\mathrm{P}]\!]$ (maximal finite and infinite execution traces
- Denotational semantics abstraction:
  - $\mathrm{S}^{\mathrm{d}}[\![\mathrm{P}]\!] = \alpha(\mathrm{S}^{\mathrm{m}}[\![\mathrm{P}]\!])$
  - $\alpha(X) = \boldsymbol{\lambda}\mathrm{s}.\{\mathrm{s}' \mid \exists\sigma.\ \mathrm{s}\sigma\mathrm{s}' \in X\} \cup \{\perp \mid \exists\sigma.\ \mathrm{s}\sigma\ldots \in X\}$

  i.e. a map of initial states to the set of final states plus $\perp$ in case of non-termination

Hierarchy of abstractions



idem for Prolog



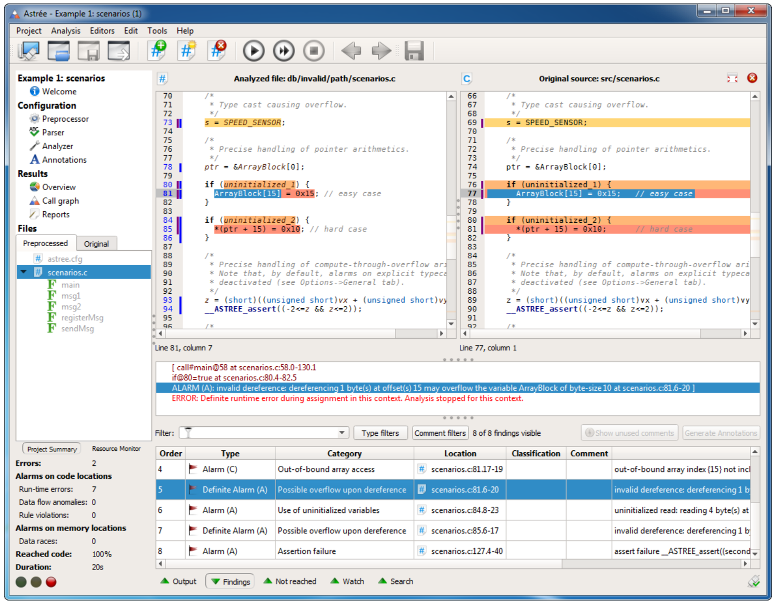
- all semantics are abstractions of $S^{d}[\![P]\!]$

Conclusion

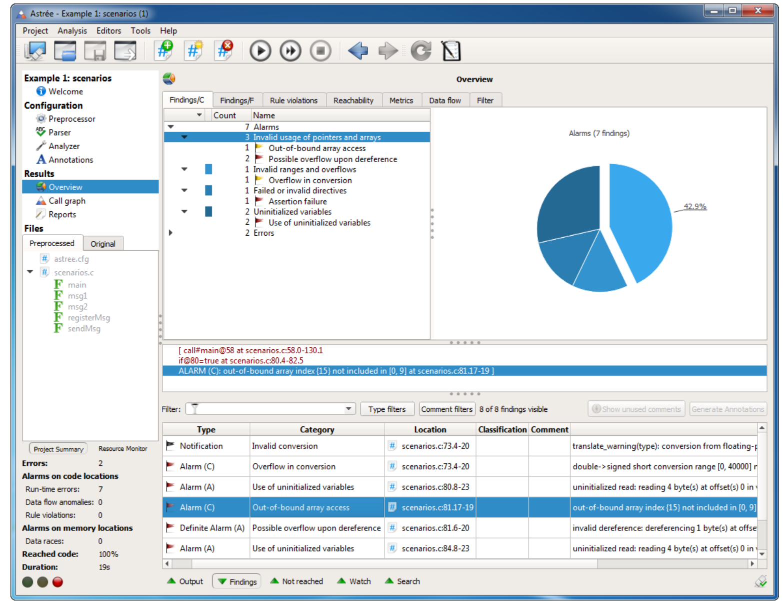
Abstract interpretation

- A well-developed theory, still in progress
- Active research e.g.
  - abstract domains to handle e.g. complex data structures
  - abstraction of parallelism with weak memory models
  - applications to biology, …
- Industrial-quality static analyzers

Industrialisation: Astrée



Industrialisation: Astrée



Many other static analyzers

- Julia (Java) http://www.juliasoft.com
- Ikos, NASA https://ti.arc.nasa.gov/opensource/ikos/
- Clousot for code contract, Microsoft, https://github.com/Microsoft/CodeContracts
- Infer (Facebook) http://fbinfer.com
- Zoncolan (Facebook)
- Google
- …

Static analysis for software development

- Users of Astrée:

  …

- Why not all software developers use static analysis tools?

Irresponsibility

- Computer engineering is the only technology where developers are not responsible for their errors, even the trivial ones:

DISCLAIMER OF WARRANTIES. ... *MICROSOFT AND ITS SUPPLIERS PROVIDE THE SOFTWARE, AND SUPPORT SERVICES (IF ANY) AS IS AND WITH ALL FAULTS, AND MICROSOFT AND ITS SUPPLIERS HEREBY DISCLAIM ALL OTHER WARRANTIES AND CONDITIONS, WHETHER EXPRESS, IMPLIED OR STATUTORY, INCLUDING, BUT NOT LIMITED TO, ANY (IF ANY) IMPLIED WARRANTIES, DUTIES OR CONDITIONS OF MERCHANTABILITY, OF FITNESS FOR A PARTICULAR PURPOSE, OF RELIABILITY OR AVAILABILITY, OF ACCURACY OR COMPLETENESS OF RESPONSES, OF RESULTS, OF WORKMANLIKE EFFORT, OF LACK OF VIRUSES, AND OF LACK OF NEGLIGENCE, ALL WITH REGARD TO THE SOFTWARE, AND THE PROVISION OF OR FAILURE TO PROVIDE SUPPORT OR OTHER SERVICES, INFORMATION, SOFTWARE, AND RELATED CONTENT THROUGH THE SOFTWARE OR OTHERWISE ARISING OUT OF THE USE OF THE SOFTWARE. ...*

The future

- Safety and security does matter to the general public
- Computer scientists will ultimately be held responsible for there errors
- At least the automatically discoverable ones
- Since this is now part of the state of the art
- Automatic static analysis, verification, etc has a brilliant future.

Francesco Logozzo, designer of the Zoncolan static analyzer at Facebook wrote me on 09/12/2016:

``Finding people who really know static analysis is very hard, you should tell your students that if they want a great job in a Silicon Valley company they should study abstract interpretation not JavaScript. Feel free to quote me on that ;-)''

Selected bibliography

- Patrick Cousot, Radhia Cousot:
  Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. POPL 1977: 238-252
- Patrick Cousot, Nicolas Halbwachs:
  Automatic Discovery of Linear Restraints Among Variables of a Program. POPL 1978: 84-96
- Patrick Cousot, Radhia Cousot:
  Systematic Design of Program Analysis Frameworks. POPL 1979: 269-282
- Patrick Cousot, Radhia Cousot:
  Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation. PLILP 1992: 269-295
- Patrick Cousot:
  Types as Abstract Interpretations. POPL 1997: 316-331
- Patrick Cousot, Radhia Cousot:
  Temporal Abstract Interpretation. POPL 2000: 12-25
- Patrick Cousot, Radhia Cousot:
  Systematic design of program transformation frameworks by abstract interpretation. POPL 2002: 178-190
- Patrick Cousot:
  Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. Theor. Comput. Sci. 277(1-2): 47-103 (2002)
- Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, Xavier Rival:
  A static analyzer for large safety-critical software. PLDI 2003: 196-207
- Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, Xavier Rival:
  The ASTREÉ Analyzer. ESOP 2005: 21-30
- Patrick Cousot, Radhia Cousot, Roberto Giacobazzi:
  Abstract interpretation of resolution-based semantics. Theor. Comput. Sci. 410(46): 4724-4746 (2009)
- Patrick Cousot, Radhia Cousot:
  An abstract interpretation framework for termination. POPL 2012: 245-258
- Patrick Cousot, Radhia Cousot:
  A Galois connection calculus for abstract interpretation. POPL 2014: 3-4
- Julien Bertrane, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, Xavier Rival:
  Static Analysis and Verification of Aerospace Software by Abstract Interpretation. Foundations and Trends in Programming Languages 2(2-3): 71-190 (2015)

The End, Thank You