Floating Point Verification

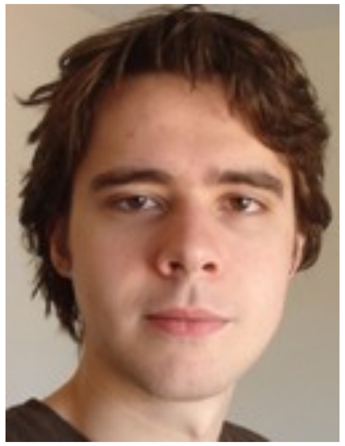
Unifying Abstract Interpretation and Decision Procedures

Leopold Haller

(joint work with Vijay D'Silva, Michael Tautschnig, Daniel Kroening)

UNIVERSITY OF OXFORD

2nd November 2011



Presentation Outline

Part I

Existing approaches to FP - Verification

- Manual, Semi-automated
- Decision Procedures
- Abstract Interpretation

Part II

Decision Procedures — Precise

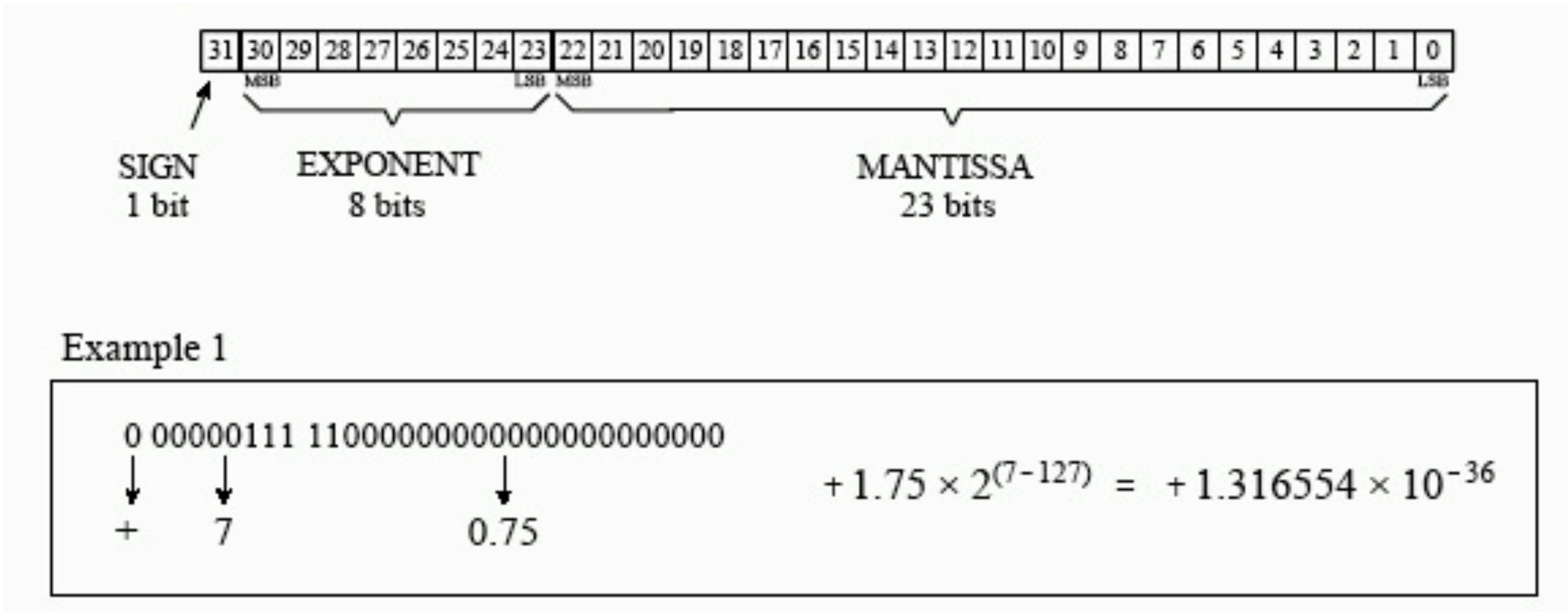
Abstract Interpretation — Scalable

Abstract Satisfiability

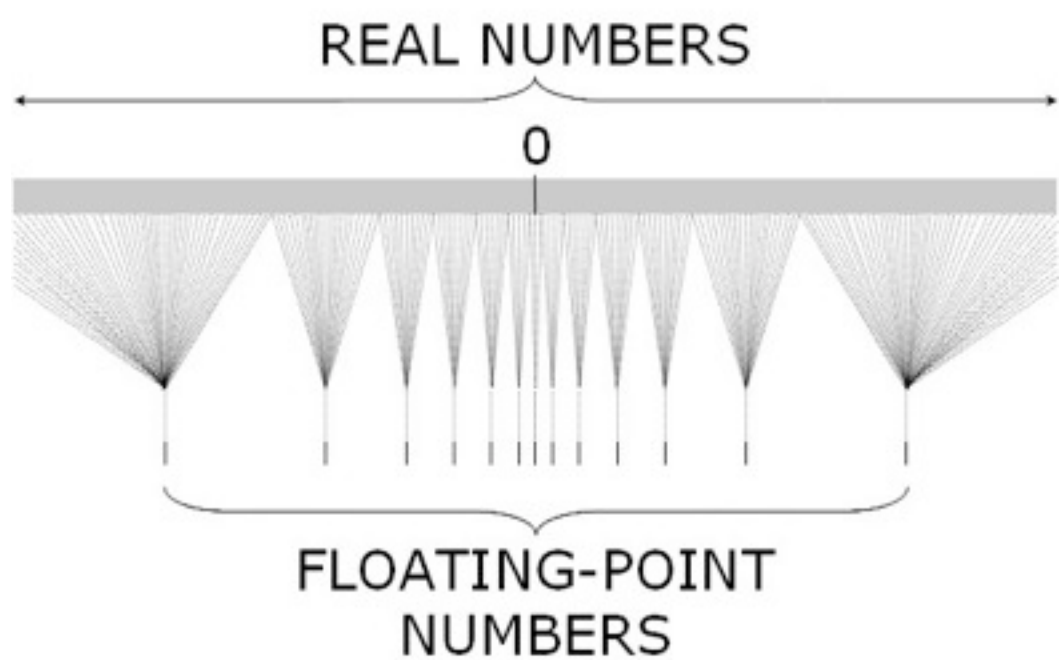
Our research

Part I

IEEE754 Floating Point Numbers



Special values: $-0, +0, -\infty, \infty, NaN$



The Pitfalls of FP

I

```
if(x < y)
  ...
else if(x > y)
  ...
else assert(x == y);
```

II

```
if(x > 0)
{
  for(float sum = 0; sum <= N; sum+=x)
    ...

  //does the loop terminate?
}
```

III

```
float r1 = a+b;
float r2 = b+c;

r1+= c; r2 += a;

assert(r1 == r2);
```

IV

```
float r1 = a+b;
float r2 = a+b;

assert(r1 == r2);
```

V

```
bool b = false;

if(f < 1)
  b = true;

if(!b)
  assert(f >= 1);
```

sine.c (~/work/cprover/src/ai) – VIM

```c
#define HALFPI 1.57079632679f

float sine_approx(float x)
{
  if(x <= -HALFPI || x >= HALFPI);
    return 0.0f;

  float result = x - (x*x*x)/6.0f;
  result += (x*x*x*x*x)/120.0f;
  result += (x*x*x*x*x*x*x)/5040.0f;

  assert(result <= 1.01 && result >= -1.01);

  return 0;
}
```

~/work/cprover/src/ai/sine.c [POS=0016,0001][100%] [LEN=16]

Is this program correct?

What does correctness mean?

sine.c (~/work/cprover/src/ai) - VIM

```c
#define HALFPI 1.57079632679f

float sine_approx(float x)
{
  if(x <= -HALFPI || x >= HALFPI);
    return 0.0f;

  float result = x - (x*x*x)/6.0f;
  result += (x*x*x*x*x)/120.0f;
  result += (x*x*x*x*x*x*x)/5040.0f;

  assert(result <= 1.01 && result >= -1.01);

  return 0;
}
```

~/work/cprover/src/ai/sine.c [POS=0016,0001][100%] [LEN=16]

Three possible meanings:

- Result is sufficiently close to the real number result
- Result is sufficiently close to the sine function
- <u>The assertion cannot be violated</u> ←

How can we check correctness?

Manual

Abstract Interpretation

Decision Procedures

Manual, semi-automated

C-Program $\xrightarrow{\text{automatic translation}}$ Theorem statement

$\downarrow$

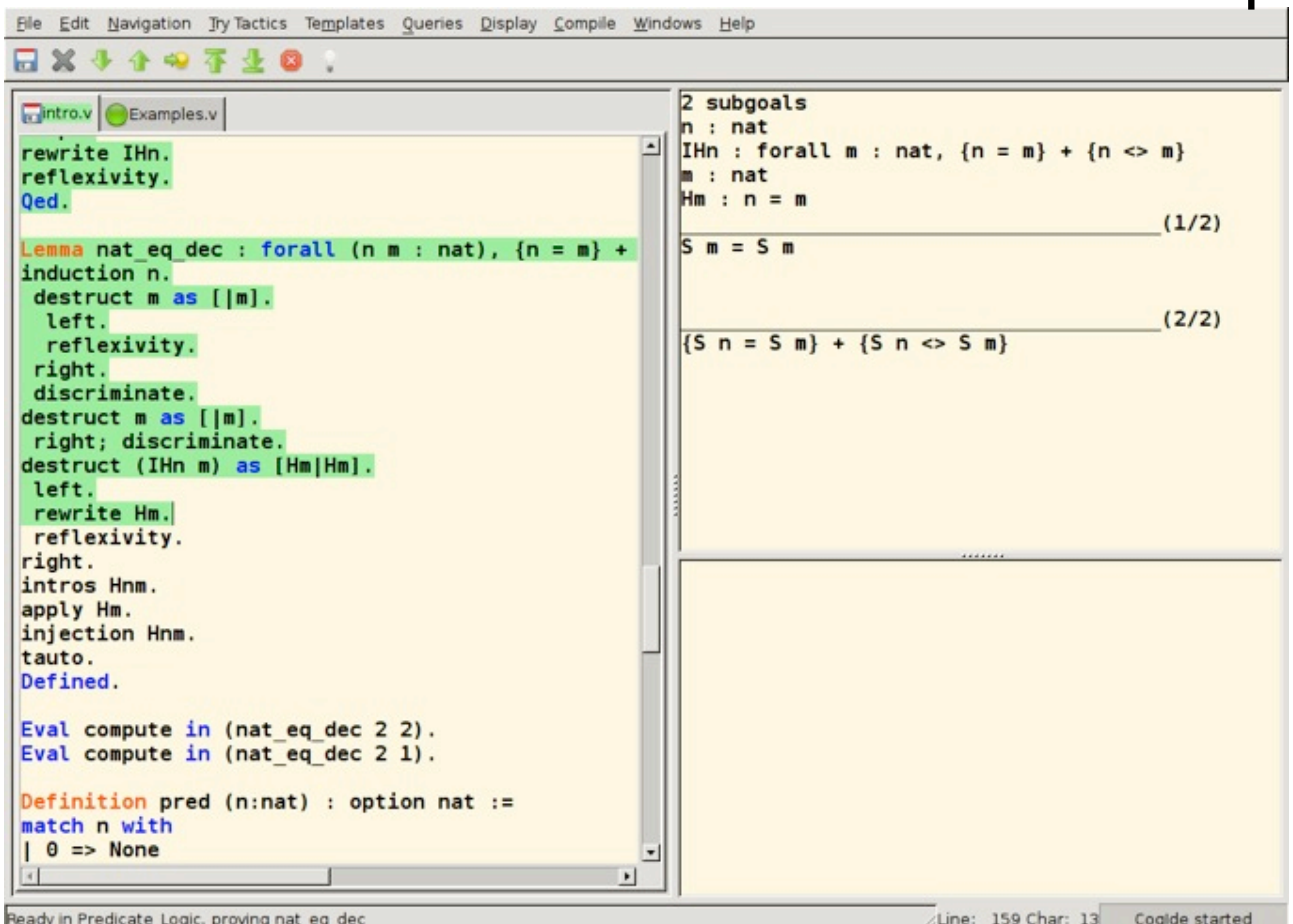
Prover returns unsolved subgoals $\longleftrightarrow$ User simplifies subgoals

- Use an interactive theorem prover
- Experts write proof scripts with machine assistance
- Potentially powerful, but expensive
- Proof scripts require expert understanding, may be much harder to write than programs

Manual, semi-automated

User enters proof

Computer keeps track of what is left to prove



Manual, semi-automated

This proof file has been written by Sylvie Boldo(1), following a proof presented by Pr William Kahan (2), and adapted to Coq proof checker with the help of Guillaume Melquiond(1) and Marc Daumas(1). This work has been partially supported by the CNRS grant PICS 2533.

(1) LIP Computer science laboratory UMR 5668 CNRS - ENS de Lyon - INRIA Lyon, France

(2) University of California at Berkeley Berkeley, California

```
Section Discriminant1.
Variable bo : Fbound.
Variable precision : nat.

Let radix := 2%Z.

Let FtoRradix := FtoR radix.
Coercion FtoRradix : float >-> R.

Theorem TwoMoreThanOne : (1 < radix)%Z.

Let radixMoreThanZERO := Zlt_1_O _ (Zlt_le_weak _ _
TwoMoreThanOne).
Hypothesis precisionGreaterThanOne : 1 < precision.
Hypothesis pGivesBound : Zpos (vNum bo) = Zpower_nat radix
precision.

Variables a b b' c p q d:float.

Let delta := (Rabs (d-(b*b'-a*c)))%R.

Hypothesis Fa : (Fbounded bo a).
Hypothesis Fb : (Fbounded bo b).
Hypothesis Fb': (Fbounded bo b').
Hypothesis Fc : (Fbounded bo c).
Hypothesis Fp : (Fbounded bo p).
Hypothesis Fq : (Fbounded bo q).
Hypothesis Fd : (Fbounded bo d).
```

There is no underflow

```
Hypothesis U1:(- dExp bo <= Fexp d - 1)%Z.
Hypothesis Nd:(Fnormal radix bo d).
Hypothesis Nq:(Fnormal radix bo q).
Hypothesis Np:(Fnormal radix bo p).

Hypothesis Square:(0 <=b*b')%R.
```

```
Hypothesis Roundp : (EvenClosest bo radix precision (b*b')%R p).
Hypothesis Roundq : (EvenClosest bo radix precision (a*c)%R q).

Hypothesis Firstcase : (p+q <= 3*(Rabs (p-q)))%R.
Hypothesis Roundd : (EvenClosest bo radix precision (p-q)%R d).

Theorem delta_inf: (delta <= (/2)*(Fulp bo radix precision d)+
   ((/2)*(Fulp bo radix precision p)+(/2)*(Fulp bo radix precision
q)))%R.

Theorem P_positive: (Rle 0 p)%R.

Theorem Fulp_le_twice_l: forall x y:float, (0 <= x)%R ->
   (Fnormal radix bo x) -> (Fbounded bo y) -> (2*x<=y)%R ->
   (2*(Fulp bo radix precision x) <= (Fulp bo radix precision
y))%R.

Theorem Fulp_le_twice_r: forall x y:float, (0 <= x)%R ->
   (Fnormal radix bo y) -> (Fbounded bo x) -> (x<=2*y)%R ->
   ((Fulp bo radix precision x) <= 2*(Fulp bo radix precision
y))%R.

Theorem Half_Closest_Round: forall (x:float) (r:R),
   (- dExp bo <= Zpred (Fexp x))%Z -> (Closest bo radix r x)
  -> (Closest bo radix (r/2)%R (Float (Fnum x) (Zpred (Fexp x)))).

Theorem Twice_EvenClosest_Round: forall (x:float) (r:R),
   (-(dExp bo) <= (Fexp x)-1)%Z -> (Fnormal radix bo x)
  -> (EvenClosest bo radix precision r x)
  -> (EvenClosest bo radix precision (2*r)%R (Float (Fnum x) (Zsucc
(Fexp x)))).

Theorem EvenClosestMonotone2: forall (p q : R) (p' q' : float),
  (p <= q)%R -> (EvenClosest bo radix precision p p') ->
  (EvenClosest bo radix precision q q') -> (p' <= q')%R.

Theorem Fulp_le_twice_r_round: forall (x y:float) (r:R), (0 <= x)%R
->
   (Fbounded bo x) -> (Fnormal radix bo y) -> (- dExp bo <= Fexp y
- 1)%Z
    -> (x<=2*r)%R ->
   (EvenClosest bo radix precision r y) ->
   ((Fulp bo radix precision x) <= 2*(Fulp bo radix precision
y))%R.

Theorem discri1: (delta <= 2*(Fulp bo radix precision d))%R.

End Discriminant1.
```

etc...

Manual, semi-automated

Selection of notable work:

- John Harrison (Intel) - Verification of FP hardware and firmware using HOL

- Various formalizations of IEEE754 FP arithmetic for different theorem provers

- Boldot, Filliâtre, Melquiond et. al. - Theorem prover combined with incomplete FP prover.

Manual, semi-automated

Conclusion:

- Manual or semi-automated techniques can be very powerful, but <u>require experts</u> and large time investments

- Results have limited reusability

- Typically feasible for small system components of <u>critical importance</u> (e.g., Intel's verification of processor components)

References

Axiomatisations of FP

M. Daumas, L. Rideau and L. Théry. A Generic Library for Floating-Point Numbers and Its Application to Exact Computing. TPHOLs 2001

G. Melquiond. Floating-point arithmetic in the Coq system. RNC 2008.

P. Miner and S. Boldo. Float PVS library. http://shemesh.larc.nasa.gov/fm/ftp/larc/PVS-library/pvslib.html

P. Miner. Defining the IEEE-854 Floating-Point Standard in PVS. PVS. Technical Memorandum NASA, Langley Research, 1995

J. Harrison. A machine-checked theory of floating-point arithmetic. TPHOLs 1999

Specification of FP properties

A. Ayad and C. Marché. Behavioral properties of floating-point programs. Hisseo publications, 2009.

A. Ayad and C. Marché. Multi-prover verification of floating-point programs. *www.lri.fr/~marche/ayad10ijcar-submission.pdf*, 2010

S. Boldo and J.C. Filliâtre. Formal verification of floating-point programs. ARITH 2007

References

Applications

J. Harrison. Floating point verification in HOL light: The exponential function. FMSD, 16(3), 2000

J. Harrison. Floating-point verification. FM 2005

J. Harrison. Formal verification of square root algorithms. FMSD, 22(2), 2003

B. Akbarpour, A.T. Abdel-Hamid, S. Tahar, and J. Harrison. Verifying a synthesized implementation of IEEE-754 floating-point exponential function using HOL. CJ 53(4), 2010

J. O'Leary, X. Zhao, R. Gerth, C.H. Seger. Formally Verifying IEEE Compliance of Floating-Point Hardware

R, Kaivola and M. D. Aagaard. Divider circuit verification with model checking and theorem proving. TPHOLs 2000

M. Cornea-Hasegan. Proving the IEEE correctness of iterative floating-point square root, divide and remainder algorithms. Intel Technology Journal, Q2, 1998

T. L. J. Strother Moore and M. Kaufmann. A mechanically checked proof of the correctness of the kernel of the AMD5K86 floating-point division algorithm. IEEE Transactions on Computers, 47(9), 1998.

D. Rusinoff. A mechanically checked proof of IEEE compliance of a register-transfer-level specification of the AMD-K7 floating-point multiplication, division, and square root instructions. LMS Journal of Computation and Mathematics, I, 1998.

J. Sawada. Formal verification of divide and square root algorithms using series calculation. ACL2 2002.

S. Boldo, J.-C. Filliâtre and G. Melquiond. Combining Coq and Gappa for Certifying Floating-Point Programs. Calculemus 2009.

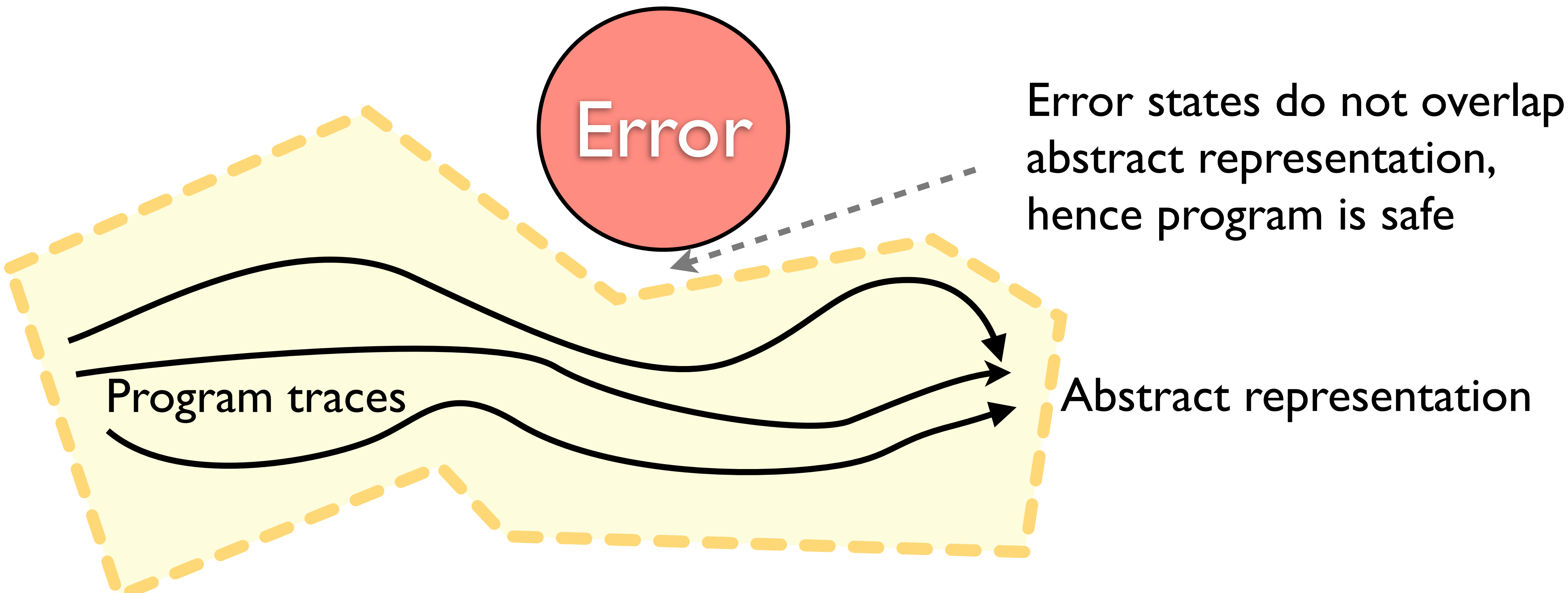
Requires experts,
expensive, powerful

Manual

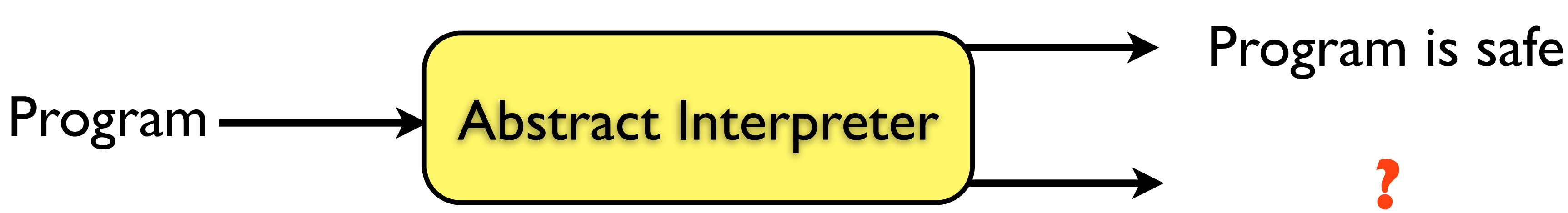
Abstract Interpretation

Decision Procedures

Abstract Interpretation



- Instead of exploring all executions, explore a single abstract execution
- Abstract execution contains all concrete executions!
- Highly efficient and scalable, but imprecise



Abstract Interpretation

An abstract interpreter modularly uses operations provided by an abstract domain. Changing the domain changes the analysis.

$$\boxed{\text{Interpreter}} \Longleftrightarrow \boxed{\text{Abstract Domain}}$$

Example

Code	Signs domain $\{+, -\} \cup \{?\}$	Constants domain $\{c \mid c \in FP\} \cup \{?\}$
`float y = 5;`	$y = +$	$y = 5$
`if(x > 0)`	$x = +$	$x = ?$
`{`		
`float z = x*y;`	$z = +$	$z = ?$
`assert(z > 0);`	safe!	Possibly unsafe
`}`		

Abstract Interpretation

An abstract interpreter modularly uses operations provided by an abstract domain. Changing the domain changes the analysis.

Interpreter ⟺ Abstract Domain

Example

Interval Domain

$$\{[l, u] \mid l, u \in Int\}$$

```
int x,y;
```

$$x, y \in [\min(Int), \max(Int)]$$

```
if(y < 0)
{
  x = y;
```

$$x, y \in [\min(Int), -1]$$

```
}
else
{
  y++;
  x = 5;
```

$$x \in [5, 5], y \in [\min(Int), \max(Int)]$$

```
}
```

$$x \in [\min(Int), 5], y \in [\min(Int), \max(Int)]$$

```
assert(x < 6);
```

Abstract Interpretation

<u>Floating Point Intervals</u> $\{[l,u] \mid l,u \in FP\} \cup \{?\}$

sine.c (~/work/cprover/src/ai) – VIM

```
#define HALFPI 1.57079632679f

float sine_approx(float x)
{
  if(x <= -HALFPI || x >= HALFPI);
    return 0.0f;

  float result = x - (x*x*x)/6.0f;
  result += (x*x*x*x*x)/120.0f;
  result += (x*x*x*x*x*x*x)/5040.0f;

  assert(result <= 1.01 && result >= -1.01);

  return 0;
}
~
~
~/work/cprover/src/ai/sine.c  [POS=0016,0001][100%]  [LEN=16]
```

$x \in [-1.570796, 1.570796]$

$result \in [-2.216760, 2.216760]$

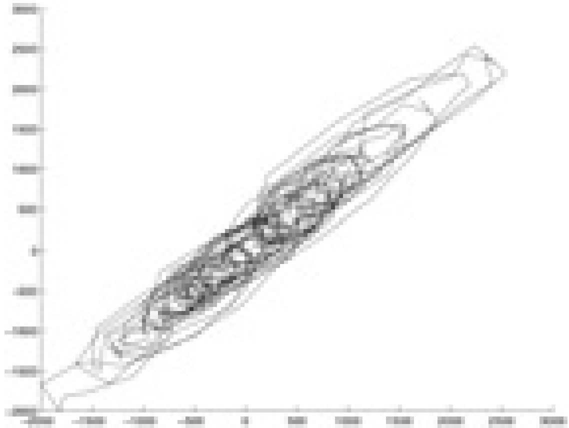
$result \in [-2.296453, 2.296453]$

$result \in [-2.301135, 2.301135]$

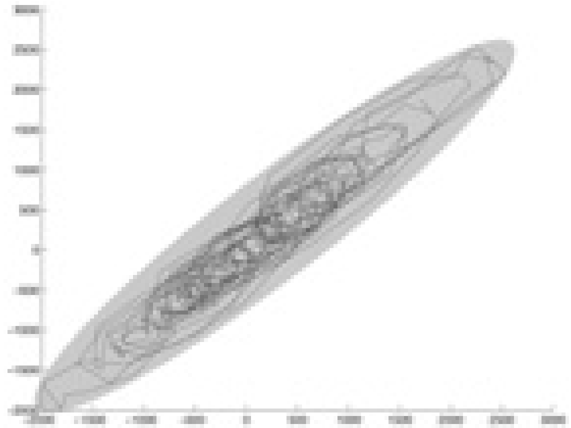
Potentially unsafe

Astrée Abstract Interpreter

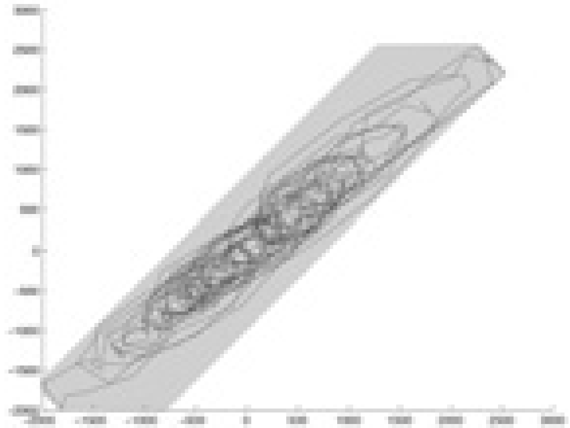
- Mature abstract interpreter by Cousot et. al
- Large number of domains
- Sold and supported by Absint GmbH
- Successful in proving correct large avionics control software: 100k lines of code in 1h -> <u>highly scalable</u>
- Various domains for floating point analysis:



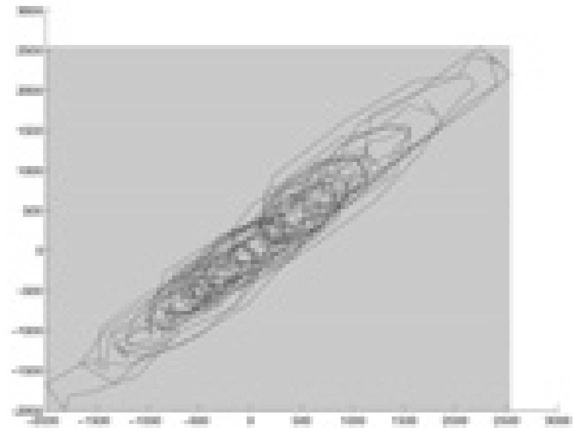
Original traces



Ellipses



Octagons



Intervals

Abstract Domains for Floating Point

- Abstract domains are typically formulated over the real or rational numbers
- Numeric domains rely on mathematical properties such as associativity which do not hold over floating point numbers

$$(a + b) + c = a + (b + c)$$

- Solution (Minet 2004): Interpret operations over floating point numbers as real number operations + error terms

```
double d;
float f1,f2;

f1 = (float) d;

f2 = f1*f2;
```

```
real d;
real f1, f2;

f1 = d + round_error(FLOAT_CAST,d);

f2 = f1*f2 + round_error(FLOAT_MULT, f1,f2);
```

Fluctuat: Errors as First Class Citizens

- Static analyser built for FP precision analysis
- Idea: Keep track separately of three distinct values for each variable

$$(f^x, r^x, e^x)$$

FP value | Real value | <u>FP error</u>

- Abstract these values separately

```
float x;

if(*) x = 1f;
else x = 0f;
```

`if(*) x = 1f;` → $([1,1],[1,1],[0,0])$

`else x = 0f;` → $([0,0],[0,0],[0,0])$

→ $([0,1],[0,1],[0,0])$ FP and real value are imprecise, but there is no rounding error

Fluctuat: Tracking errors with Zonotopes

- Fluctuat uses zonotope abstractions which combines intervals with noise symbols

```
void foo(float x, bool b)
{
  if(!(x <= 1 && x >= 0))
    return;

  float y = 1.0f + 2*x;

}
```

Noise variables take values in $[0, 1]$

$x = \varepsilon_1 \qquad (\equiv x \in [0, 1])$

$y = 2 * \varepsilon_1 + c * \varepsilon_2 + 1.0f$

Relation to x is preserved

new error symbol models rounding error

- The source of imprecisions can be precisely tracekd

Imprecision in Abstract Interpretation

- The <u>efficiency</u> of abstract interpreters comes <u>at the cost of precision</u>. Imprecision is accumulated from three sources:

  - Statements

    $x \in [-5, 5]$ `y = x * x;` $y \in [-25, 25]$

    $x \in [0, 1]$ `y = x;` $x, y \in [0, 1]$

  - Control-flow

    ```
    if(y < 0)
      x = 1;
    else
      x = -1;
    ```

    $x \in [-1, 1]$

  - Loops

    $x, y \in [1, 1]$

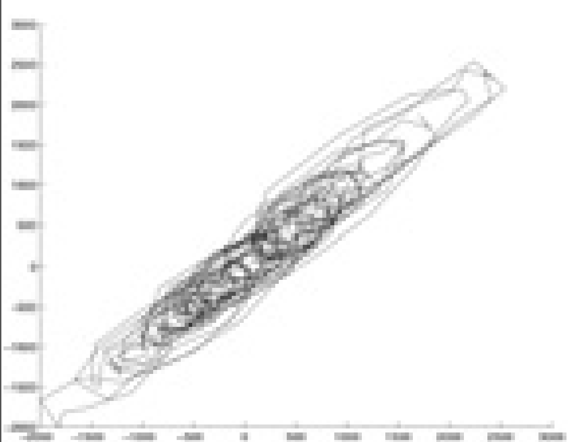
    ```
    while(x < 100000)
    {  x++;  y++;  }
    ```

    $x \in [100001, \max(Int)]$

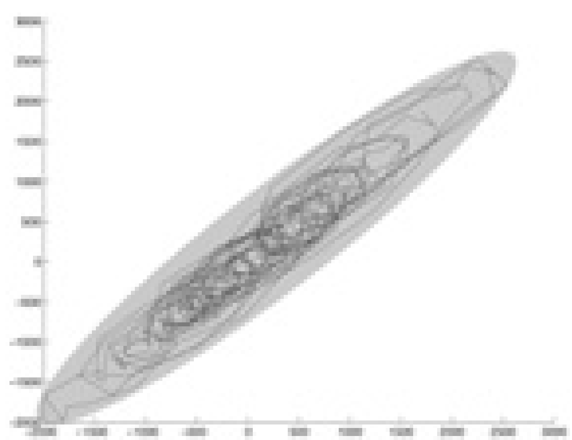
    $y \in [\min(Int), \max(Int)]$

Imprecision in Abstract Interpretation

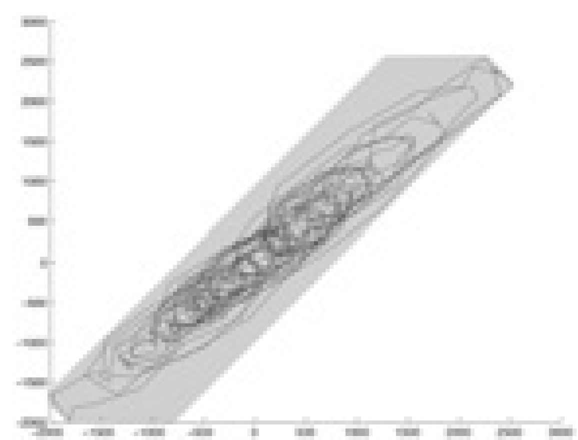
- For efficiency reasons, most numeric abstract domains are convex



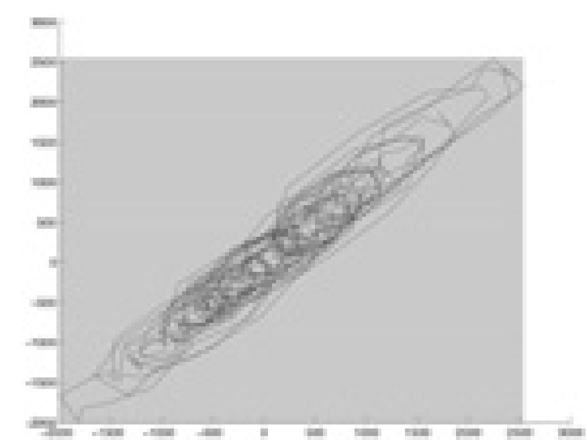
Original traces



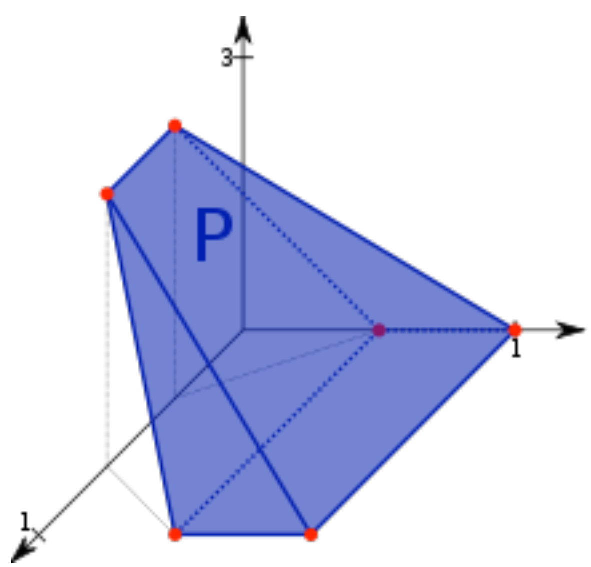
Ellipses



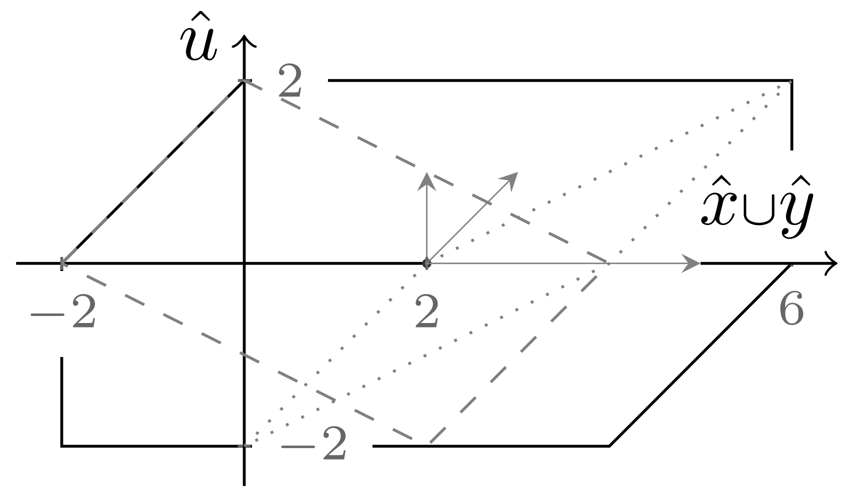
Octagons



Intervals



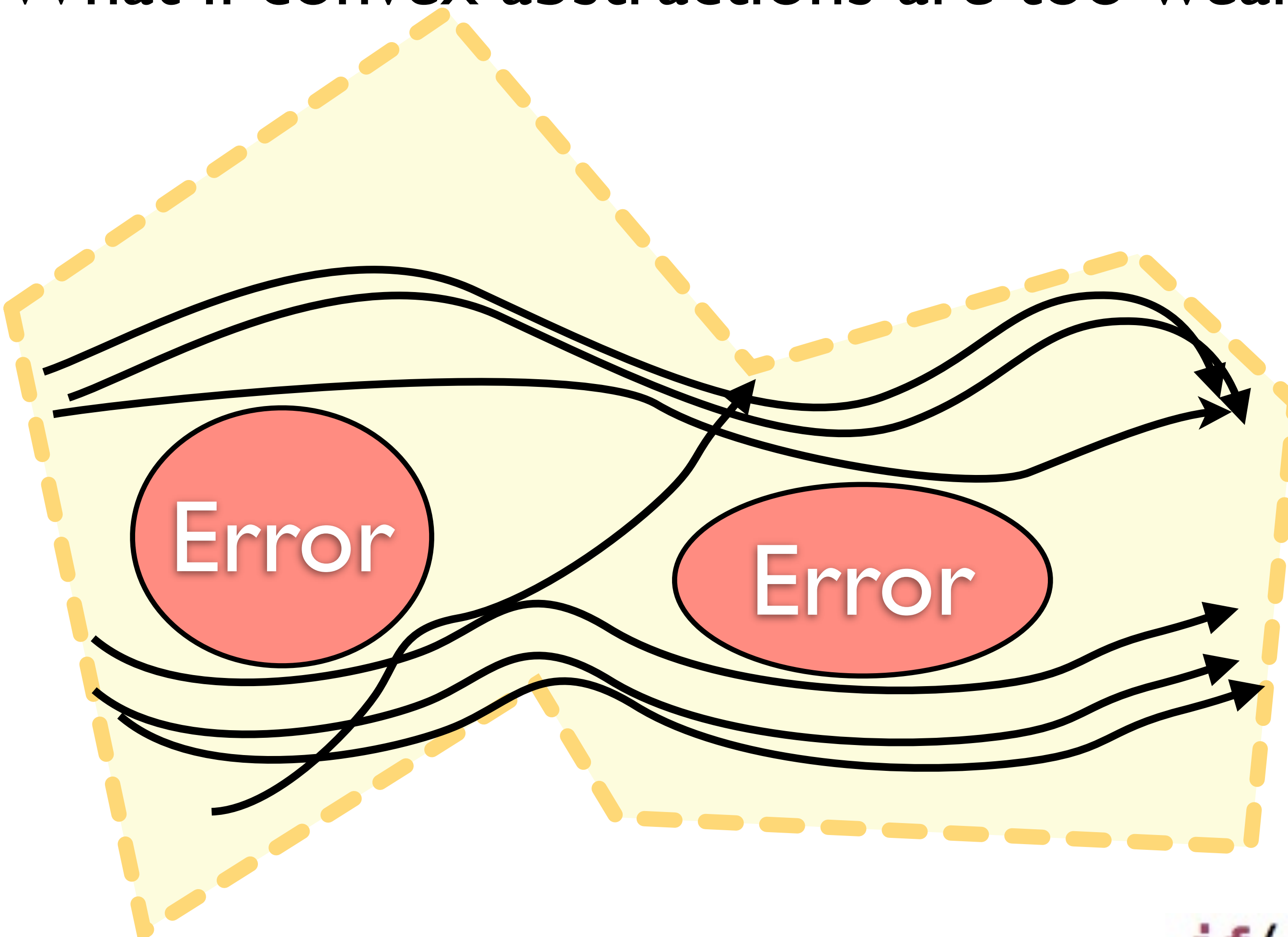
Convex polyhedra



Zonotope

Imprecision in Abstract Interpretation

What if convex abstractions are too weak?



<u>Very</u> common scenario

```
if(*)
  x = 1;
else
  x = -1;

assert(x != 0);
```

Handling Imprecision

What happens if the analysis is imprecise?

Customer

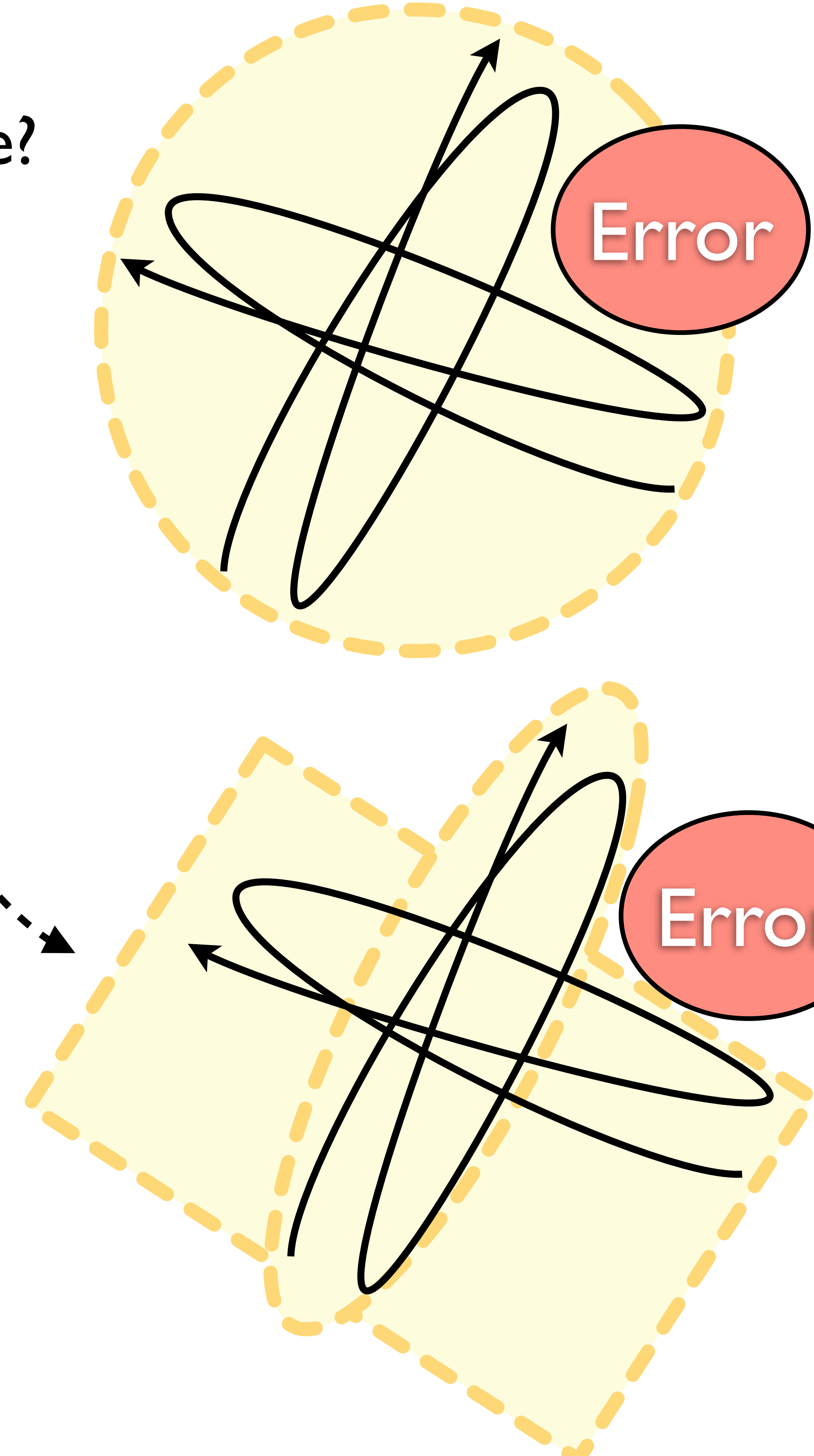
sends code

sends manually
created configuration

AbsInt GmbH

manually creates
new abstract domain

Researcher



Abstract Interpretation

Conclusion:

- Very scalable
- Imprecise
- Precise results require experts and research effort
- Expert created domains are moderately reusable
- Feasible for programs with homogenous structure and behaviour (success in avionics)

References

Floating point abstract domains

A. Chapoutot. Interval slopes as a numerical abstract domain for floating-point variables. SAS 2010

L. Chen, A. Miné and P. Cousot. A sound floating-point polyhedra abstract domain. APLAS 2008

A. Miné. Relational abstract domains for the detection of floating-point run-time errors. ESOP 2004

L. Chen, A. Miné, J. Wang and P. Cousot. An abstract domain to discover interval Linear Equalities. VMCAI 2010

L. Chen, A. Miné, J. Wang and P. Cousot. Interval polyhedra: An Abstract Domain to Infer Interval Linear Relationships. SAS 2009

K. Ghorbal, E. Goubault and S. Putot. The zonotope abstract domain Taylor1. CAV 2009

B. Jeannet, and A. Miné. Apron: A library of numerical abstract domains for static analysis. CAV 2009

D. Monniaux. Compositional analysis of floating-point linear numerical filters. CAV 2005

J. Feret. Static analysis of digital filters. ESOP 2004

F. Alegre, E. Feron and S. Pande. Using ellipsoidal domains to analyze control systems software. CoRR 2009

E. Goubault and S. Putot. Weakly relational domains for floating-point computation analysis. NSAD 2005

E. Goubault. Static analyses of the precision of floating-point operations. SAS 2001

References

Industrial Case Studies

E. Goubault, S. Putot, P. Baufreton, J. Gassino. Static analysis of the accuracy in control systems: principles and experiments. FMICS 2007

D. Delmas, E. Goubault, S. Putot, J. Souyris, K. Tekkal, F. Védrine. Towards an industrial use of FLUCTUAT on safety-critical avionics software. FMICS 2009

J. Souyris and D. Delmas. Experimental assessment of Astrée on safety-critical avionics software. SAFECOMP 2007

J. Souyris. Industrial experience of abstract interpretation-based static analyzers. IFIP 2004

P. Cousot. Proving the absence of run-time errors in safety-critical avionics code. EMSOFT 2007

FP Static Analysers

B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux and X. Rival. A static analyzer for large safety-critical software. SIGPLAN 38(5), 2003

P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux and Xavier Rival. The ASTREÉ analyzer. ESOP 2005

E. Goubault, M. Martel and S. Putot. Asserting the precision of floating-point computations: a simple abstract interpreter. ESOP 2002

Requires experts,
expensive, powerful

Manual

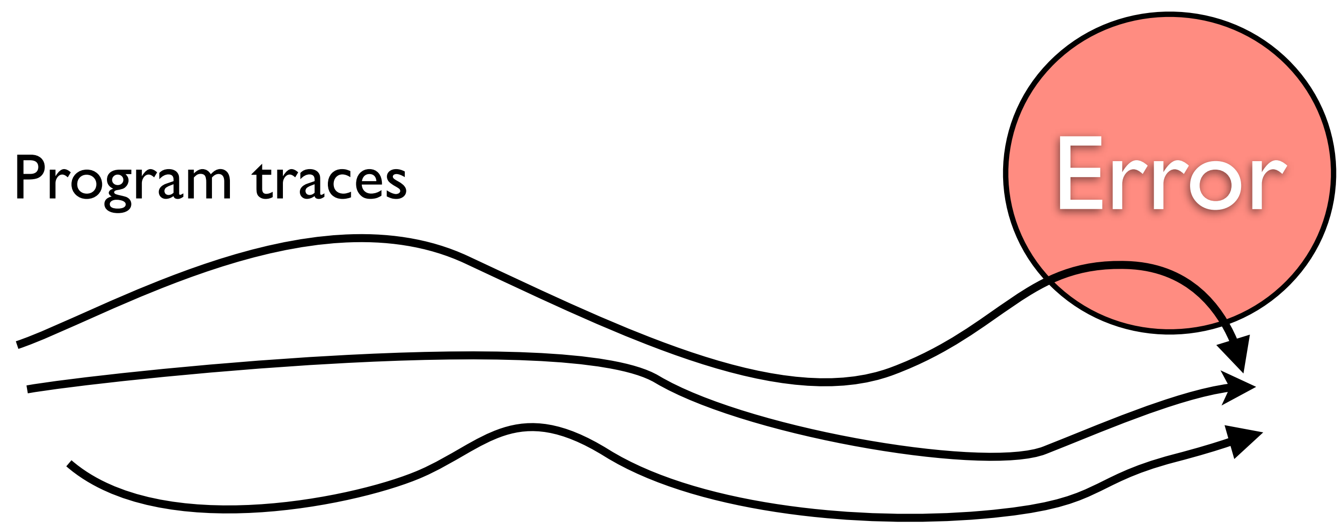
Abstract Interpretation

Decision Procedures

Scalable and efficient.
Precise analysis requires experts

Decision Procedures

- Precisely explore a large set of program traces
- For efficiency, represent problem *symbolically* as satisfiability of a logical formula

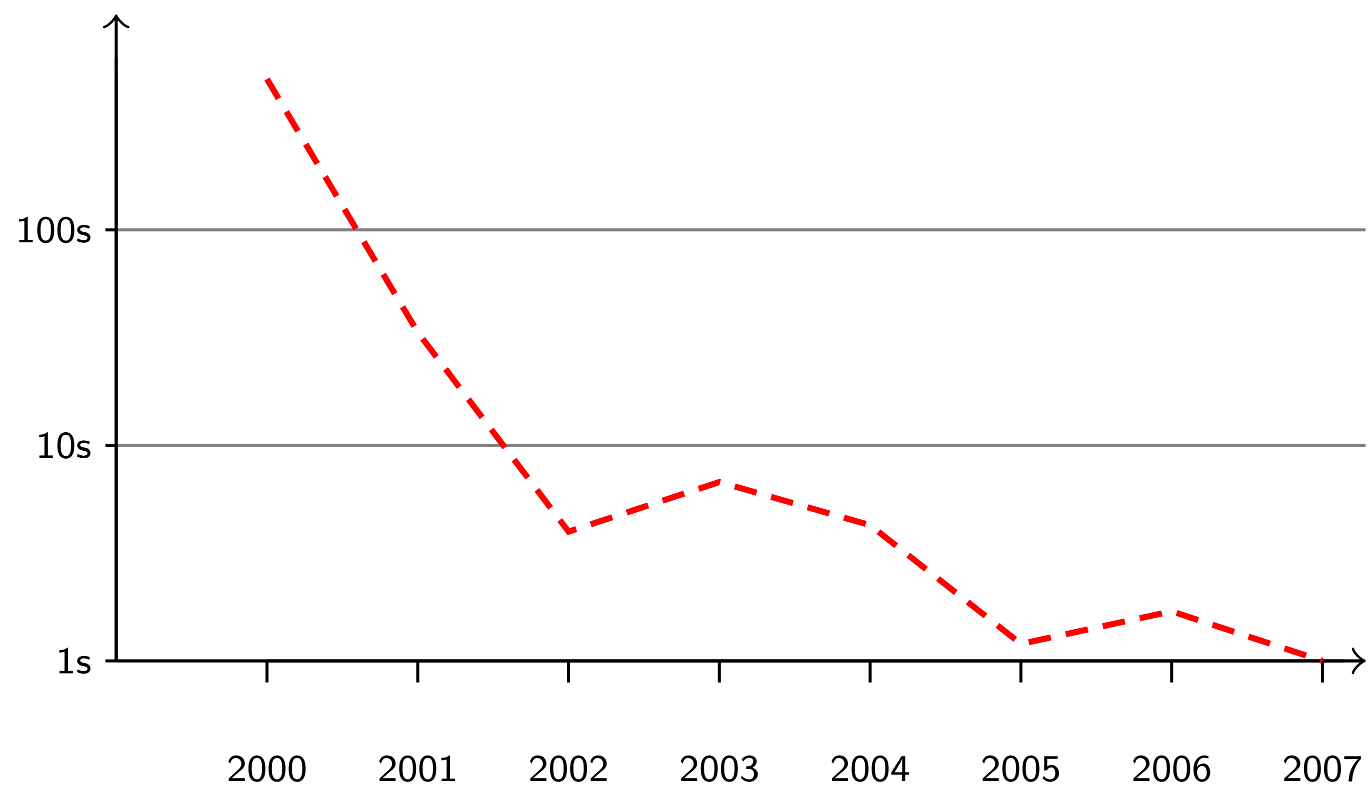


Program is safe exactly if $isTrace(t) \wedge error(t)$ is satisfied by some t

Propositional SAT

Propositional formula: $\varphi = (a \vee \neg b) \wedge (\neg a \vee b) \wedge \neg b$

Is there an assignment to a,b that makes the formula true?



Decrease in SAT solving time for SAT algorithms 2000-2007

Why are SAT solvers so efficient

Probe for solution $\xrightarrow{\text{failure}}$ Learn from failure

- SAT solvers learn from failure
- SAT solvers spot relevance

Decision Procedures

Example

```
int foo(int a, int b, bool c)
{
  int result;

  if(c)
    result = a/b;
  else
    result = a*b;

  if(a>0 && b>0)
    assert(result >= 0);

}
```

$$
\begin{array}{ll}
 & c \rightarrow (r = a /_{32} b) \\
\wedge & \neg c \rightarrow (r = a *_{32} b) \\
\wedge & a > 0 \wedge b > 0 \wedge r < 0
\end{array}
$$

Can be translated to *propositional logic* using divider and multiplier circuits

The formula evaluates to true under the following assignment:

$$
\begin{aligned}
a, b &\mapsto 123456789 \\
r &\mapsto -1757895751 \\
c &\mapsto \text{false}
\end{aligned}
$$

Counter-example!

Bounded Model Checking

Loops require unrolling
before translation

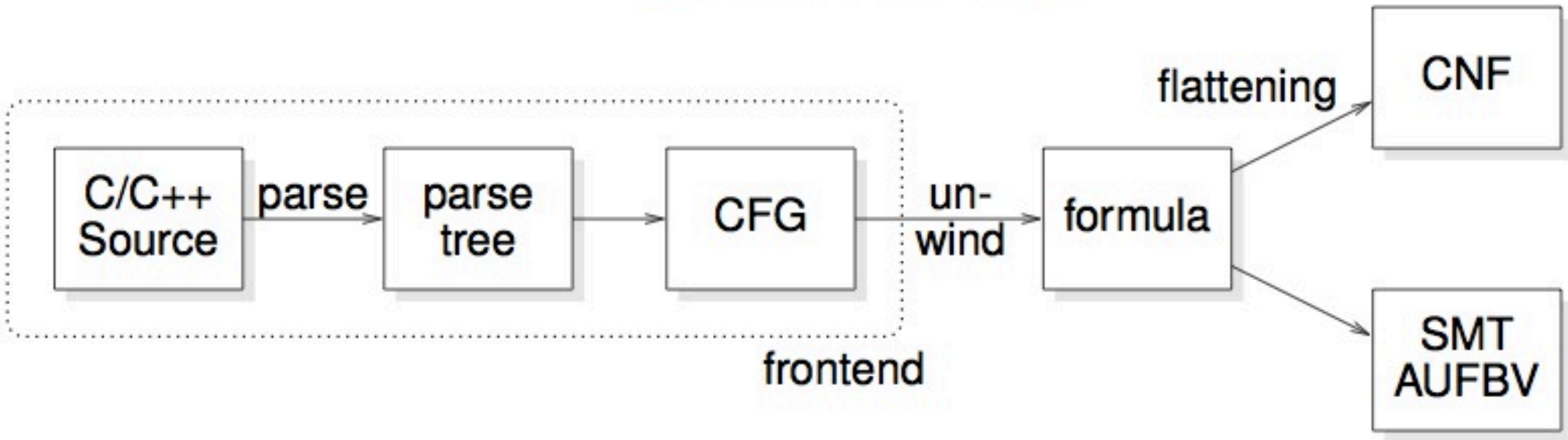
```
int foo(int *a)
{
  int sum;

  for(int i = 0; i < N; i++)
    sum+=a[i];

  assert(sum > 0);
  return sum;
}
```

```
int foo(int *a)
{
  int sum;

  int i = 0;

  if(i < N)
  {
    sum += a[i];
    if(++i < N)
    {
      sum += a[i];
      if(++i < N)
      {
        ...
      }
    }
  }

  assert(sum > 0);
  return sum;
}
```

If the loop does not have a known fixed bound,
the result is unrolled up to a chosen depth.

Bounded Model Checking

CBMC



Decision Procedure

Unsatisfiable → ?

Satisfiable → Program has bug, *counter-example* is returned

FP support in CBMC (2008)

- CBMC implements <u>bit-precise reasoning</u> over floating-point numbers using a propositional encoding
- Uses IEEE-754 semantics with support various rounding-modes
- Allows proofs of <u>complex, bit-level</u> properties

```c
int main()
{
  union {
    int i;
    float f;
  } u;

  u.f += u.i + 1;

  assert(u.i != 0);
}
```

tmp — bash — 80×24

```
leo@scythe tmp$ cbmc test.c
file test.c: Parsing
Converting
Type-checking test
Generating GOTO Program
Adding CPROVER library
Function Pointer Removal
Partial Inlining
Generic Property Instrumentation
Starting Bounded Model Checking
size of program expression: 9 assignments
simple slicing removed 1 assignments
Generated 1 VCC(s), 1 remaining after simplification
Passing problem to propositional reduction
Running propositional reduction
Solving with MiniSAT2 with simplifier
3733 variables, 16338 clauses
SAT checker: negated claim is UNSATISFIABLE, i.e., holds
Runtime decision procedure: 0.068s
VERIFICATION SUCCESSFUL
leo@scythe tmp$
```

Scalability of Propositional Encoding

- Floating-point arithmetic is flattened to propositional logic
- Requires instantiation of <u>large</u> floating point arithmetic circuits

```
for(int i = 0; i < N; i++)
{
  f *= f;
}
```

N	Nr. Variables	Memory use
5	~130000	~90MB
10	~260000	~180MB

- Resulting formulas are hard for SAT solvers and take up large amounts of memory

Mixed Abstractions for Floating Point Arithmetic (2009)

- Use propositional abstraction to increase efficiency and ease memory requirements

- Novel *mixed abstraction framework*
  - Over-approximations allow more behaviours: Reduce the initial number of variables. Eases memory requirements and improves efficiency.
  - Under-approximations restrict behaviours: Allows us to quickly identify solutions.

- Integrated with CBMC and the Boolector SMT solver

Mixed Abstractions for Floating Point Arithmetic (2009)

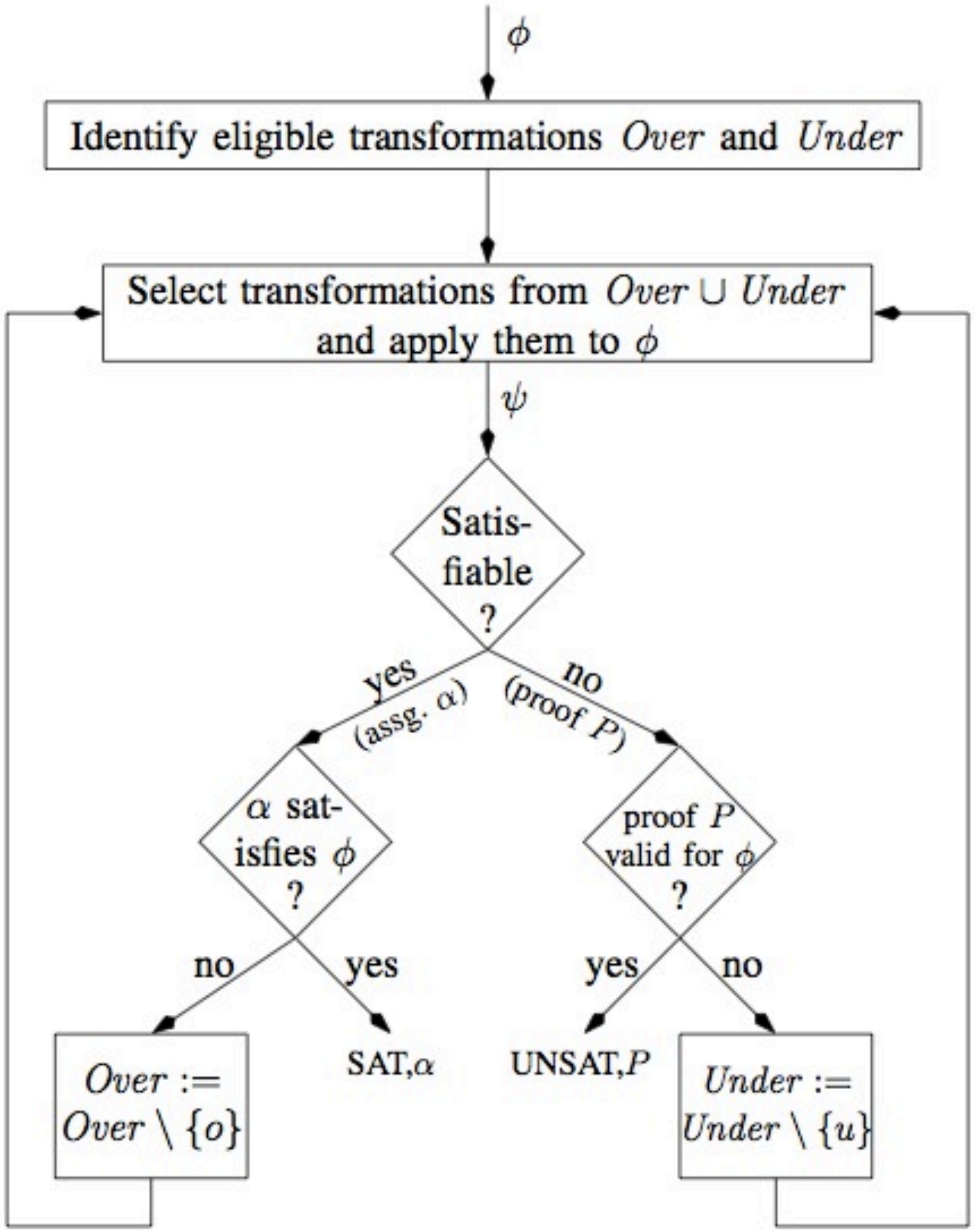


Fig. 4. The Framework of Mixed Abstraction

Benchmark	Lines of Code	Satis-fiable?	No Abstr. time (s)	Mixed time (s)	Mixed #iter.
qurt.c, claim 1	109	no	25	2	15
qurt.c, claim 2	109	no	25	0.6	7
qurt.c, claim 3	109	no	25	1	13
qurt.c, claim 4	109	no	OM	478	103
qurt.c, claim 5	109	no	25	1.2	15
qurt.c, claim 6	109	no	25	0.6	7
qurt.c, claim 7	109	no	6716	84	86
sqrt.c, claim 1	51	no	24	13589	44
sqrt.c, claim 2	51	yes	9	TO	107
minver.c, claim 1	156	no	1	0.1	1
minver.c, claim 2	156	yes	2	0.1	1
sin.c, claim 1	46	no	13864	281	47
sin.c, claim 2	46	no	13831	281	47
sin.c, claim 3	46	no	TO	1074	63
gaussian.c, claim 1	108	no	TO	14437	137

Related work

Constraint satisfaction

C. Michel, M. Rueher and Y. Lebbah: Solving constraints over floating-point numbers. CP2001

B. Botella, A. Gotlieb and C. Michel: Symbolic execution of floating-point computations. STVR2006

SMT

P. Ruemmer and T. Wahl. An SMT-LIB theory of binary floating-point arithmetic. SMT 2010

A. Brillout, D. Kroening and T. Wahl. Mixed abstractions for floating point arithmetic. FMCAD 2009

R. Brummayer and A. Biere. Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays. TACAS 2009

Incomplete Solvers

S. Boldo, J.-C. Filliâtre and G. Melquiond. Combining Coq and Gappa for Certifying Floating-Point Programs. Calculemus 2009.

Requires experts,
scalable, precise

Manual

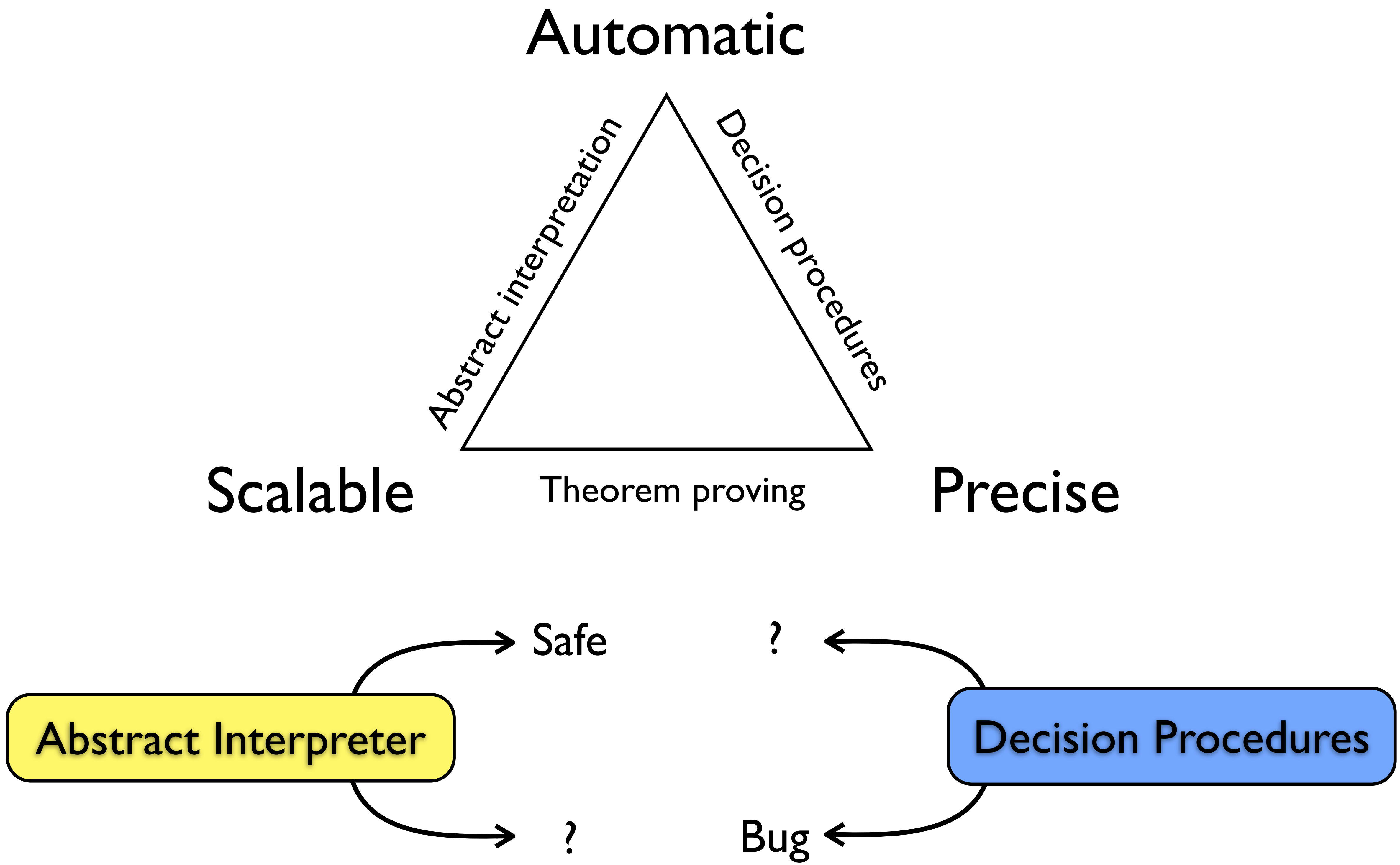
Abstract Interpretation

Scalable.
Precision requires experts

Decision Procedures

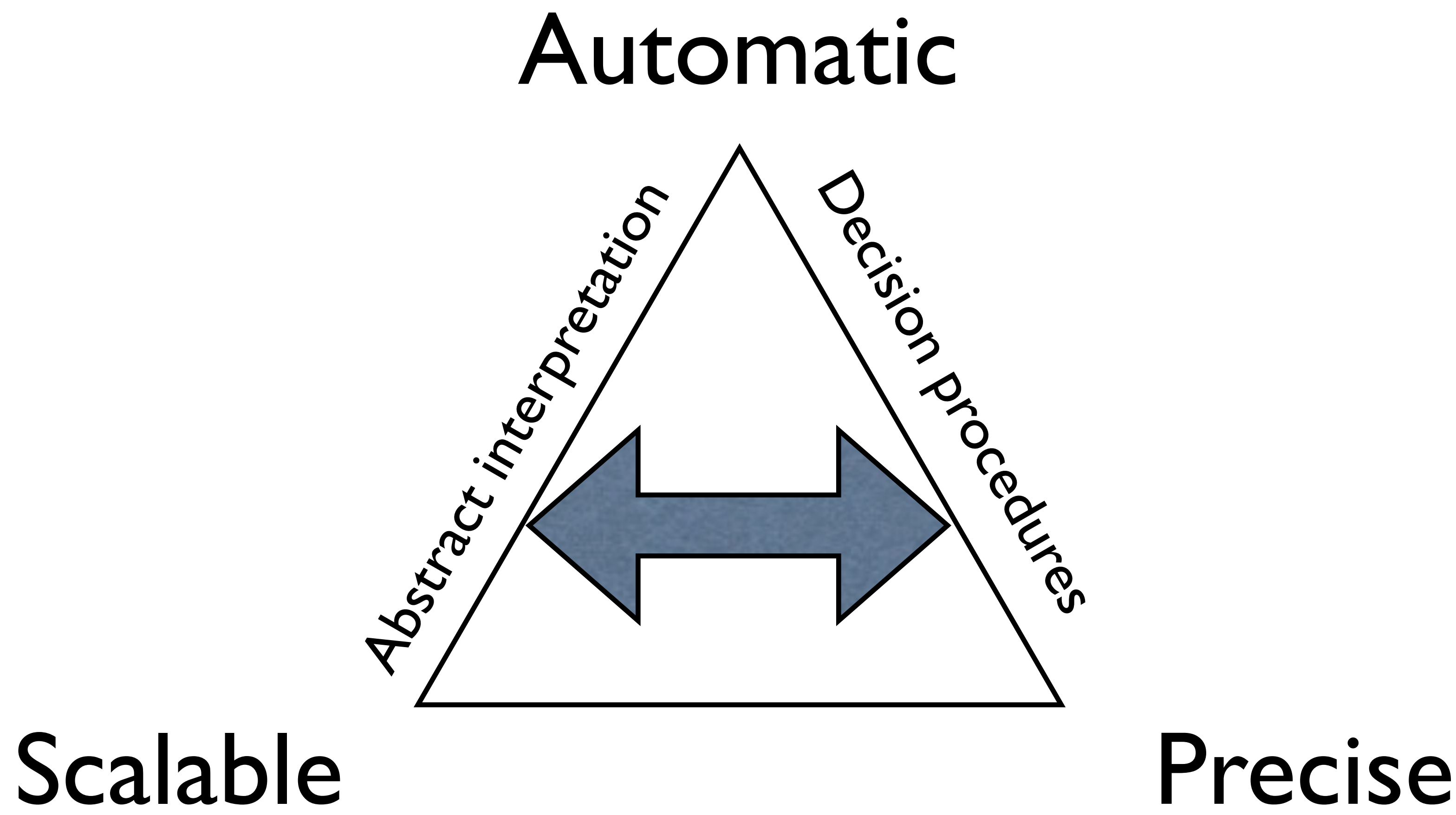
Precise.
Scalability requires experts

Conclusion Part I



Questions so far?

Part II



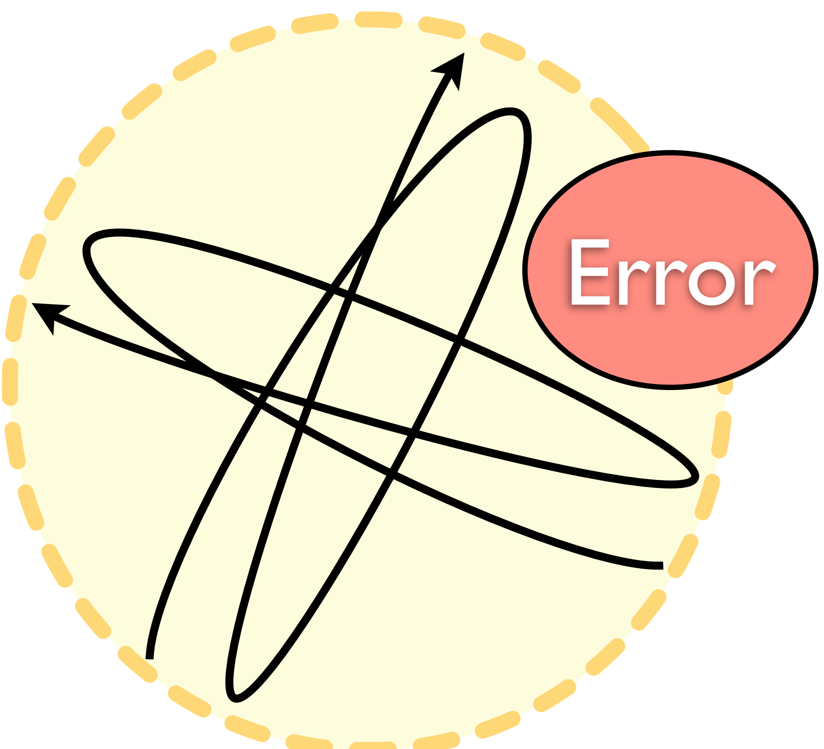
We are interested in techniques that are

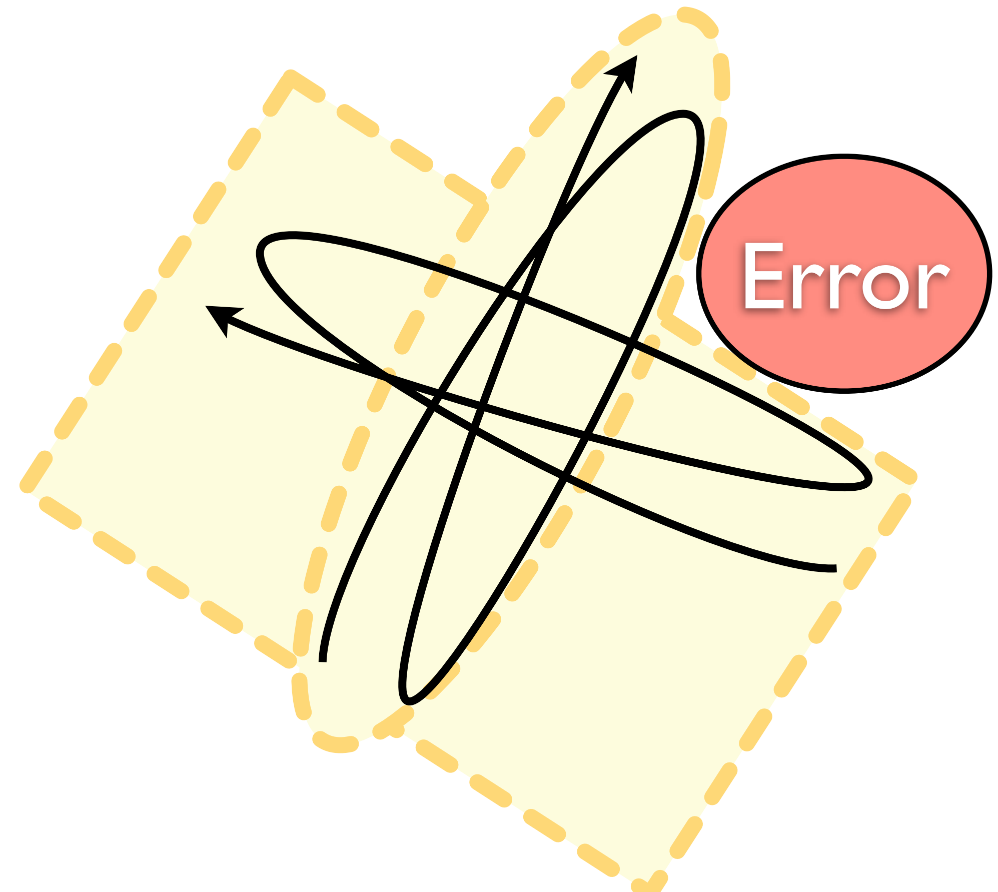
- scalable
- sufficiently precise to prove safety
- fully automatic

Central insight:

Modern decision procedure *are* abstract interpreters!

Manually adjusting analysis precision by abstract partitioning






```
void foo(int x)
{

  int y;

  if(x < 0)
    y = 1;
  else
    y = -1;
```

$y \in [-1, 1]$

```
  assert(y != 0);

}
```

Potentially unsafe!

```
void foo_precise(int x)
{
  if(x < 0)
    foo(x)
  else
    foo(x);
}

void foo(int x)
{
  ...
}
```

Safe!

How do we find the partition automatically?

SAT solving by example

SAT solvers accept formulas in conjunctive normal form

literals

$$\varphi = \underbrace{(p \vee \neg q)} \wedge \ldots \wedge \underbrace{(\neg r \vee w \vee q)}$$

clauses

Their main data structure is a <u>partial</u> variable assignment which represents a solution candidate

$$V \to \{\mathsf{t}, \mathsf{f}\}$$

SAT solving: Deduction

$$\varphi = p \wedge (\neg p \vee \neg q) \wedge (q \vee r \vee \neg w) \wedge (q \vee r \vee w)$$

SAT deduces new facts from clauses:

$\Longrightarrow$ $p \mapsto \mathsf{t}$ $\Longrightarrow$ $p \mapsto \mathsf{t}$, $q \mapsto \mathsf{f}$

At this point, clauses yield no further information

SAT is Abstract Analysis: Deduction

$$\varphi = p \wedge (\neg p \vee \neg q) \wedge (q \vee r \vee \neg w) \wedge (q \vee r \vee w)$$

$$p \mapsto \mathsf{t}$$

$$p \mapsto \mathsf{t}$$
$$q \mapsto \mathsf{f}$$

```
void foo(void)
{
  bool p, q, r, w;

  if(p)
   if(!p || !q)
    if(q || r || !w)
     if(q || r || w)
      assert(0);
}
```

$p \in [1, 1]$

$q \in [0, 0]$

The result of deduction is <u>identical</u> to applying interval analysis to the program:

Deduction in a SAT solver <u>is</u> abstract analysis

SAT solving: Decisions

$$\varphi = p \wedge (\neg p \vee \neg q) \wedge (q \vee r \vee \neg w) \wedge (q \vee r \vee w)$$

SAT solver makes a "guess"

Pick an unassigned variable and assign a truth value

$$p \mapsto \mathsf{t} \qquad\qquad p \mapsto \mathsf{t}$$

$$q \mapsto \mathsf{f} \quad \Longrightarrow \quad q \mapsto \mathsf{f}$$

$$r \mapsto \mathsf{f}$$

Now new deductions are possible

SAT solving: Learning

$$\varphi = p \wedge (\neg p \vee \neg q) \wedge (q \vee r \vee \neg w) \wedge (q \vee r \vee w)$$

$$p \mapsto \mathsf{t}$$

$$q \mapsto \mathsf{f}$$

$$r \mapsto \mathsf{f}$$

The variable *w* would have to be both true and false.

The contradiction is the result of *r* being assigned to false as part of a decision. The SAT solver therefore learns that *r* must be true:

$$\varphi \leftarrow \varphi \wedge r$$

SAT solving: Learning

$$\varphi = p \wedge (\neg p \vee \neg q) \wedge (q \vee r \vee \neg w) \wedge (q \vee r \vee w)$$

$$\begin{array}{l} p \mapsto \mathsf{t} \\ q \mapsto \mathsf{f} \\ r \mapsto \mathsf{f} \end{array} \Longrightarrow \begin{array}{l} p \mapsto \mathsf{t} \\ q \mapsto \mathsf{f} \\ r \mapsto \mathsf{f} \\ w \mapsto \mathsf{f} \end{array} \Longrightarrow \text{conflict}$$

The variable *w* would have to be both true and false.

The contradiction is the result of *r* being assigned to false as part of a decision. The SAT solver therefore learns that *r* must be true:

$$\varphi \leftarrow \varphi \wedge r$$

SAT is Abstract Analysis: Decisions & Learning

$\varphi \longrightarrow \varphi \wedge r$

```c
void foo(void)
{
  bool p, q, r, w;

  if(p)
   if(!p || !q)
    if(q || r || !w)
     if(q || r || w)
      assert(0);
}
```

```c
void foo_precise()
{
  if(r)
    foo();
}

void foo()
{
  ...
}
```

Decisions and learning in a SAT solver <u>are</u> abstract partitioning

SAT is Abstract Analysis

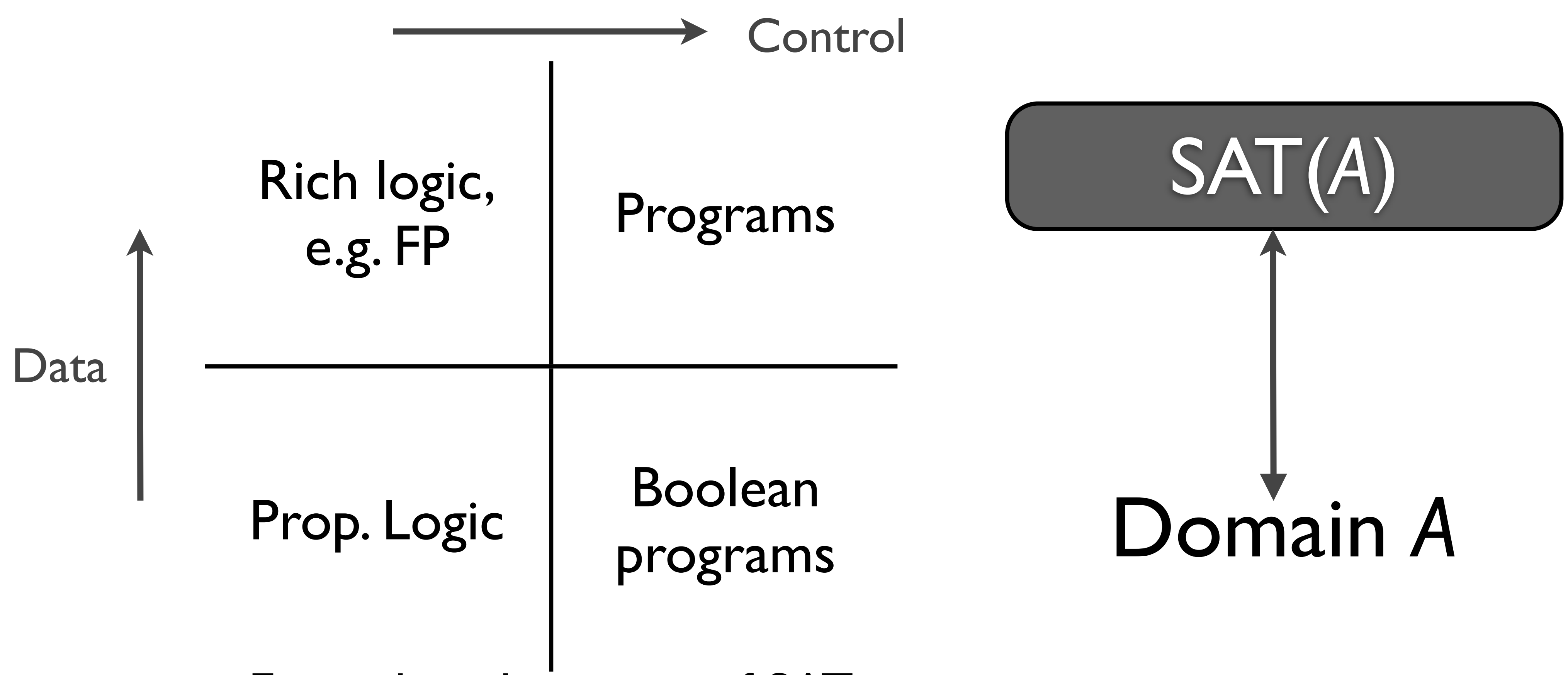
- Deduction in SAT is abstract interpretation
- Decisions and learning are abstract partitioning
- The SAT algorithm is really an automatic partition refinement algorithm.

SAT(A)

Domain A

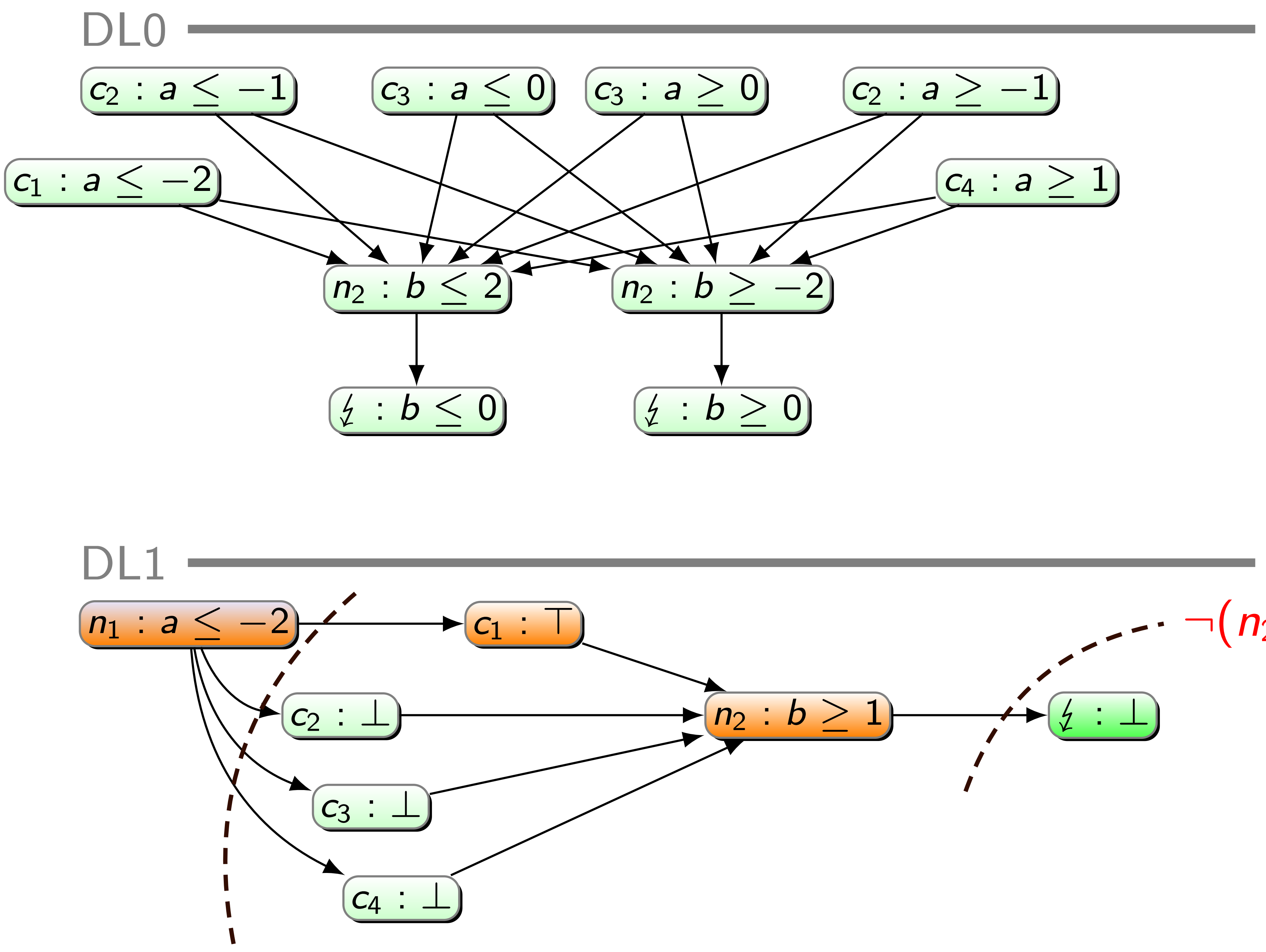
SAT is Abstract Analysis

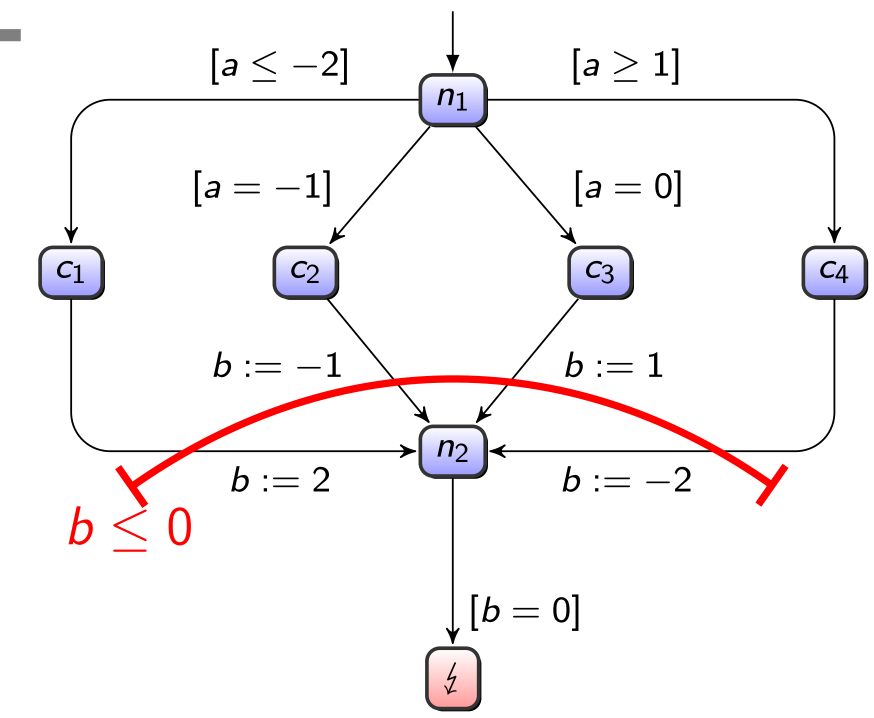
- Deduction in SAT is abstract interpretation
- Decisions and learning are abstract partitioning
- The SAT algorithm is really an automatic partition refinement algorithm.



Expanding the scope of SAT

SAT for programs





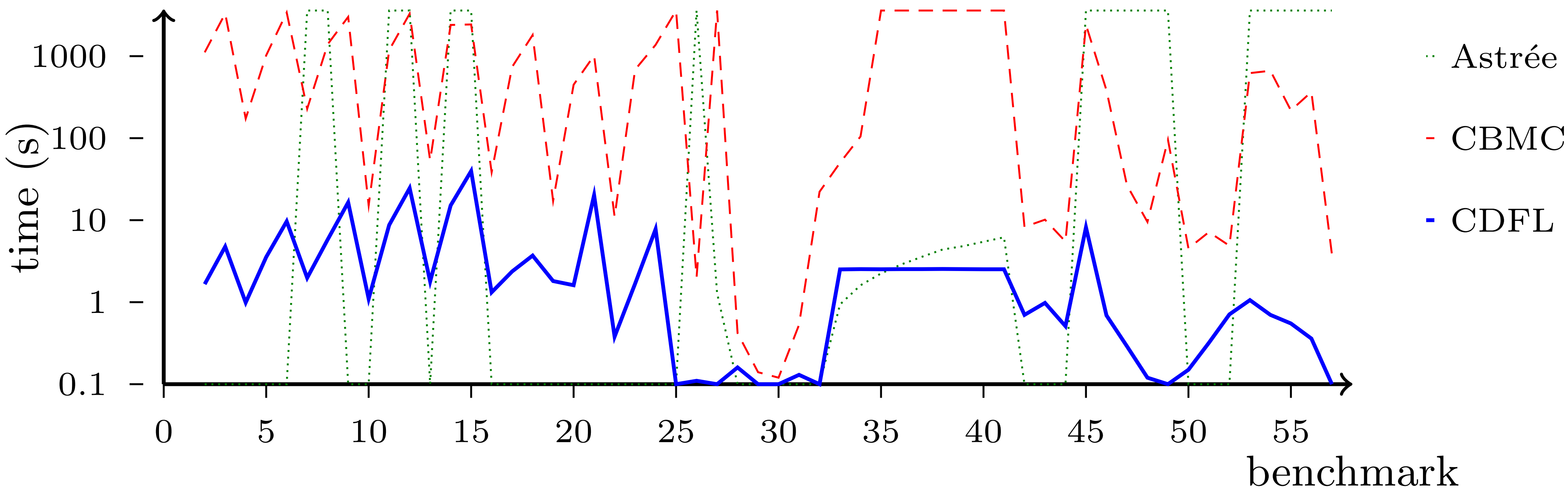
SAFE $\rightarrow$ find cut

Prototype: Abstract Conflict Driven Learning (ACDL)

- Implementation over floating-point intervals
- Automatically refines an analysis in a way that is
  - Property dependent
  - Program dependent
- Uses <u>learning</u> to intelligently explore partitions
- <u>Significantly more precise</u> than mature abstract interpreters
- <u>Significantly more efficient</u> than floating-point decision procedures on short non-linear programs

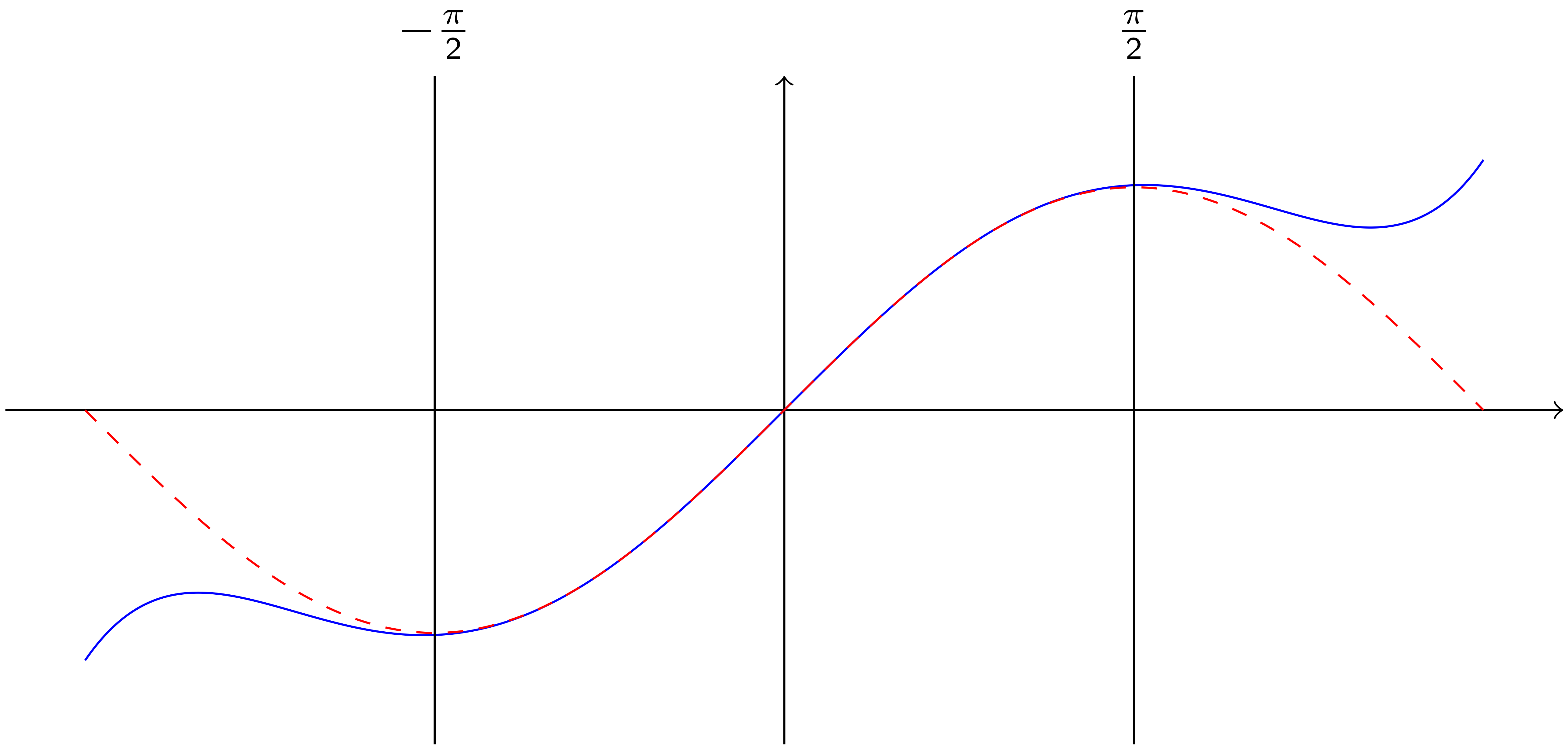
Demo

More results

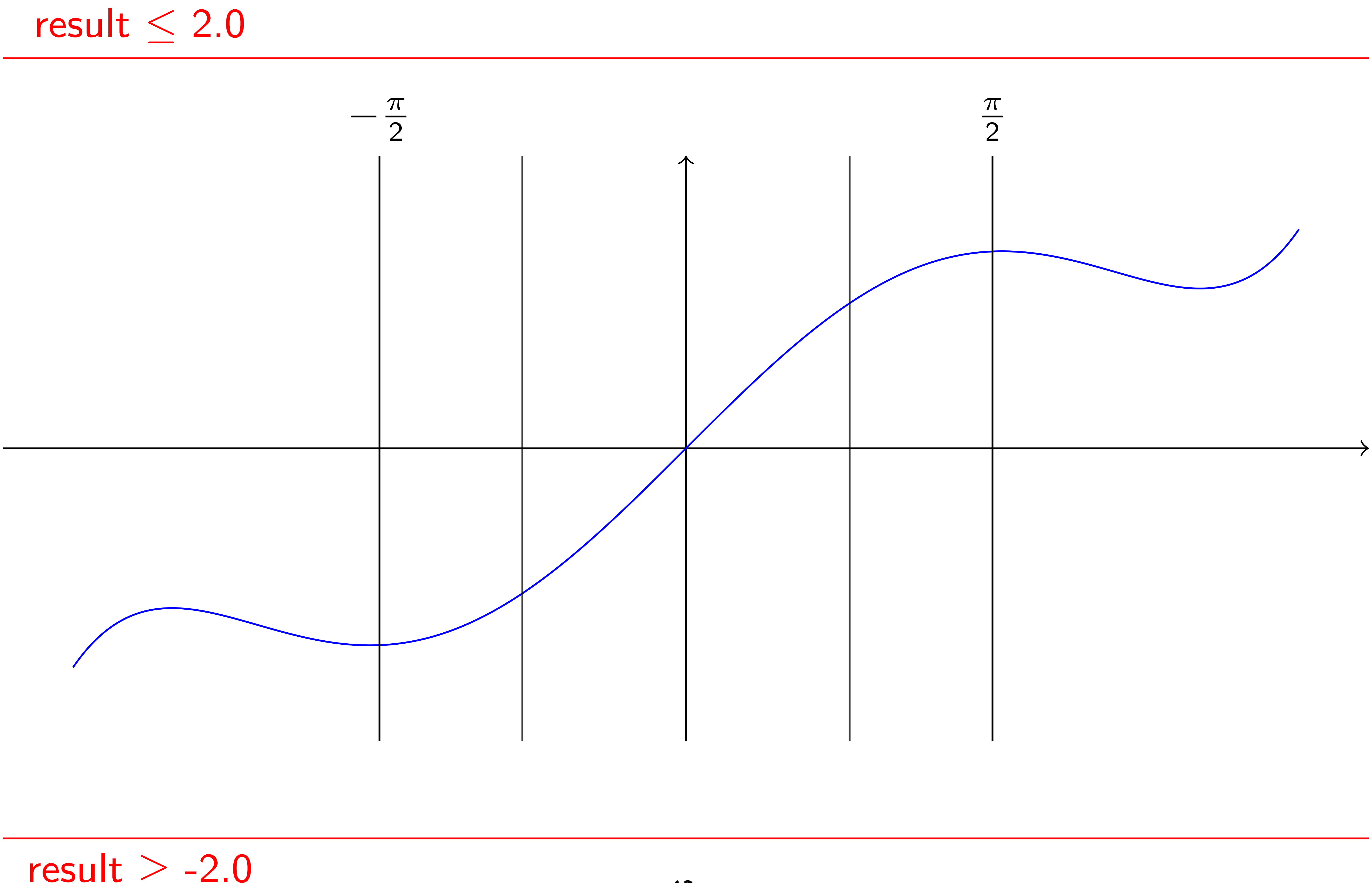


Average speedup over CBMC ~270x

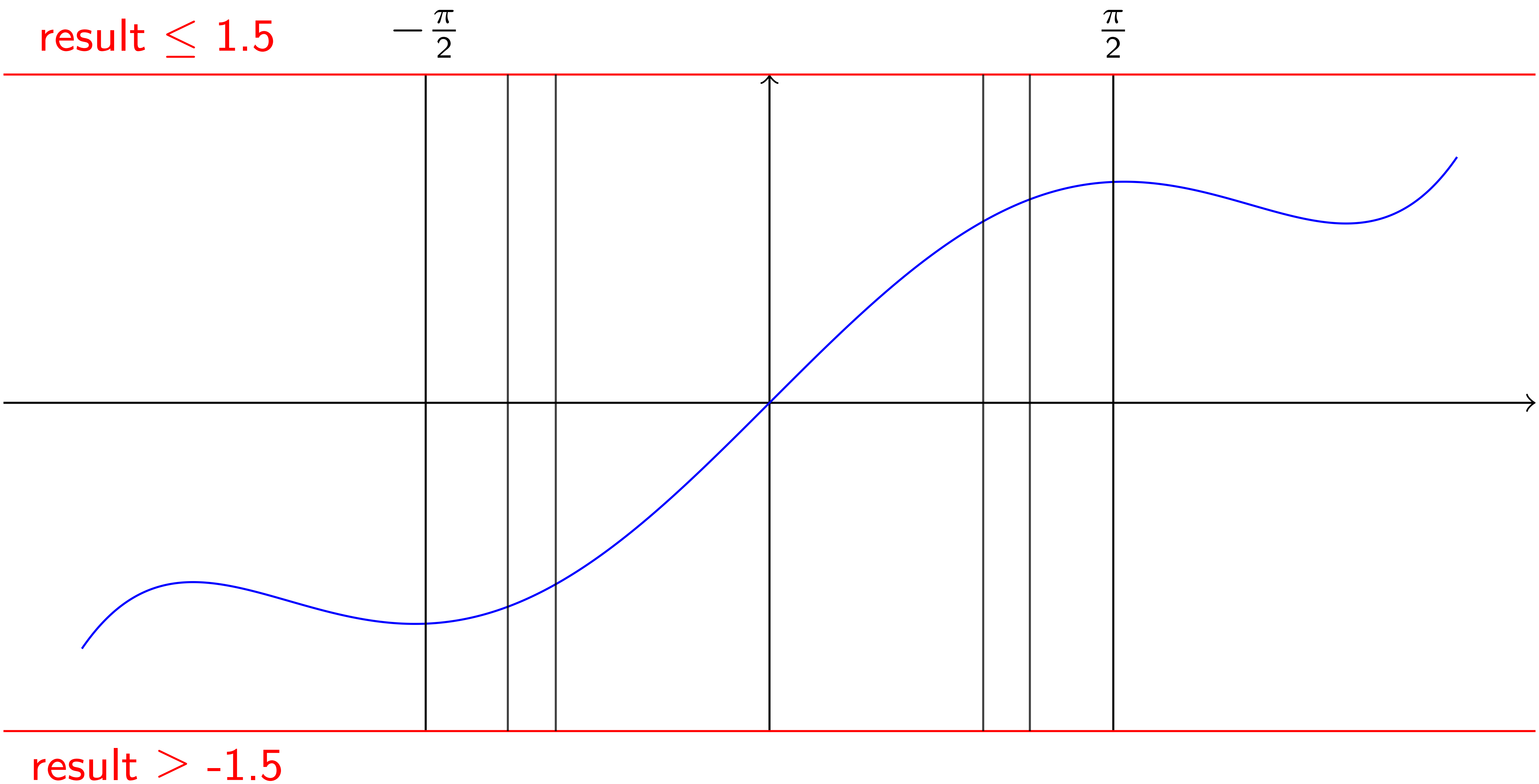
Implementation



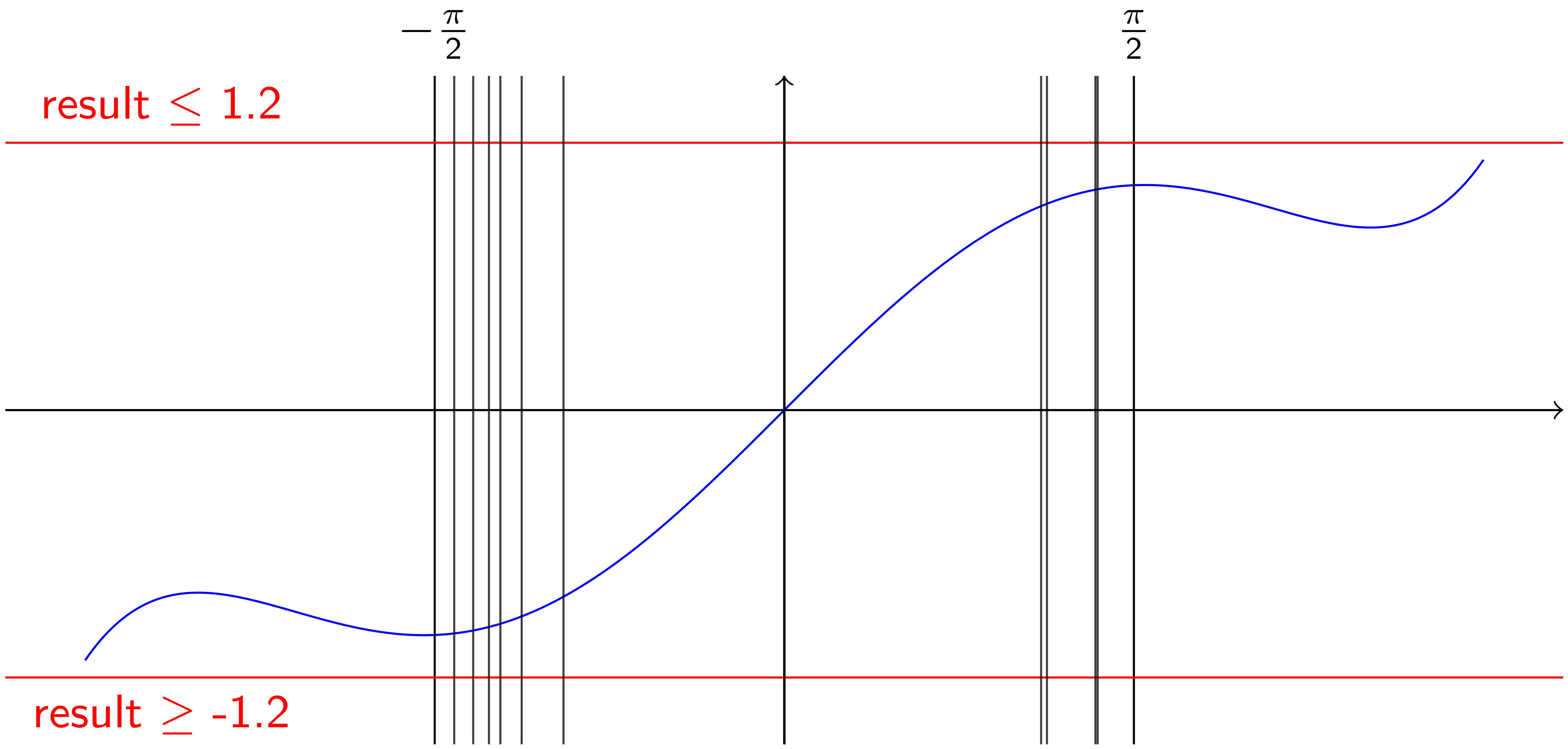
Number of partitions vs. tightness of bound



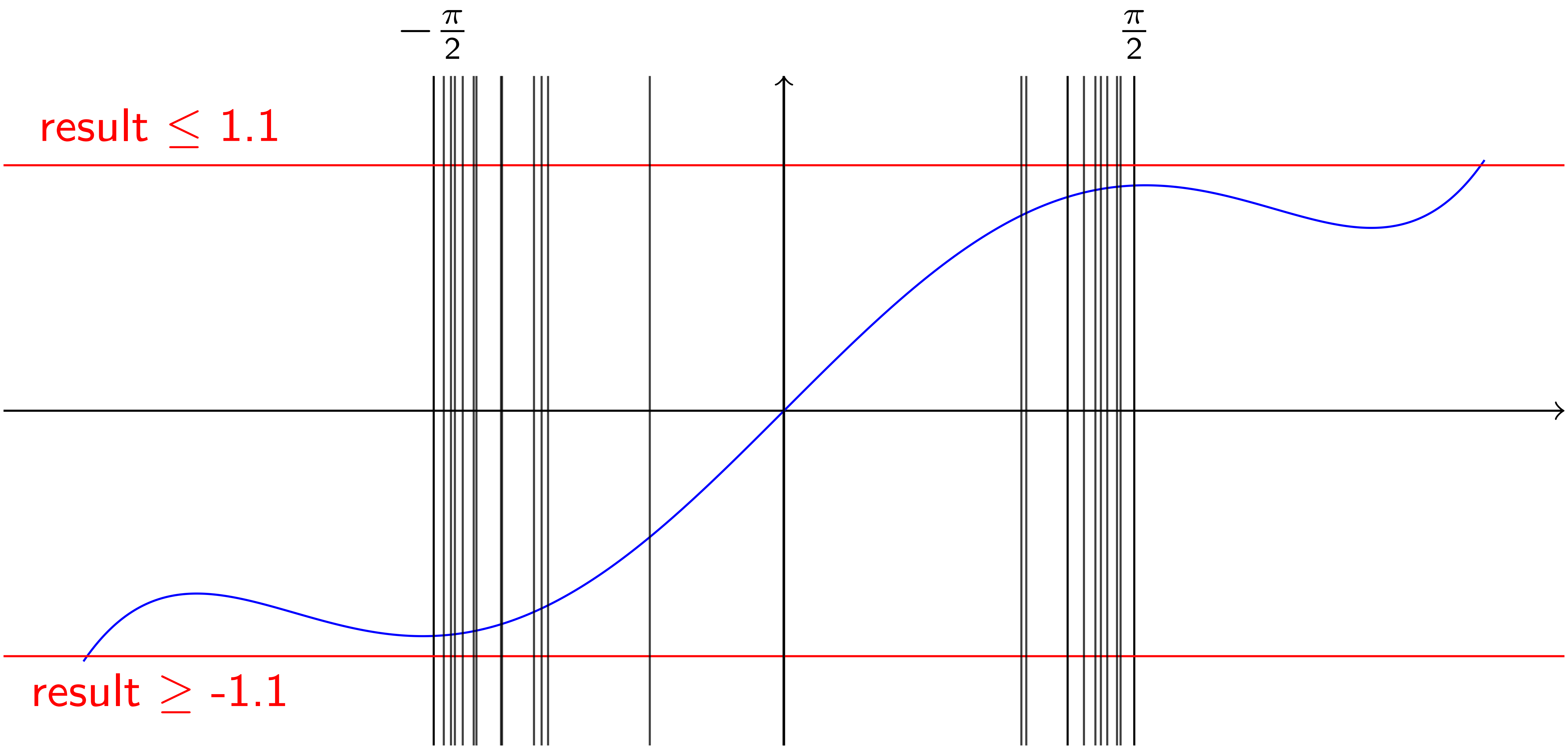
Number of partitions vs. tightness of bound



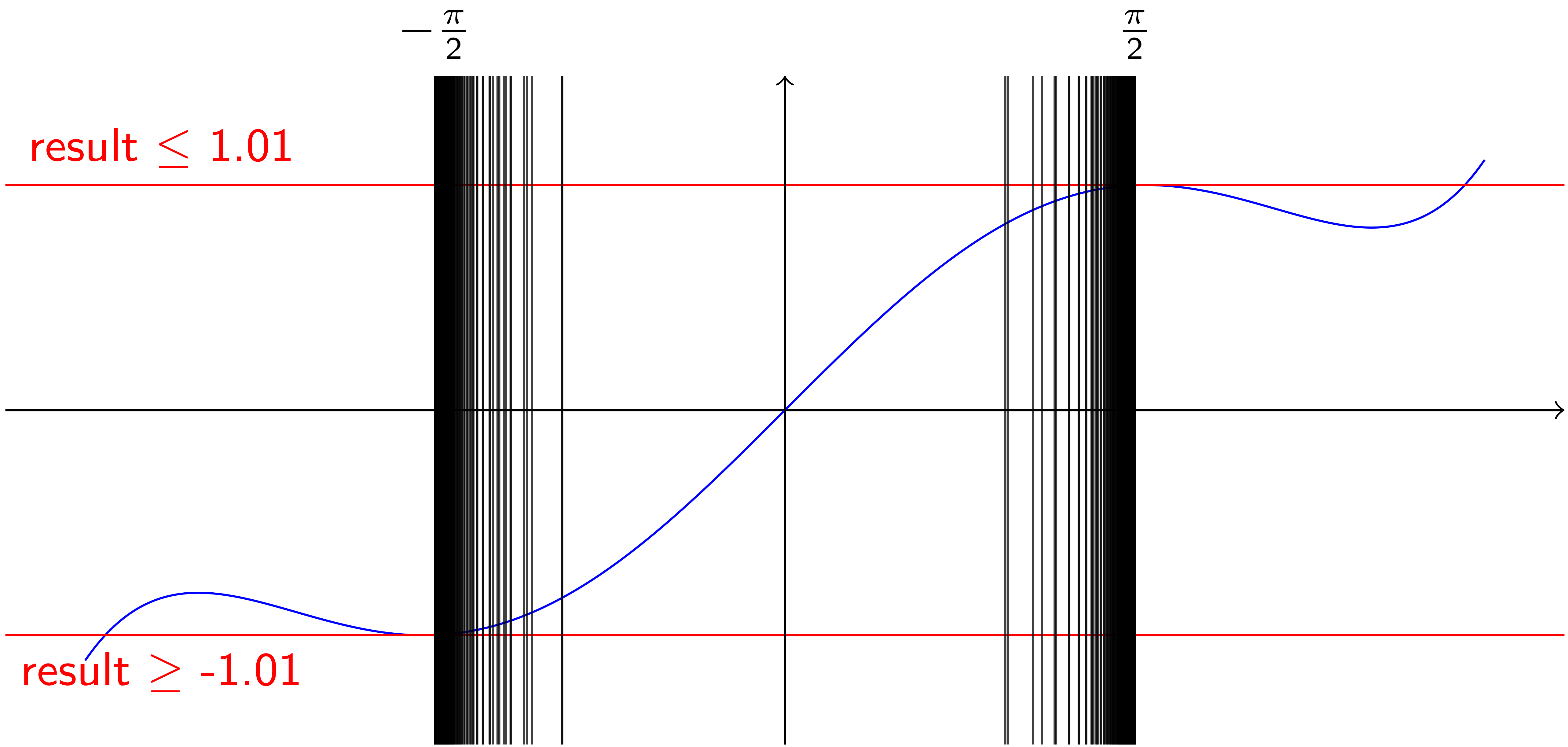
Number of partitions vs. tightness of bound



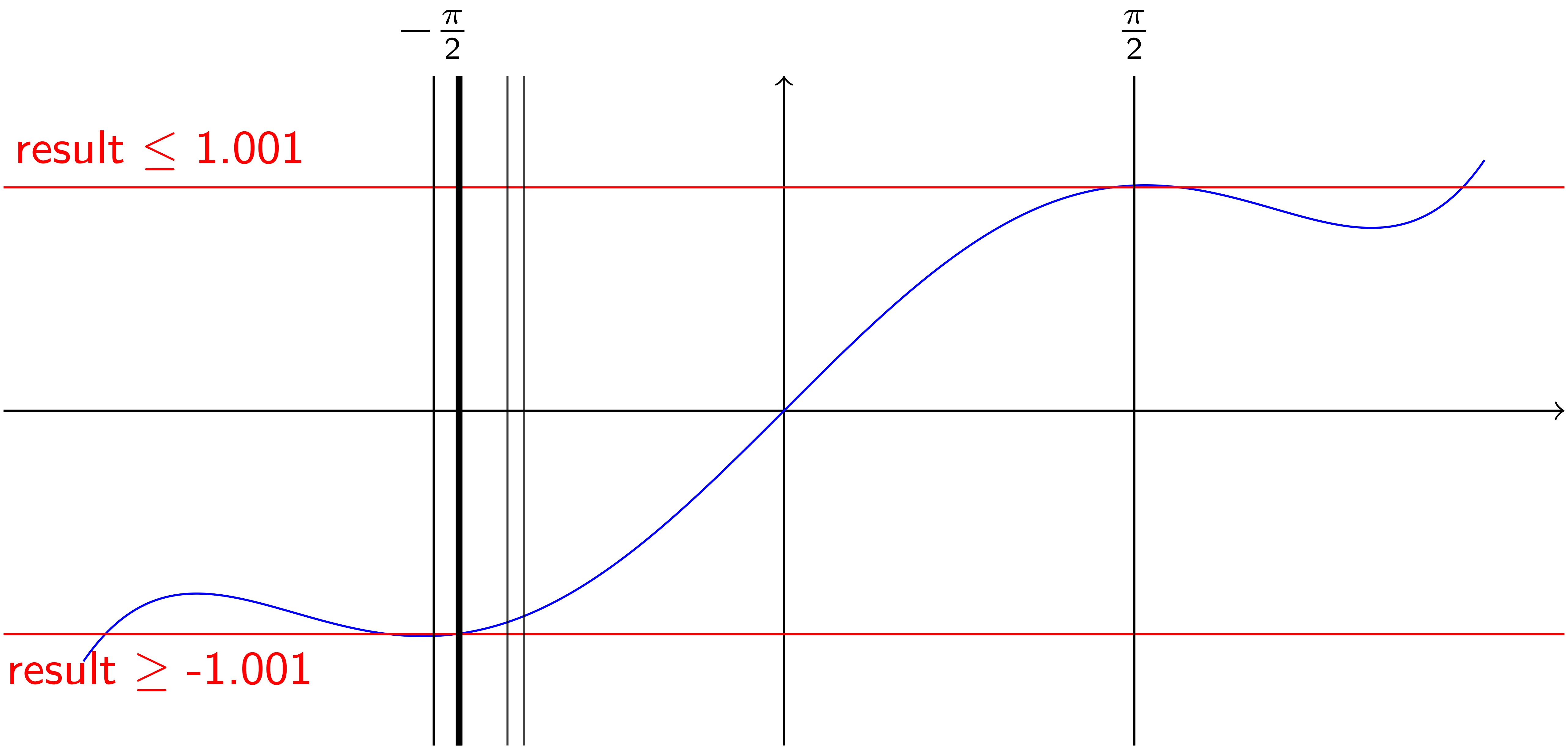
Number of partitions vs. tightness of bound



Number of partitions vs. tightness of bound

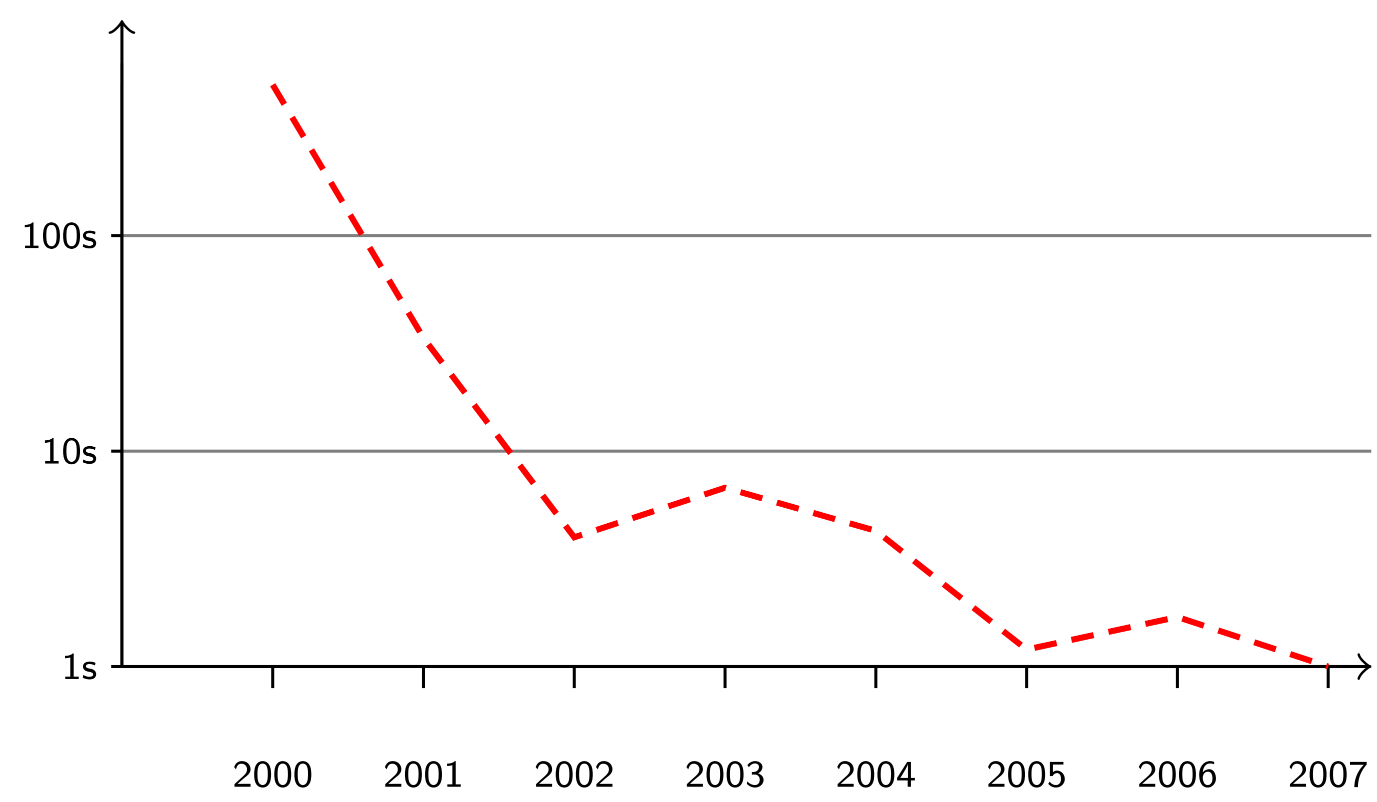


Number of partitions vs. tightness of bound



Current and Future Work

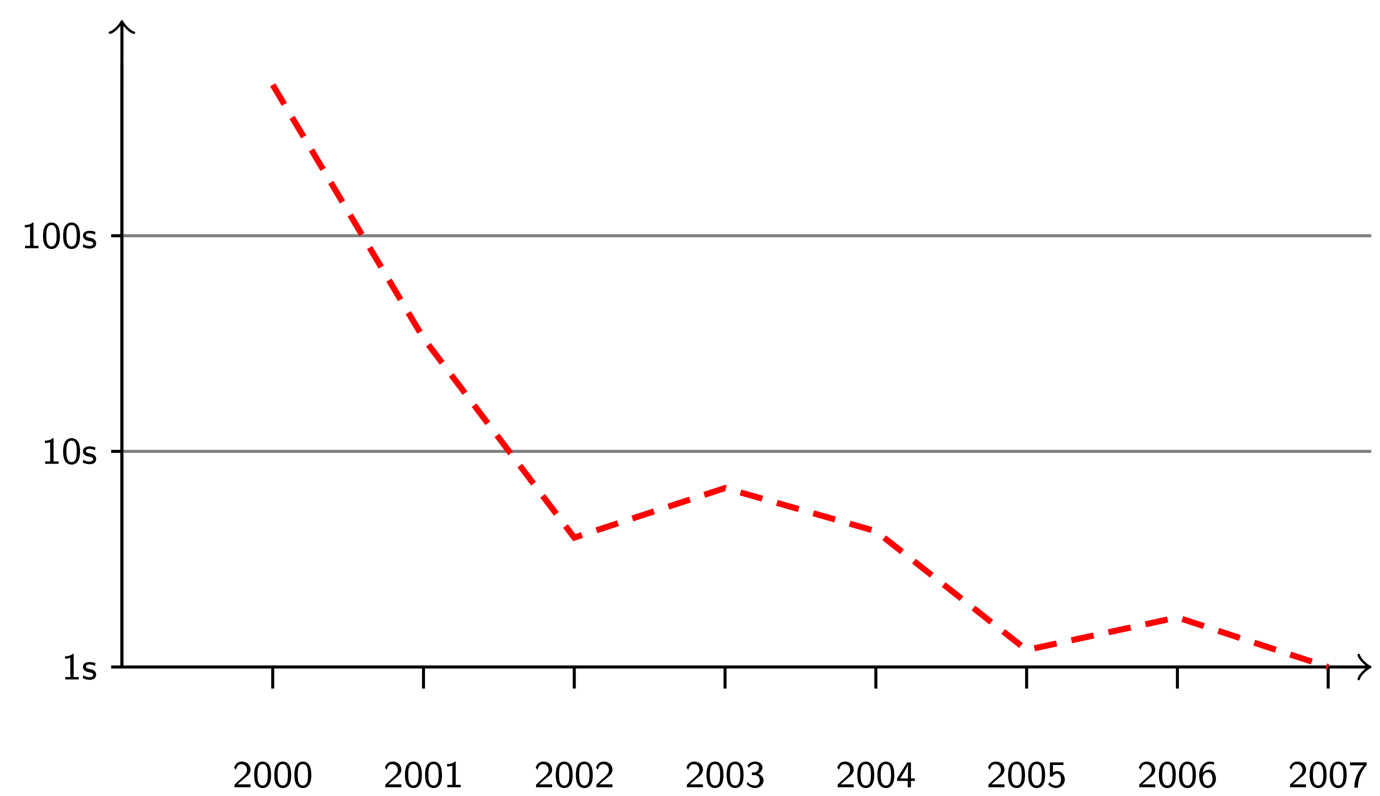
- Develop an SMT solver for floating point logic
- Model on the success of propositional SAT:
  - Simple abstract domain
  - Highly efficient data structures



Current and Future Work

- Develop an SMT solver for floating point logic
- Model on the success of propositional SAT:
  - Simple abstract domain
  - Highly efficient data structures

Rich logic, e.g. FP	Programs
Prop. Logic	Boolean programs

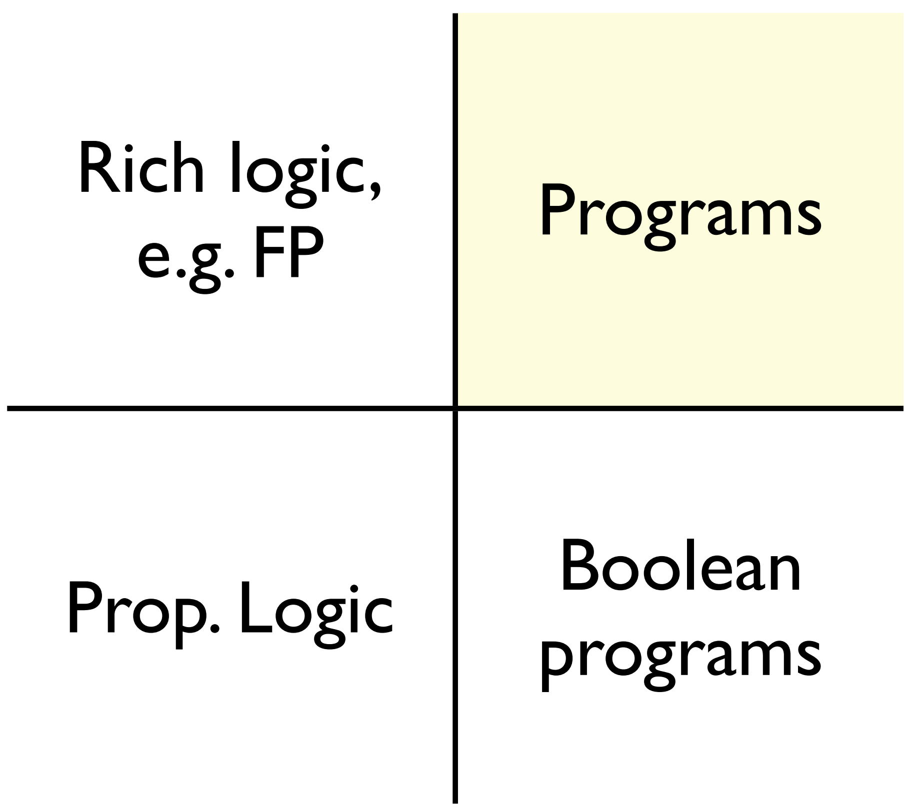


Current and Future Work

- Reengineer prototype into a tool for floating point verification
  - Significantly improved efficiency
  - Generic interface for integrating abstract domains
  - Development and generalisation of heuristics and learning strategies

Current and Future Work

- Reengineer prototype into a tool for floating point verification
  - Significantly improved efficiency
  - Generic interface for integrating abstract domains
  - Development and generalisation of heuristics and learning strategies



Refining loops with ACDL

- Currently, loops cause imprecision in our analysis
- Analysis may fail to prove safety

```
void foo()
{
  int i = -1;

  while(*)
    i *= -1;

  assert(i != 0);
}
```

Successful analysis requires distinguishing between even and odd numbers of loop iterations

Refining loops with ACDL

- Solution: Apply the SAT algorithm to control flow itself
  - Make decisions over control-flow (e.g., assume odd number of loop iterations)
  - Learning permanently alters control flow

- Resulting analysis can dynamically vary precision from full abstraction to precise case exploration

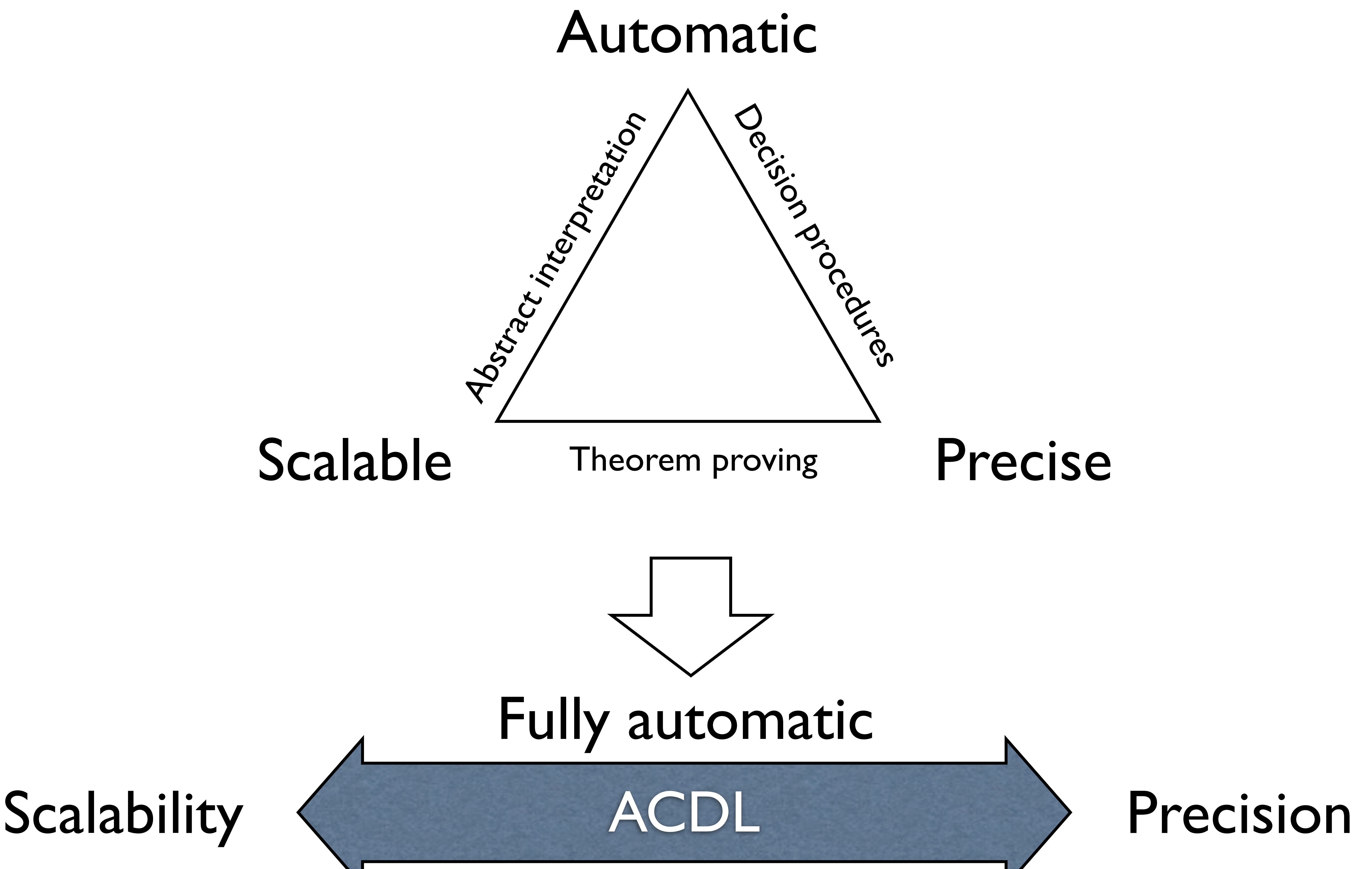
Full abstraction

Exhaustive testing

Maximal efficiency

Maximal precision

Conclusion - Part II



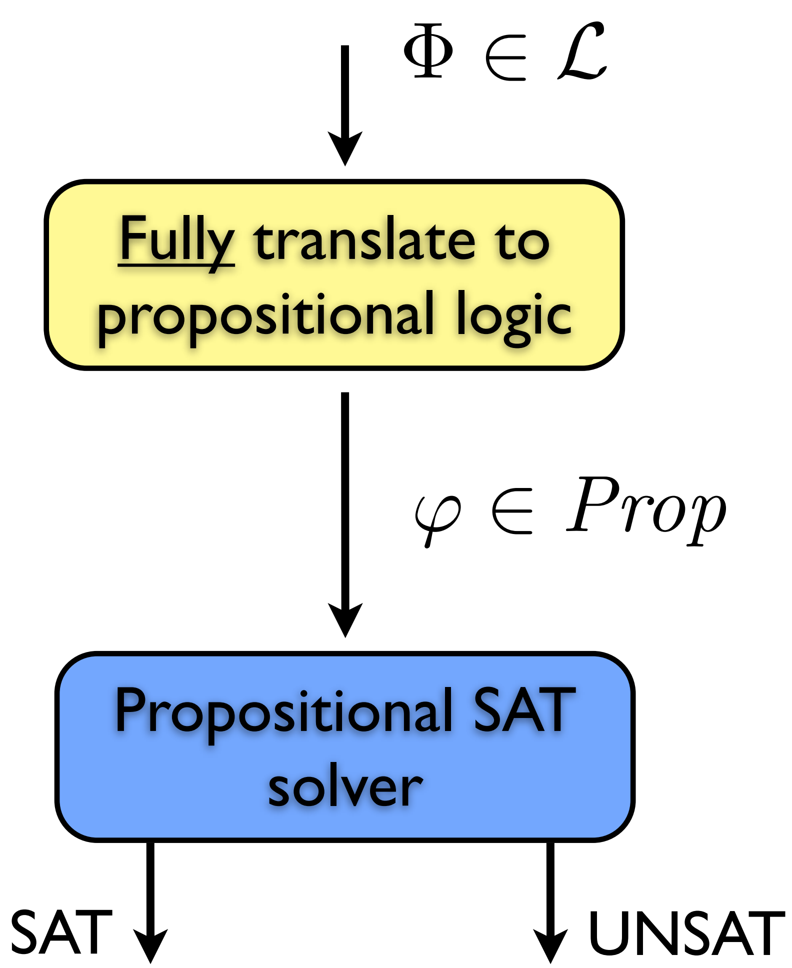
Thank you for your attention

Additional slides

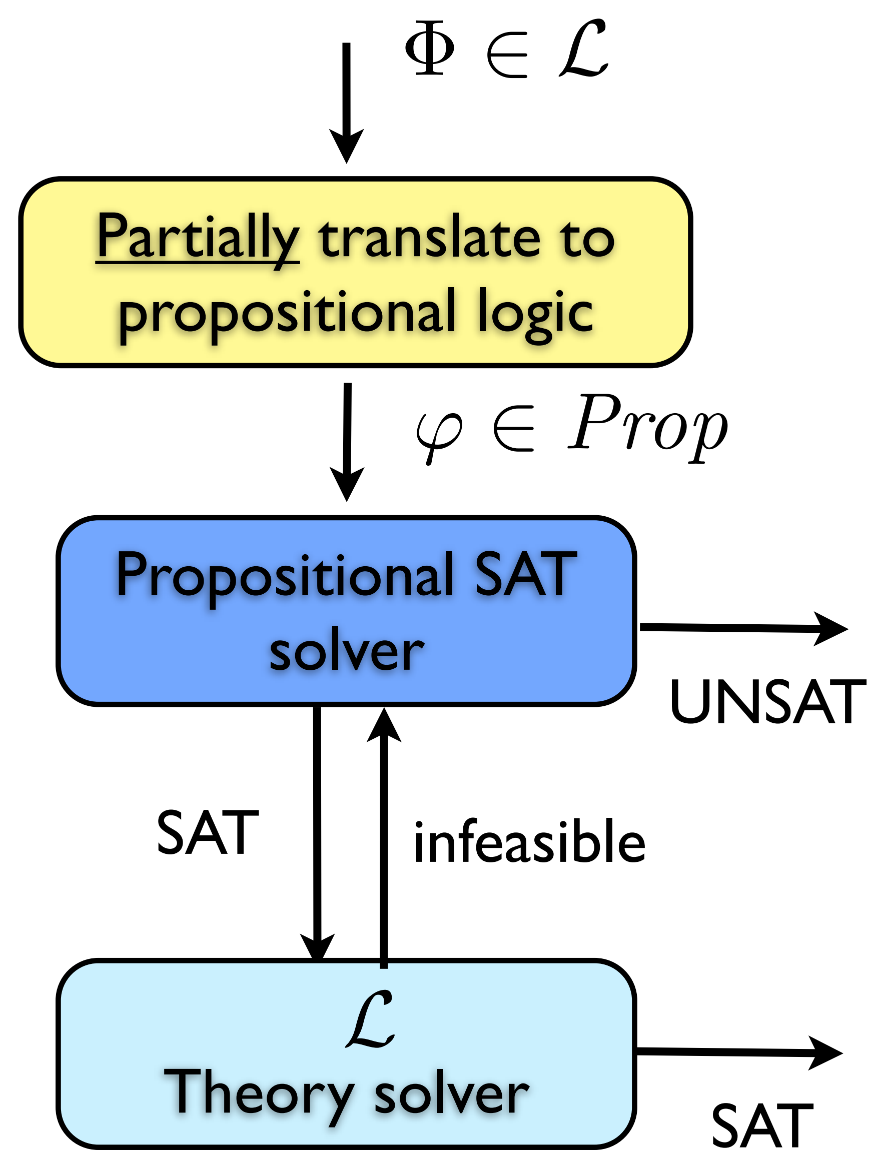
Lazy and eager SMT

Two approaches to lift SAT to a richer logic $\mathcal{L}$

Eager approach

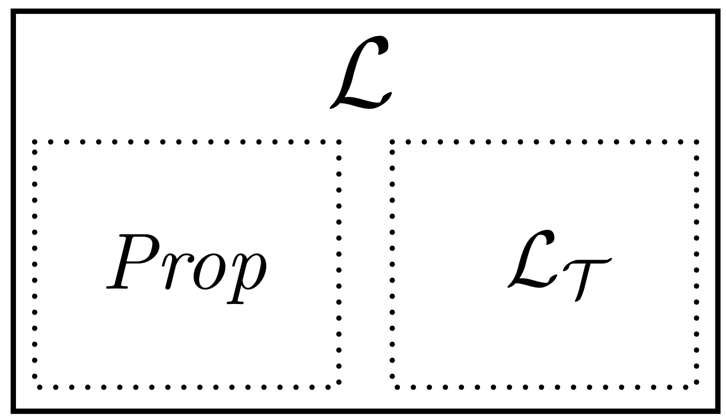


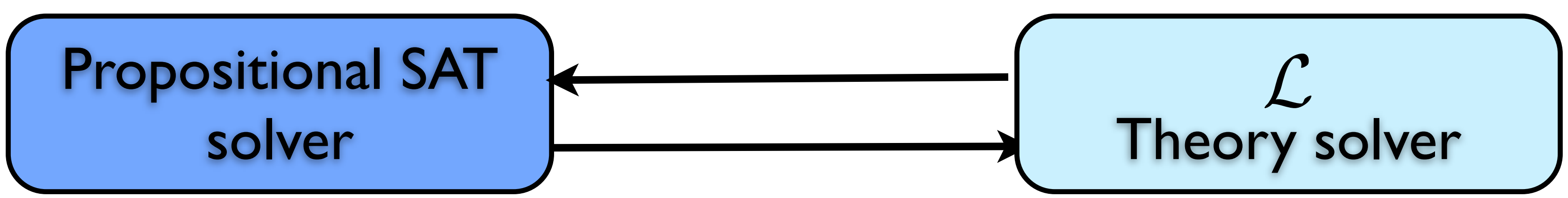
Lazy approach



Limits of lazy SMT for FP

Lazy SMT works if the logic can be decomposed into an efficiently solvable theory component and a propositional component.





The approach breaks down if significant communication is necessary between the two.

Due to the non-numeric behaviour in floating-point arithmetic such as rounding, special values, etc., there is no clear decomposition. Therefore, analysis is often performed over the real numbers instead, which may lead to unsound results.

Collected references

References

Axiomatisations of FP

M. Daumas, L. Rideau and L. Théry. A Generic Library for Floating-Point Numbers and Its Application to Exact Computing. TPHOLs 2001

G. Melquiond. Floating-point arithmetic in the Coq system. RNC 2008.

P. Miner and S. Boldo. Float PVS library. http://shemesh.larc.nasa.gov/fm/ftp/larc/PVS-library/pvslib.html

P. Miner. Defining the IEEE-854 Floating-Point Standard in PVS. PVS. Technical Memorandum NASA, Langley Research, 1995

J. Harrison. A machine-checked theory of floating-point arithmetic. TPHOLs 1999

Specification of FP properties

A. Ayad and C. Marché. Behavioral properties of floating-point programs. Hisseo publications, 2009.

A. Ayad and C. Marché. Multi-prover verification of floating-point programs. *www.lri.fr/~marche/ayad10ijcar-submission.pdf*, 2010

S. Boldo and J.C. Filliâtre. Formal verification of floating-point programs. ARITH 2007

References

Applications

J. Harrison. Floating point verification in HOL light: The exponential function. FMSD, 16(3), 2000

J. Harrison. Floating-point verification. FM 2005

J. Harrison. Formal verification of square root algorithms. FMSD, 22(2), 2003

B. Akbarpour, A.T. Abdel-Hamid, S. Tahar, and J. Harrison. Verifying a synthesized implementation of IEEE-754 floating-point exponential function using HOL. CJ 53(4), 2010

J. O'Leary, X. Zhao, R. Gerth, C.H. Seger. Formally Verifying IEEE Compliance of Floating-Point Hardware

R, Kaivola and M. D. Aagaard. Divider circuit verification with model checking and theorem proving. TPHOLs 2000

M. Cornea-Hasegan. Proving the IEEE correctness of iterative floating-point square root, divide and remainder algorithms. Intel Technology Journal, Q2, 1998

T. L. J. Strother Moore and M. Kaufmann. A mechanically checked proof of the correctness of the kernel of the AMD5K86 floating-point division algorithm. IEEE Transactions on Computers, 47(9), 1998.

D. Rusinoff. A mechanically checked proof of IEEE compliance of a register-transfer-level specification of the AMD-K7 floating-point multiplication, division, and square root instructions. LMS Journal of Computation and Mathematics, I, 1998.

J. Sawada. Formal verification of divide and square root algorithms using series calculation. ACL2 2002.

S. Boldo, J.-C. Filliâtre and G. Melquiond. Combining Coq and Gappa for Certifying Floating-Point Programs. Calculemus 2009.

References

Floating point abstract domains

A. Chapoutot. Interval slopes as a numerical abstract domain for floating-point variables. SAS 2010

L. Chen, A. Miné and P. Cousot. A sound floating-point polyhedra abstract domain. APLAS 2008

A. Miné. Relational abstract domains for the detection of floating-point run-time errors. ESOP 2004

L. Chen, A. Miné, J. Wang and P. Cousot. An abstract domain to discover interval Linear Equalities. VMCAI 2010

L. Chen, A. Miné, J. Wang and P. Cousot. Interval polyhedra: An Abstract Domain to Infer Interval Linear Relationships. SAS 2009

K. Ghorbal, E. Goubault and S. Putot. The zonotope abstract domain Taylor1. CAV 2009

B. Jeannet, and A. Miné. Apron: A library of numerical abstract domains for static analysis. CAV 2009

D. Monniaux. Compositional analysis of floating-point linear numerical filters. CAV 2005

J. Feret. Static analysis of digital filters. ESOP 2004

F. Alegre, E. Feron and S. Pande. Using ellipsoidal domains to analyze control systems software. CoRR 2009

E. Goubault and S. Putot. Weakly relational domains for floating-point computation analysis. NSAD 2005

E. Goubault. Static analyses of the precision of floating-point operations. SAS 2001

References

Industrial Case Studies

E. Goubault, S. Putot, P. Baufreton, J. Gassino. Static analysis of the accuracy in control systems: principles and experiments. FMICS 2007

D. Delmas, E. Goubault, S. Putot, J. Souyris, K. Tekkal, F. Védrine. Towards an industrial use of FLUCTUAT on safety-critical avionics software. FMICS 2009

J. Souyris and D. Delmas. Experimental assessment of Astrée on safety-critical avionics software. SAFECOMP 2007

J. Souyris. Industrial experience of abstract interpretation-based static analyzers. IFIP 2004

P. Cousot. Proving the absence of run-time errors in safety-critical avionics code. EMSOFT 2007

FP Static Analysers

B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux and X. Rival. A static analyzer for large safety-critical software. SIGPLAN 38(5), 2003

P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux and Xavier Rival. The ASTREÉ analyzer. ESOP 2005

E. Goubault, M. Martel and S. Putot. Asserting the precision of floating-point computations: a simple abstract interpreter. ESOP 2002

Related work

Constraint satisfaction

C. Michel, M. Rueher and Y. Lebbah: Solving constraints over floating-point numbers. CP2001

B. Botella, A. Gotlieb and C. Michel: Symbolic execution of floating-point computations. STVR2006

SMT

P. Ruemmer and T. Wahl. An SMT-LIB theory of binary floating-point arithmetic. SMT 2010

A. Brillout, D. Kroening and T. Wahl. Mixed abstractions for floating point arithmetic. FMCAD 2009

R. Brummayer and A. Biere. Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays. TACAS 2009

Incomplete Solvers

S. Boldo, J.-C. Filliâtre and G. Melquiond. Combining Coq and Gappa for Certifying Floating-Point Programs. Calculemus 2009.