Reasoning about floating-point arithmetic with ACDCL

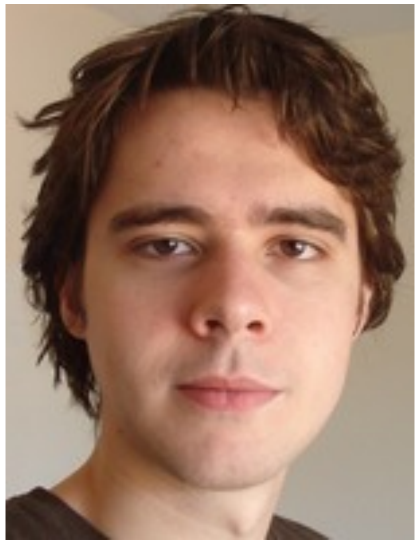
Unifying Abstract Interpretation and Decision Procedures

Daniel Kroening

(joint work with Leopold Haller, Vijay D'Silva, Michael Tautschnig, Martin Brain)

UNIVERSITY OF OXFORD

9 January 2013



Leopold
Haller

Vijay
D'Silva

Michael
Tautschnig

+ Martin Brain
(no photo)

References

- TACAS 2012: paths in floating-point programs with intervals
- POPL 2013: Framework
- VMCAI 2013: DPLL(T)
- FMCAD 2012: Learning for intervals
- SAS 2012: propositional SAT

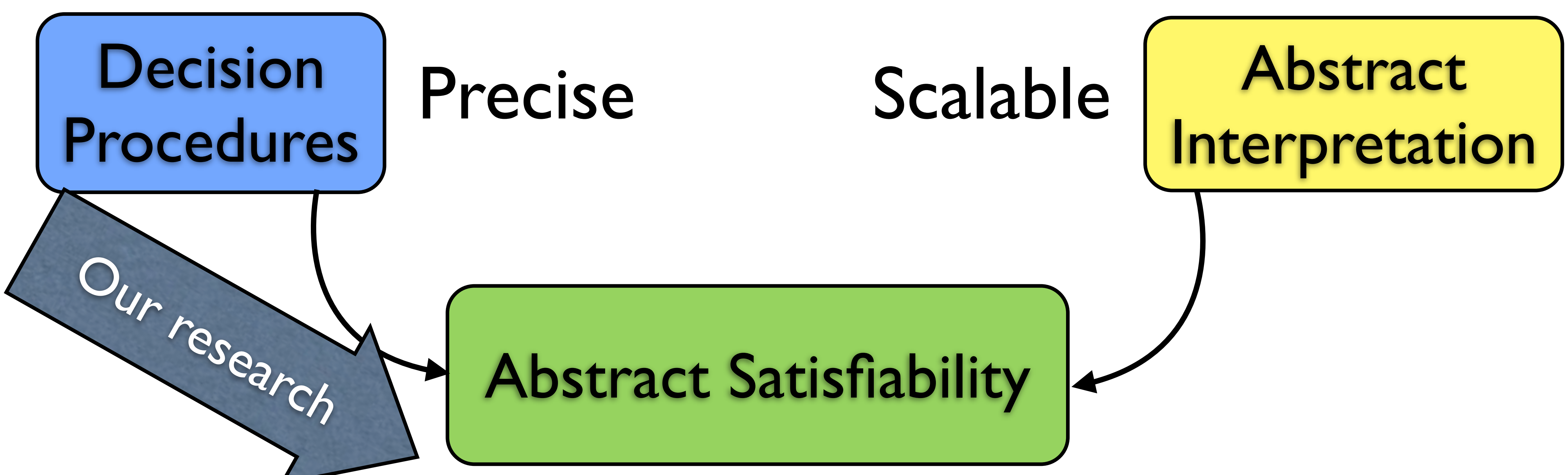
Presentation Outline

Part I

Existing approaches to FP - Verification

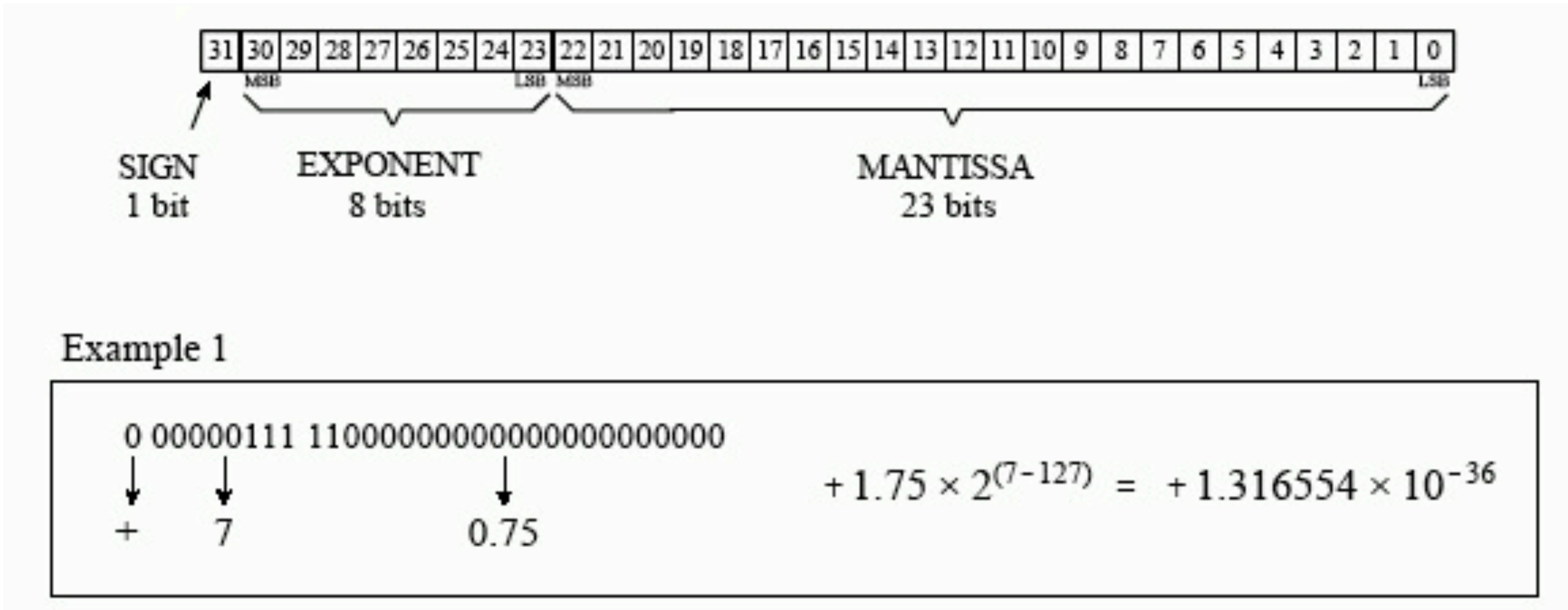
- Manual, Semi-automated
- Decision Procedures
- Abstract Interpretation

Part II

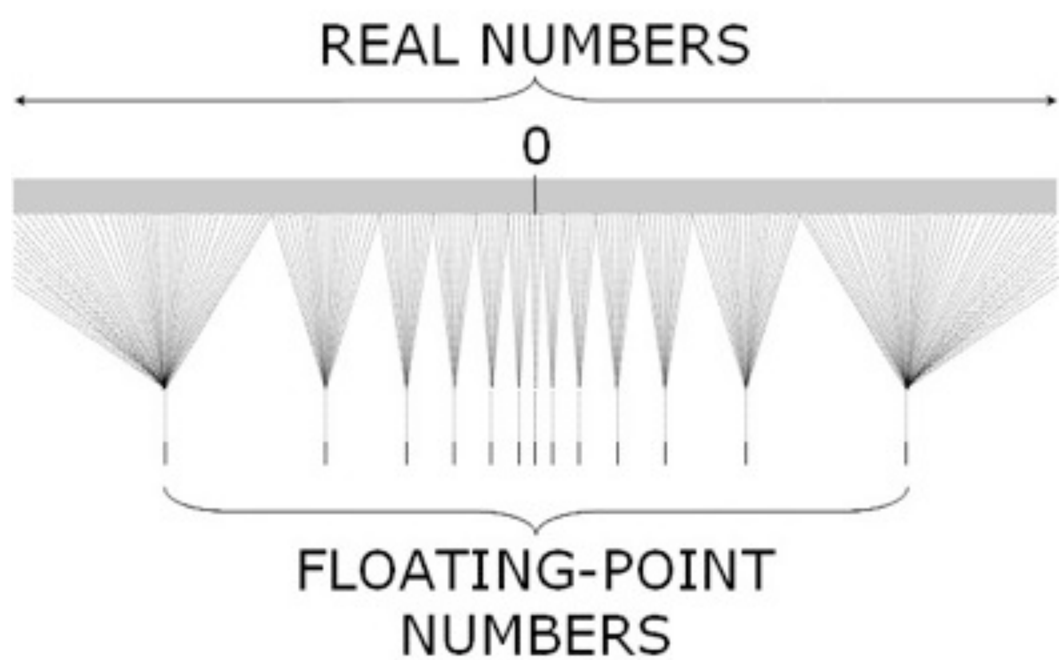


Part I

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?

(We will ignore the case x=NaN)

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

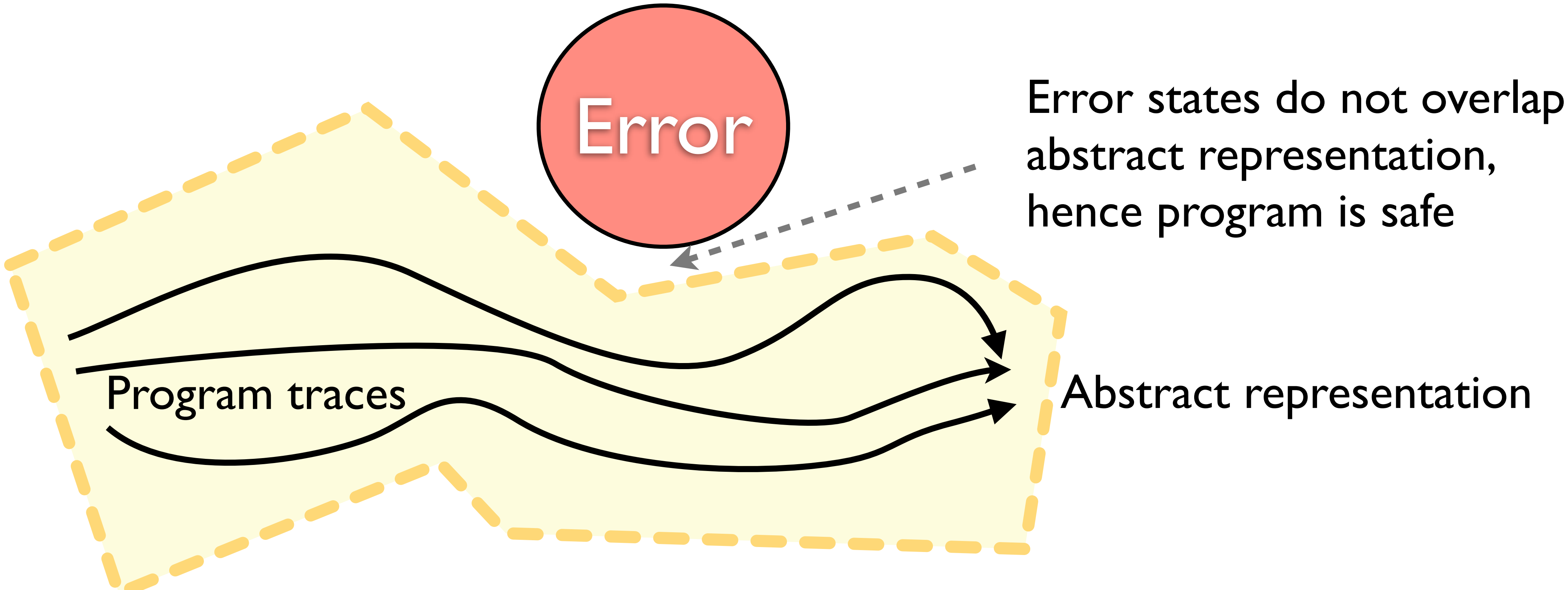
Requires experts,
expensive, powerful

Manual

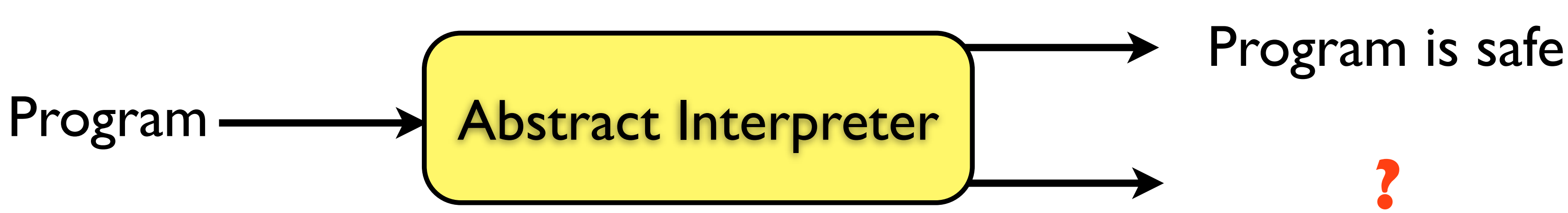
Abstract Interpretation

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.

Interpreter ⟺ 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

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

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]

$x \in [-1.570796, 1.570796]$

$result \in [-2.216760, 2.216760]$

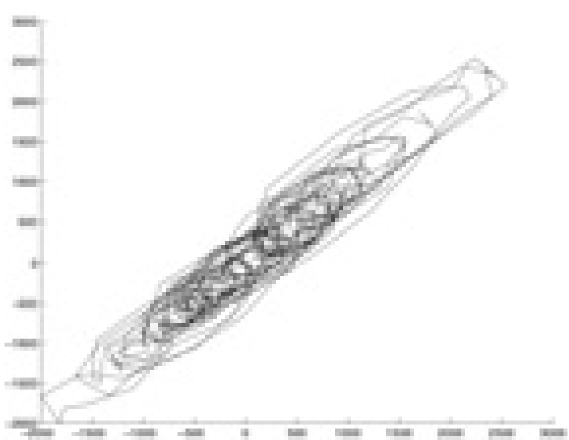
$result \in [-2.296453, 2.296453]$

$result \in [-2.301135, 2.301135]$

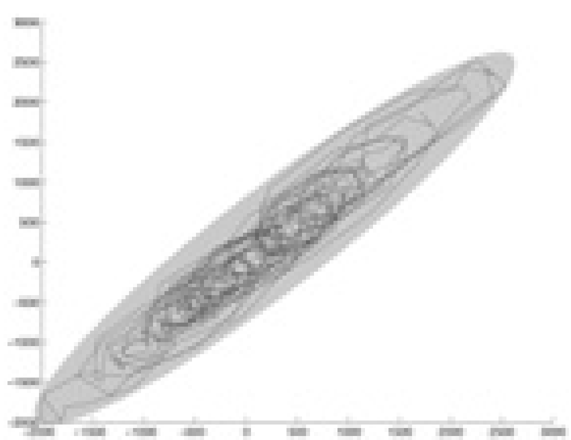
Potentially unsafe

Astrée Abstract Interpreter

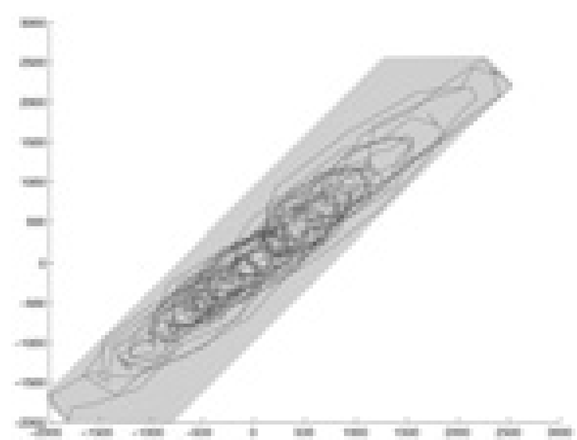
- Mature abstract interpreter by Cousot et. al
- Large number of domains
- Sold and supported by Absint GmbH
- Successful in proving correct large avionics control software: 100k lines of code in 1h -> <u>highly scalable</u>
- Various domains for floating point analysis:



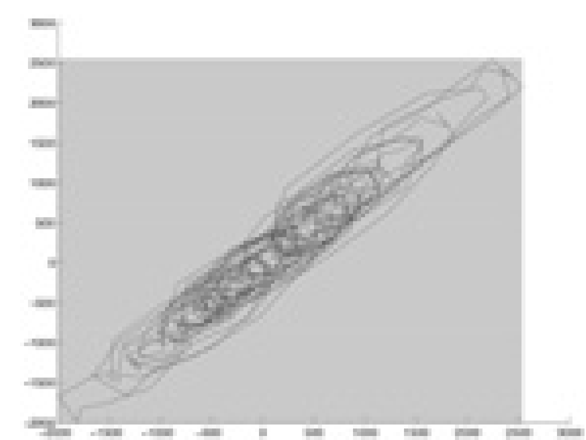
Original traces



Ellipses



Octagons



Intervals

Abstract Domains for Floating Point

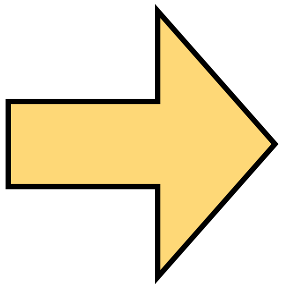
- Abstract domains are typically formulated over the real or rational numbers

- Numeric domains rely on mathematical properties such as associativity which do not hold over floating point numbers

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

- Solution (Mine 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);
```

Imprecision in Abstract Interpretation

- The efficiency of abstract interpreters comes at the cost of precision. 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

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

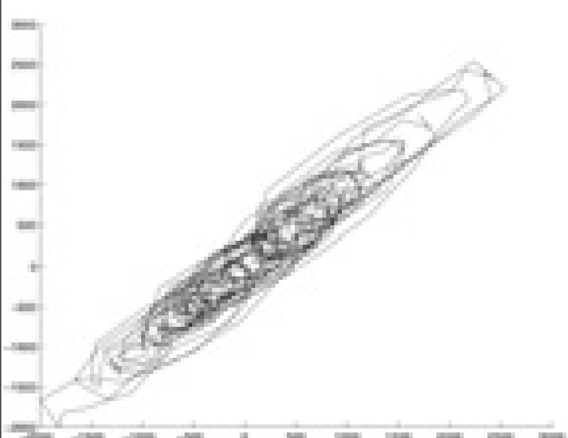
```
while(x < 100000)
{  x++; y++; }
```

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

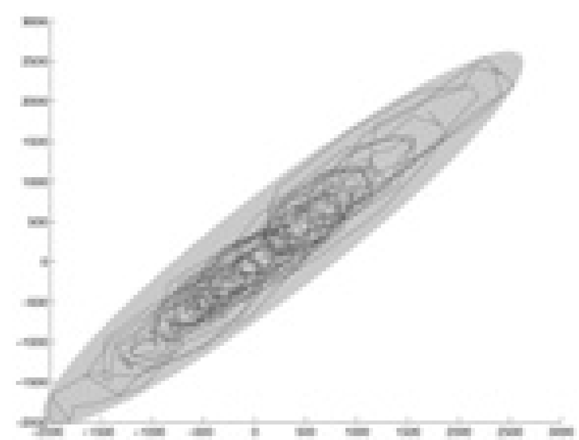
$y \in [\min(Int), \max(Int)]$

Imprecision in Abstract Interpretation

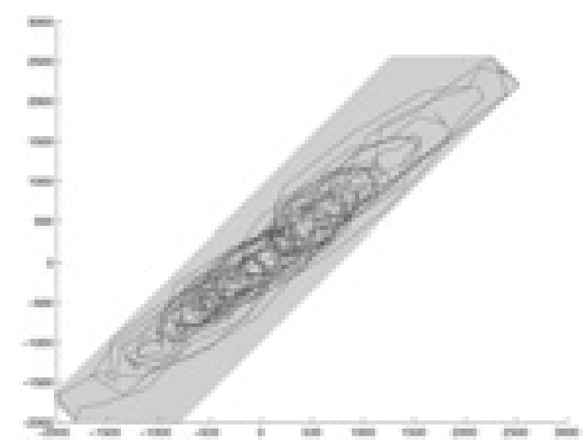
- For efficiency reasons, most numeric abstract domains are convex



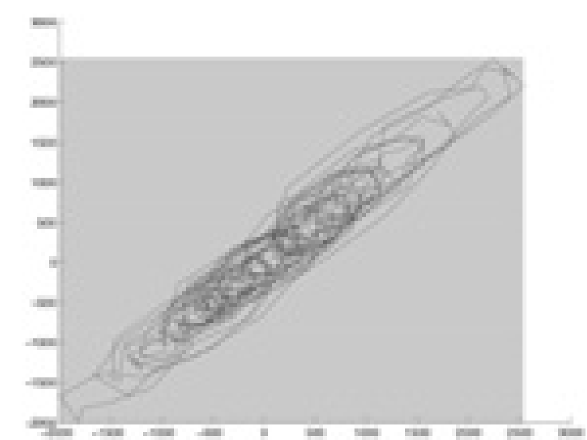
Original traces



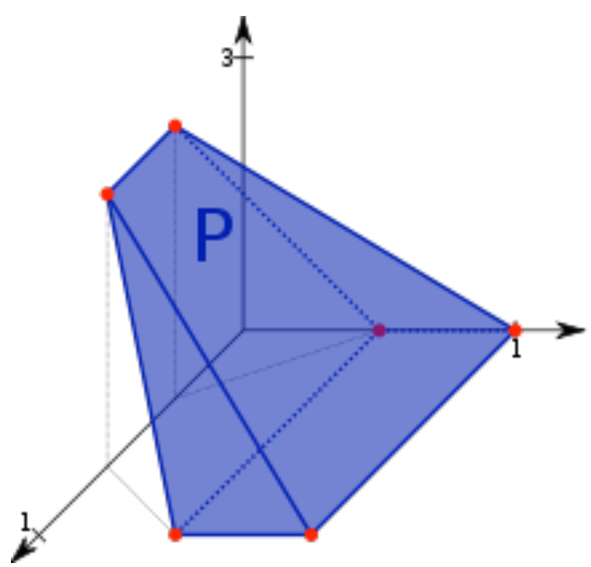
Ellipses



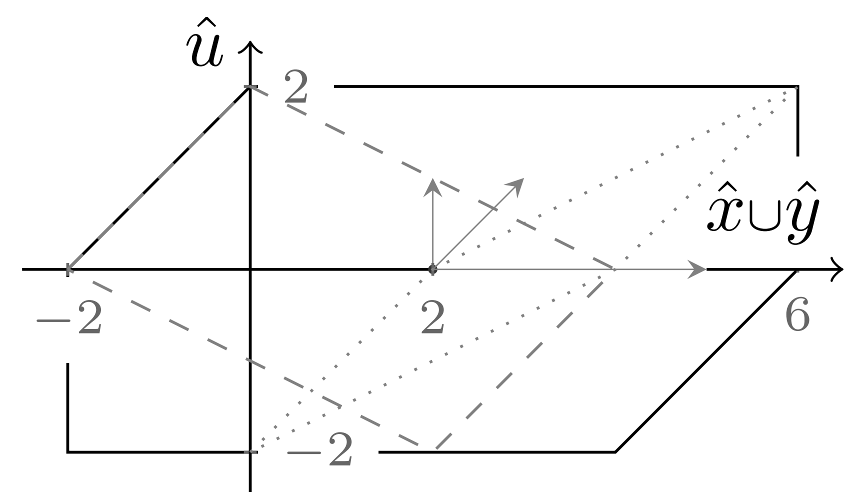
Octagons



Intervals



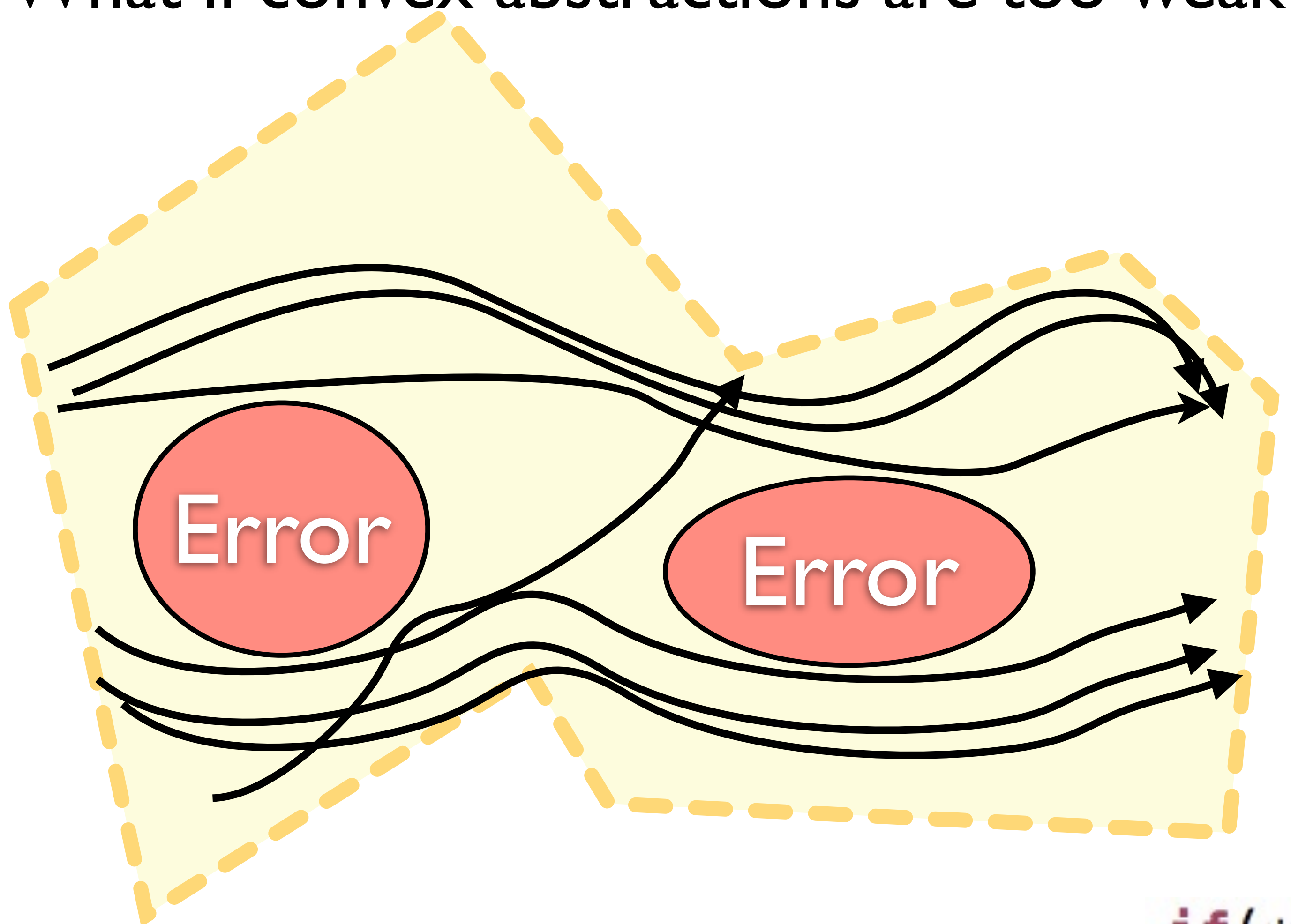
Convex polyhedra



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);
```

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

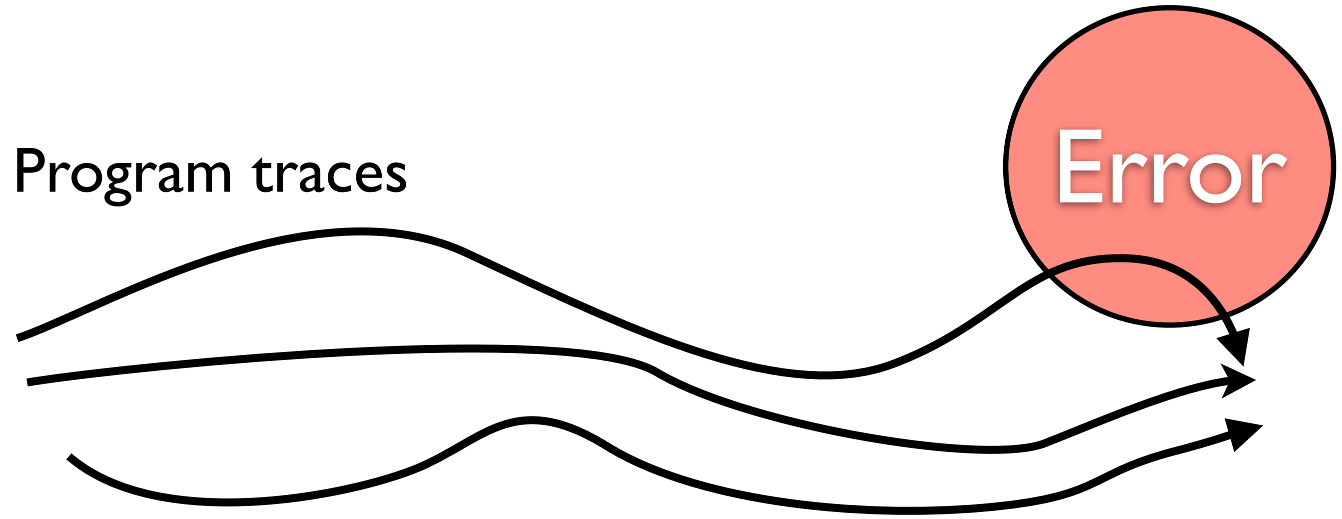
Abstract Interpretation

Scalable and efficient.
Precise analysis requires experts

Decision Procedures

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

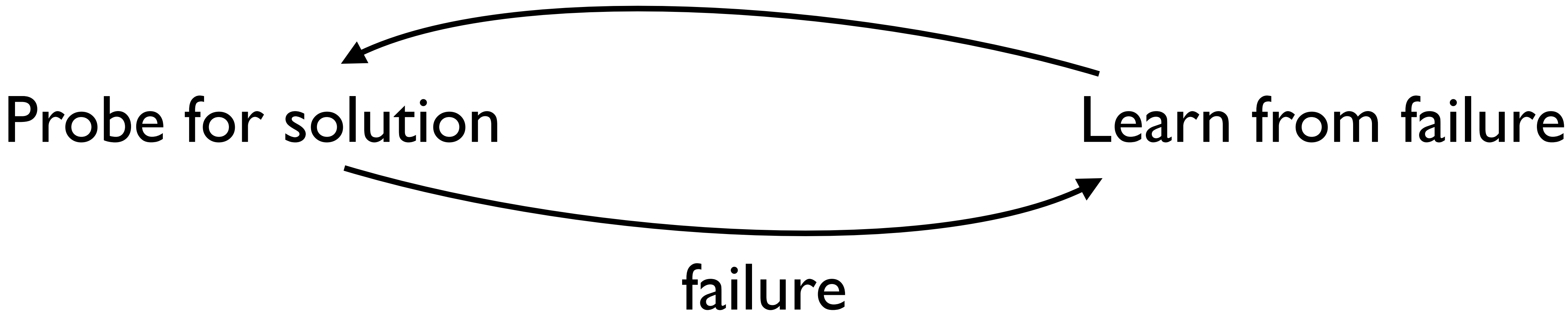
Propositional SAT

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



- 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);

}
```

$$c \to (r = a /_{32} b)$$
$$\wedge \quad \neg c \to (r = a *_{32} b)$$
$$\wedge \quad a > 0 \wedge b > 0 \wedge r < 0$$

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

The formula evaluates to true under the following assignment:

$$a, b \mapsto 123456789$$
$$r \mapsto -1757895751$$
$$c \mapsto \text{false}$$

Counterexample!

Bounded Model Checking

Loops require unrolling before translation

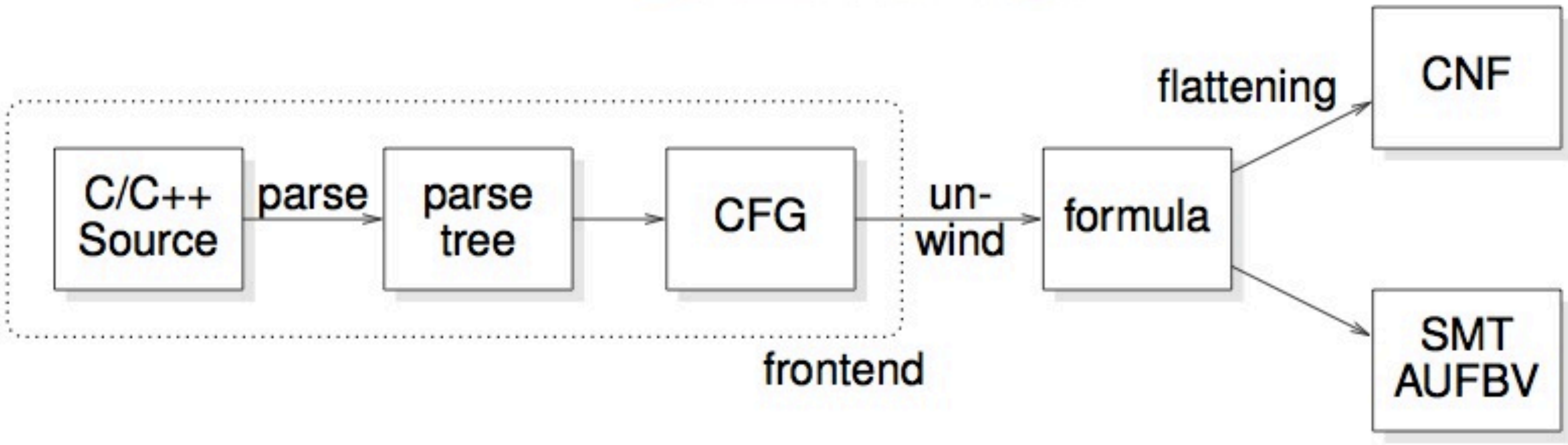
```
int foo(int *a)
{
  int sum;

  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 bit-precise reasoning over floating-point numbers using a propositional encoding
- Uses IEEE-754 semantics with support various rounding-modes
- Allows proofs of complex, bit-level 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 large 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

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

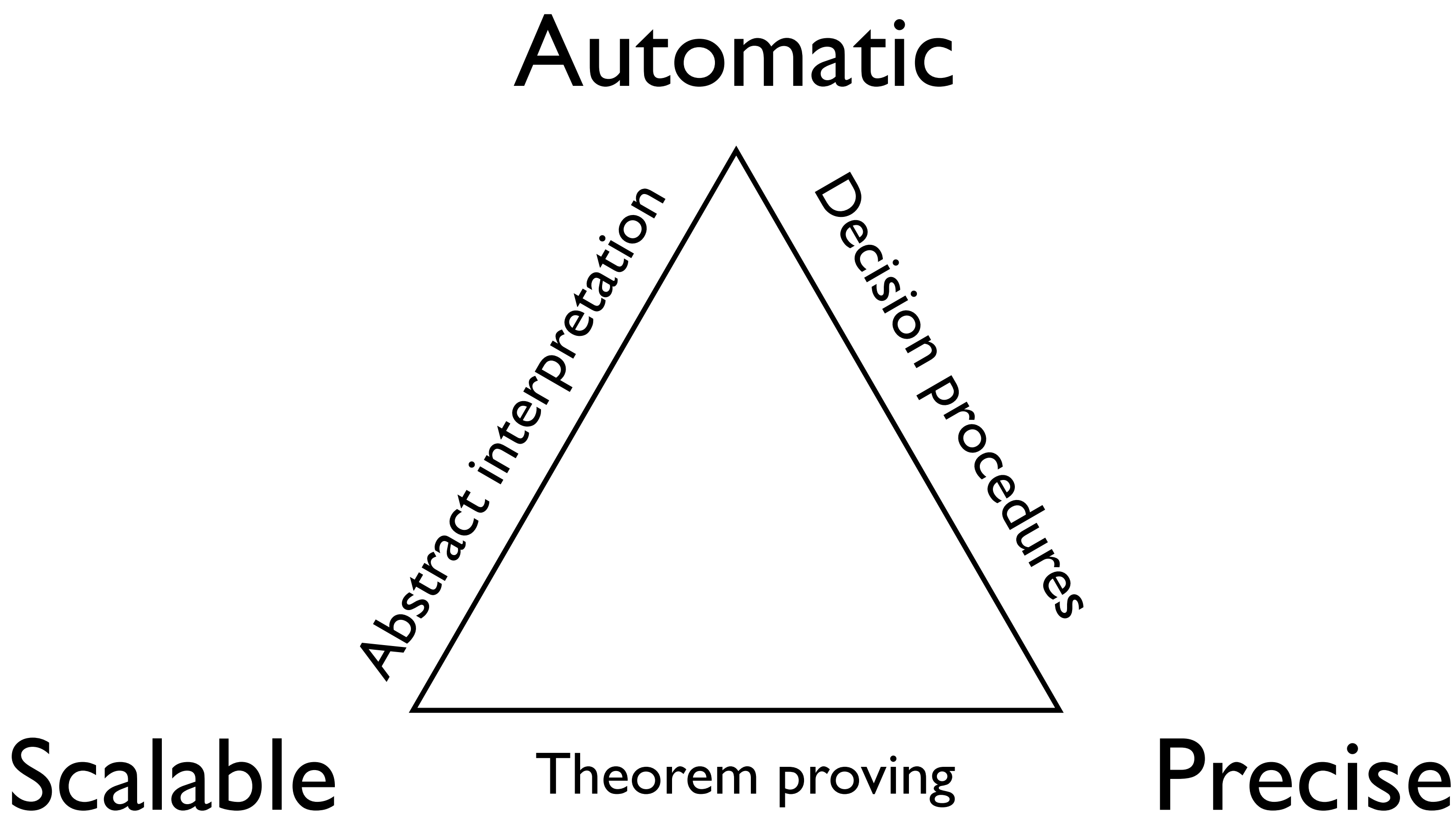
Abstract Interpretation

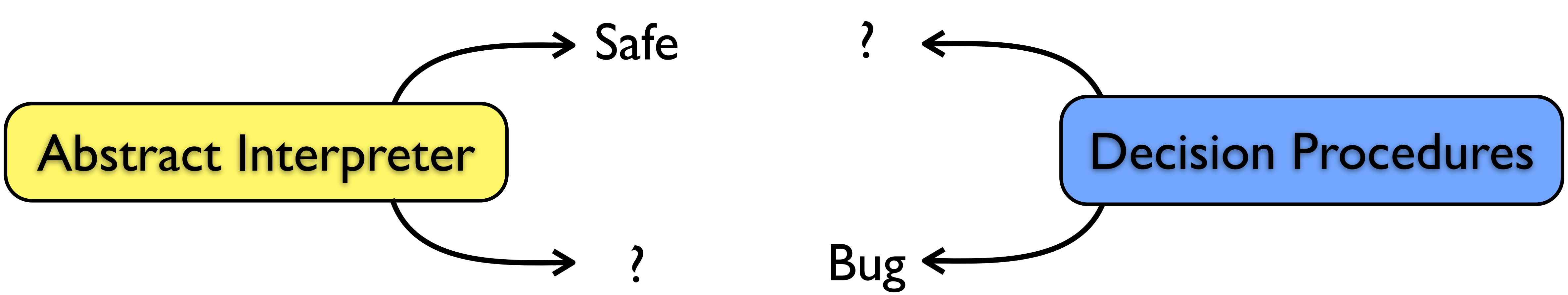
Scalable.
Precision requires experts

Decision Procedures

Precise.
Scalability requires experts

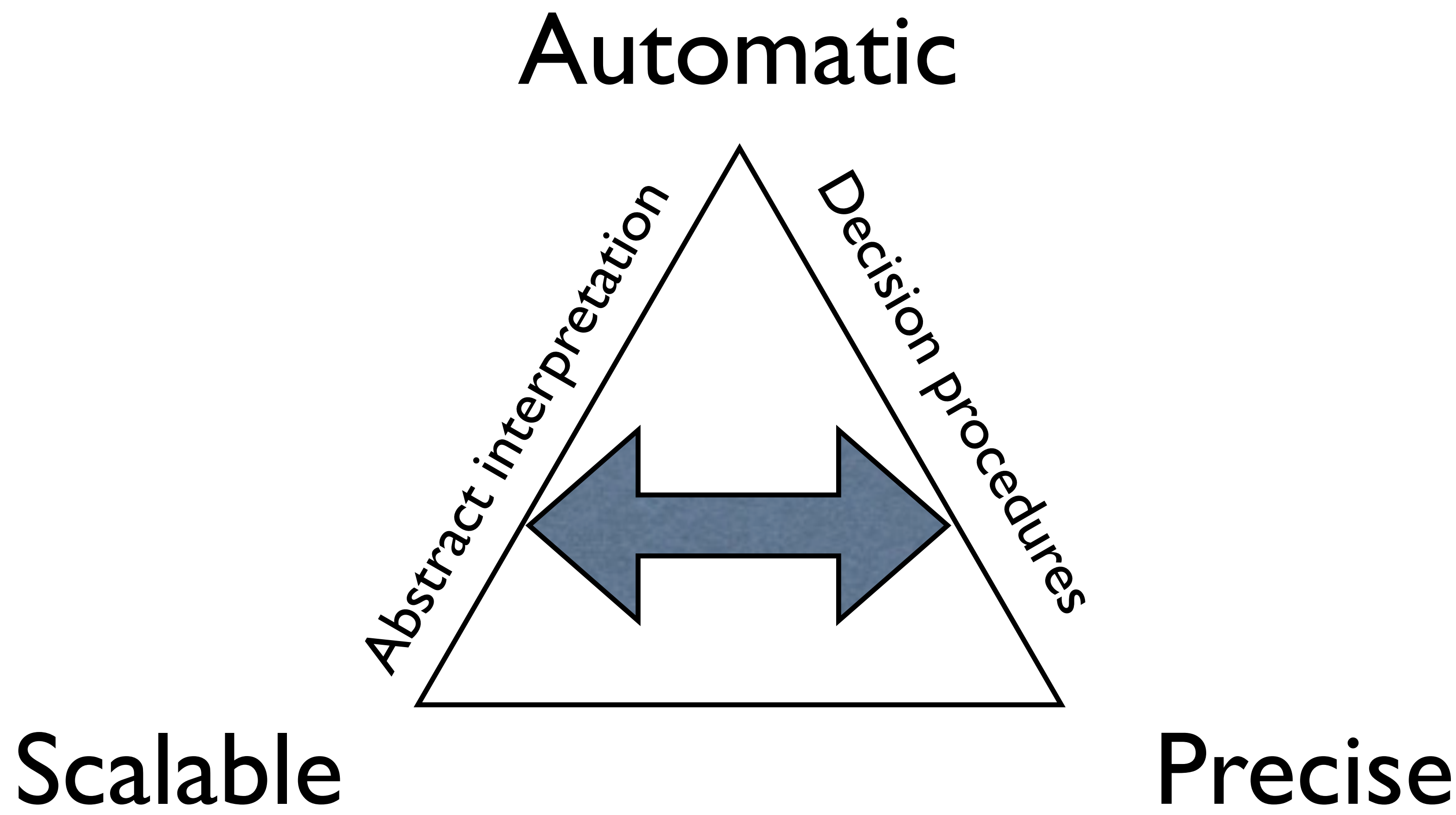
Conclusion Part I





Questions so far?

Part II



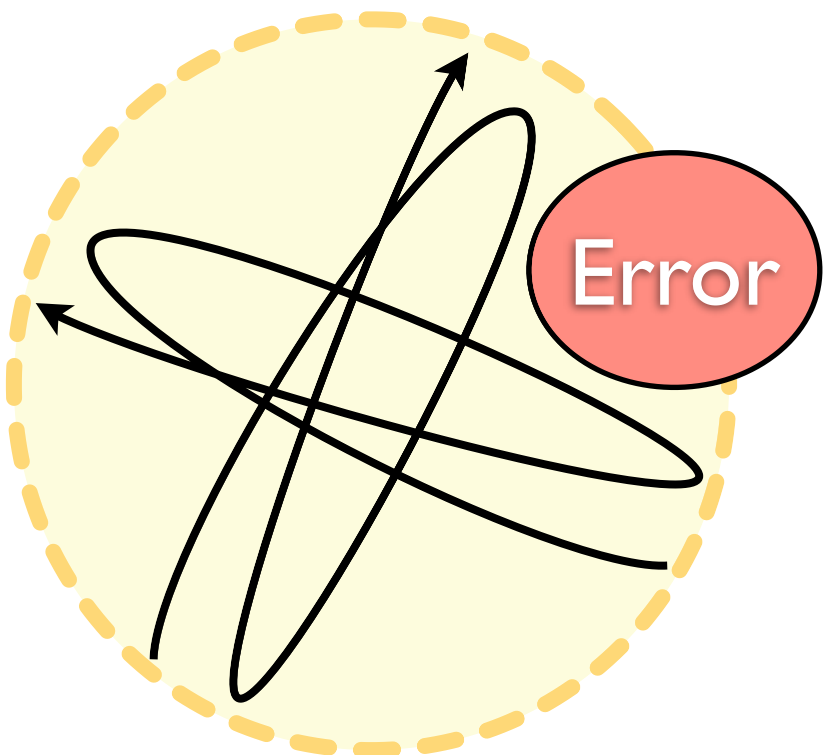
We are interested in techniques that are

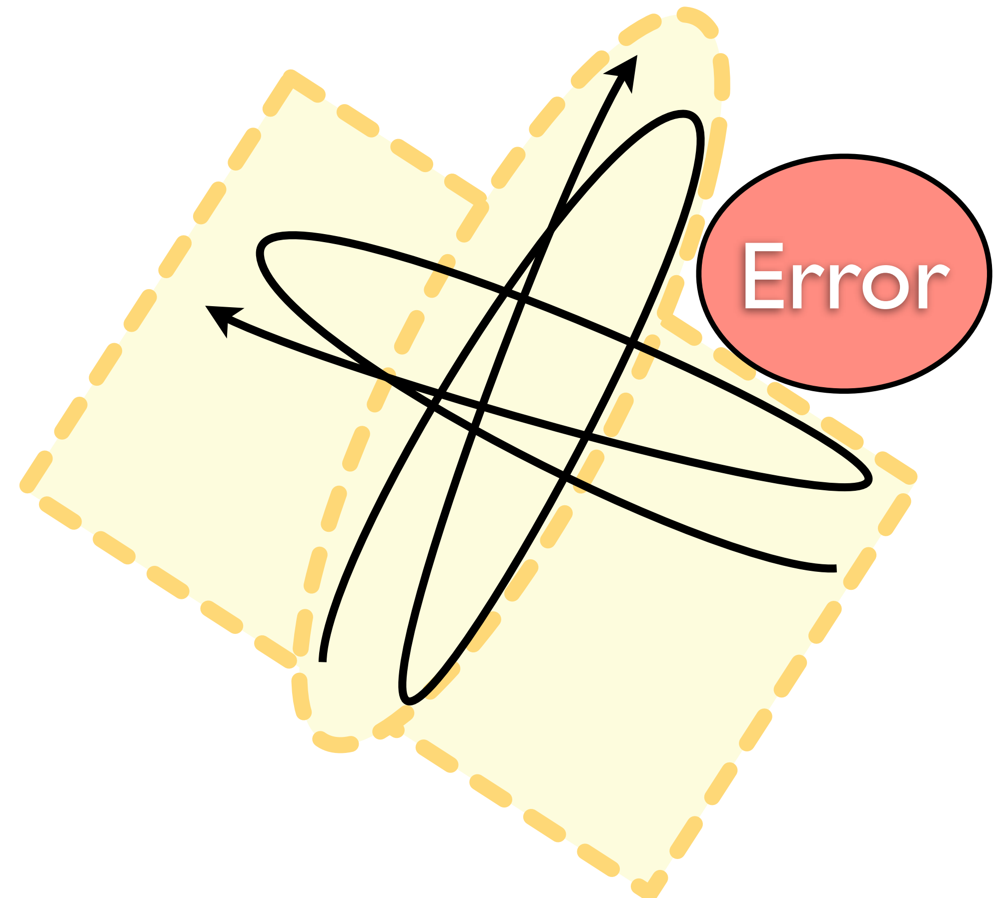
- scalable
- sufficiently precise to prove safety
- fully automatic

<u>Central insight:</u>

Modern decision procedures *are* abstract interpreters!

Manually adjusting analysis precision by <u>abstract partitioning</u>






```
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

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

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

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

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

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

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

$p \in [1, 1]$ (after `if(p)`)

$q \in [0, 0]$ (after `if(!p || !q)`)

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

$p \mapsto$ t

$q \mapsto$ f

$\Longrightarrow$

$p \mapsto$ t

$q \mapsto$ f

$r \mapsto$ 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

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

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

⇒

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

⇒ 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

φ $\longrightarrow$ $\varphi \wedge r$

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

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

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

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

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

SAT is Abstract Analysis

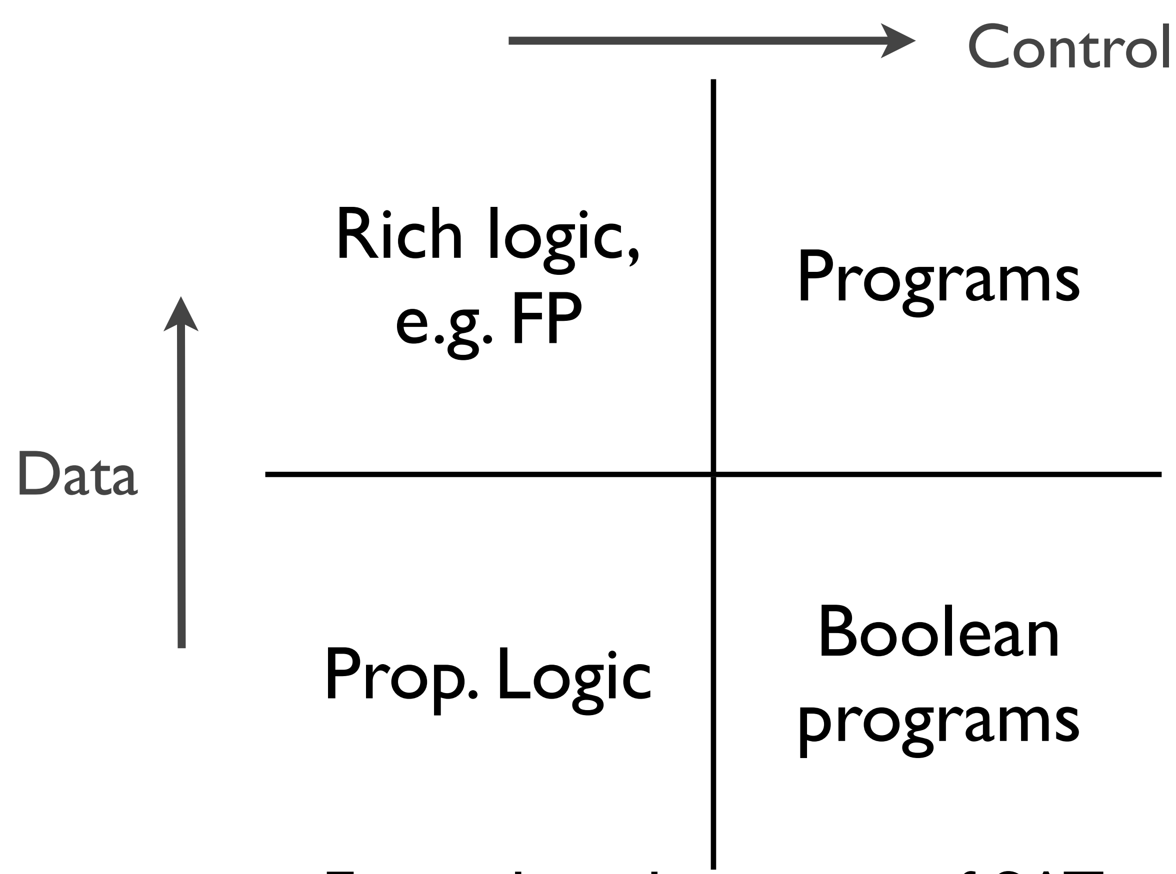
- Deduction in SAT is abstract interpretation
- Decisions and learning are abstract partitioning
- The SAT algorithm is really an automatic partition refinement algorithm.

SAT(*A*)

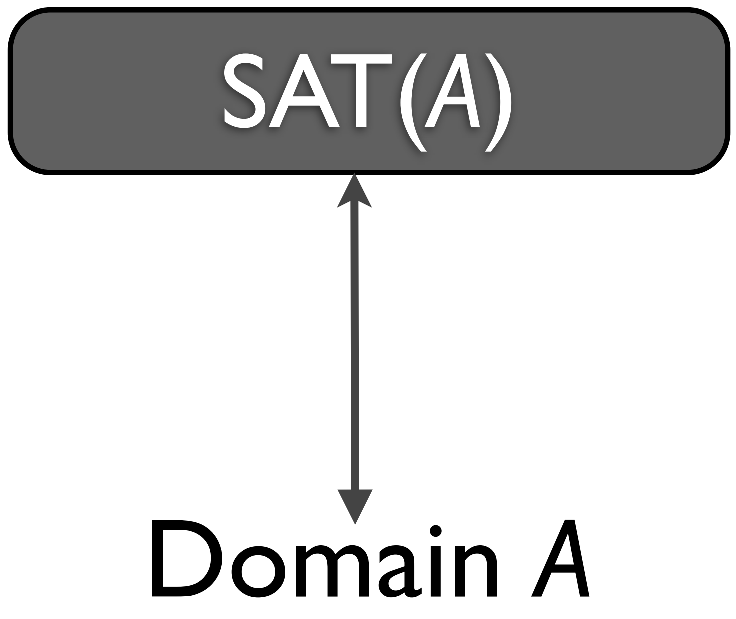
Domain *A*

SAT is Abstract Analysis

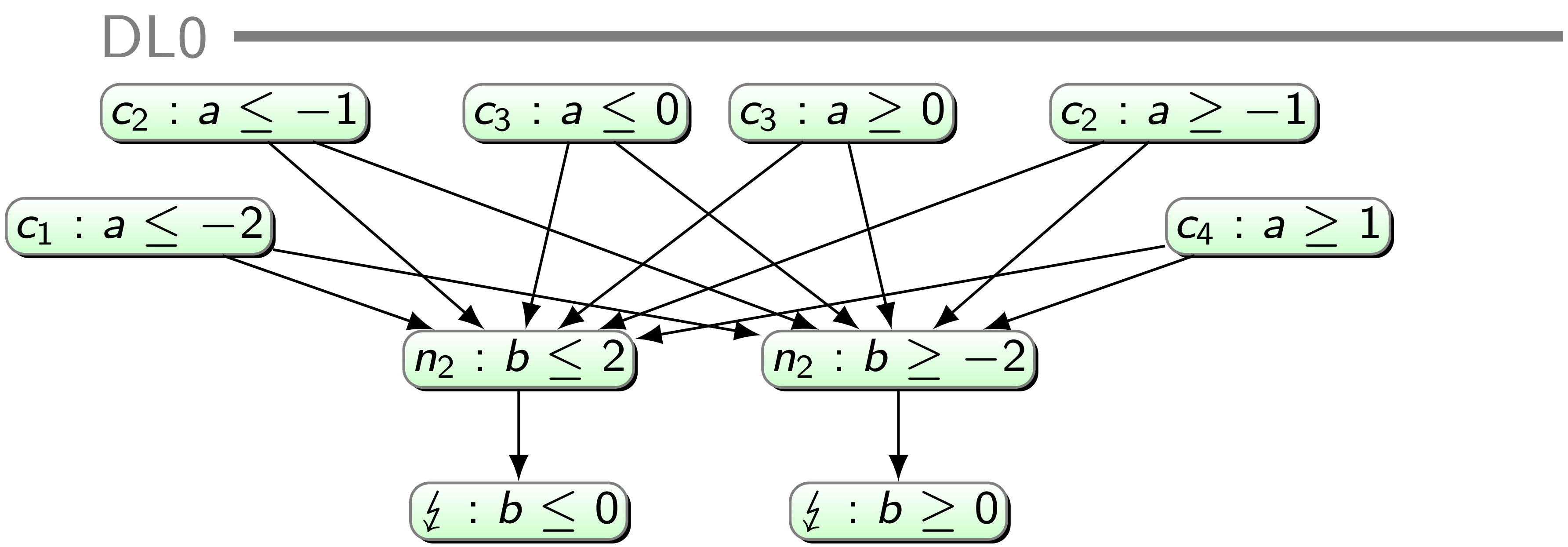
- Deduction in SAT is abstract interpretation
- Decisions and learning are abstract partitioning
- The SAT algorithm is really an automatic partition refinement algorithm.

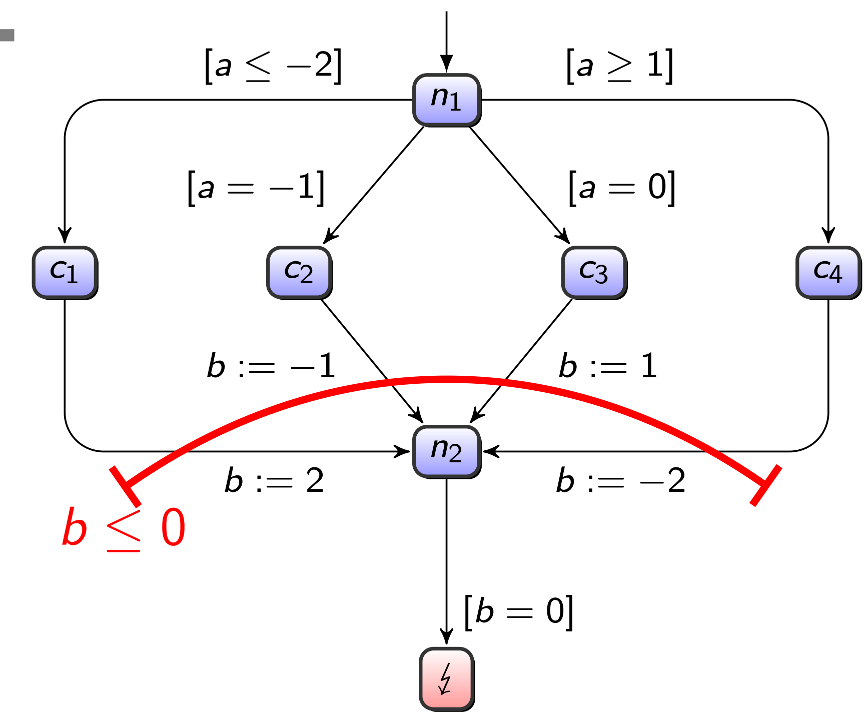


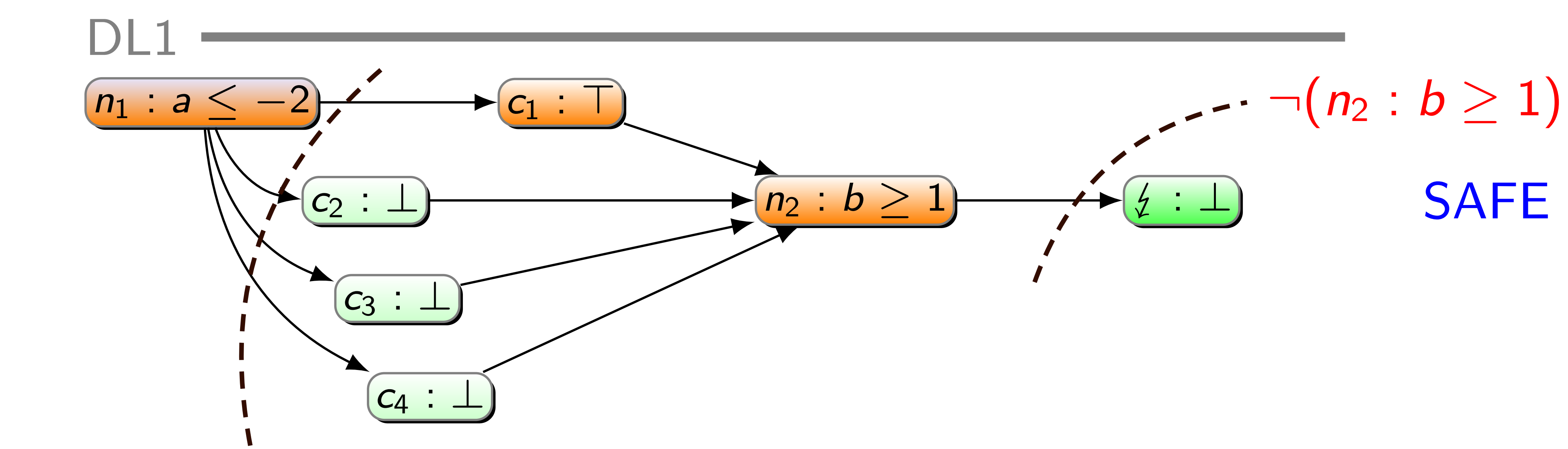
Expanding the scope of SAT



SAT for programs





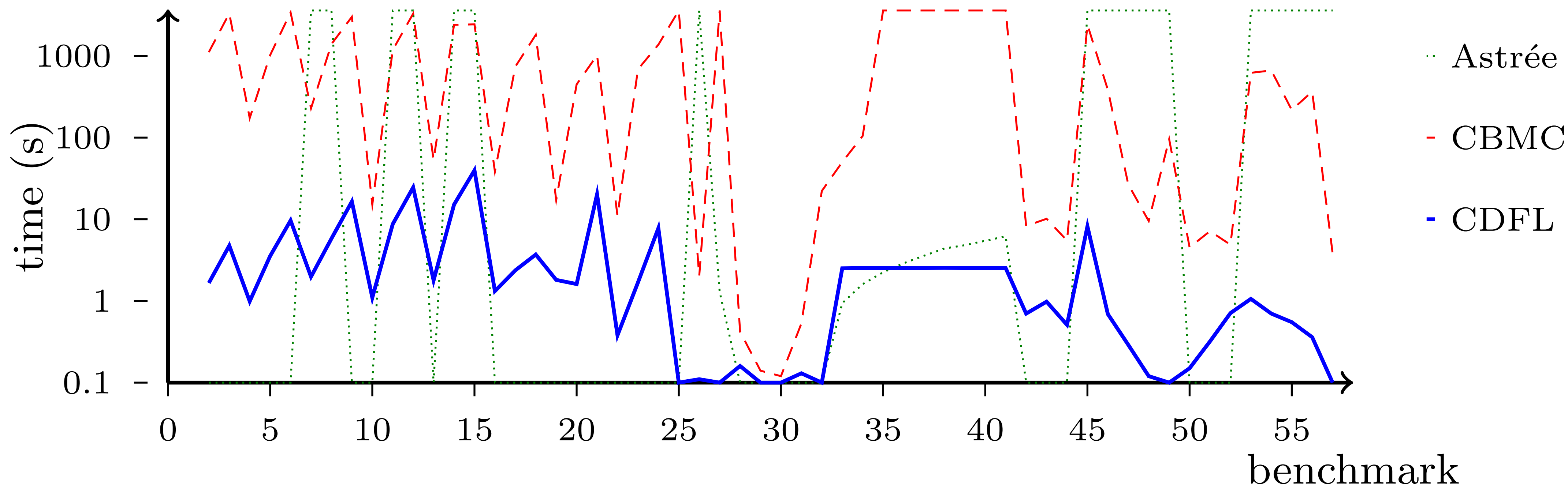


SAFE $\rightarrow$ find cut

Prototype: Abstract Conflict Driven Learning (ACDL)

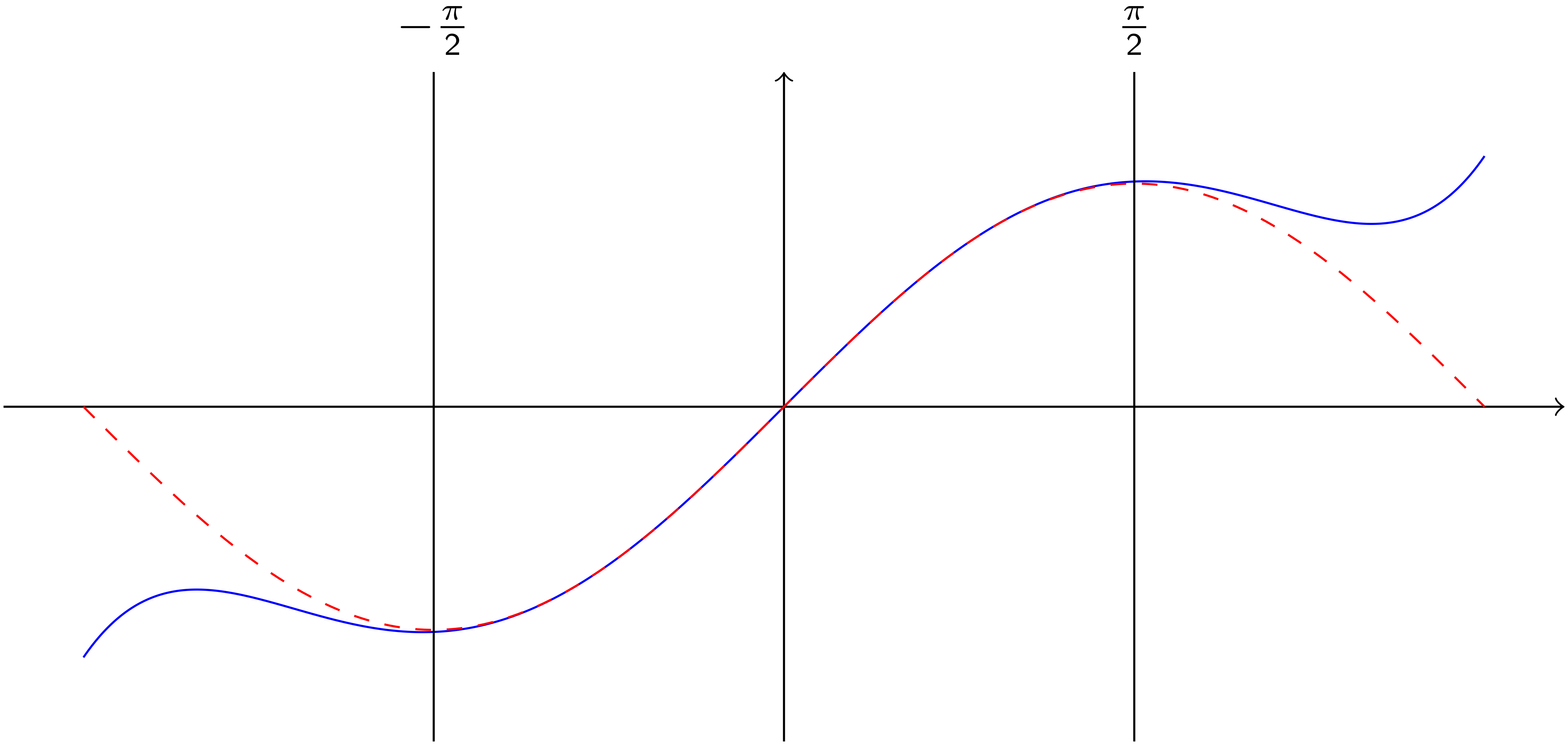
- Implementation over floating-point intervals
- Automatically refines an analysis in a way that is
  - Property dependent
  - Program dependent
- Uses learning to intelligently explore partitions
- Significantly more precise than mature abstract interpreters
- Significantly more efficient than floating-point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

More results



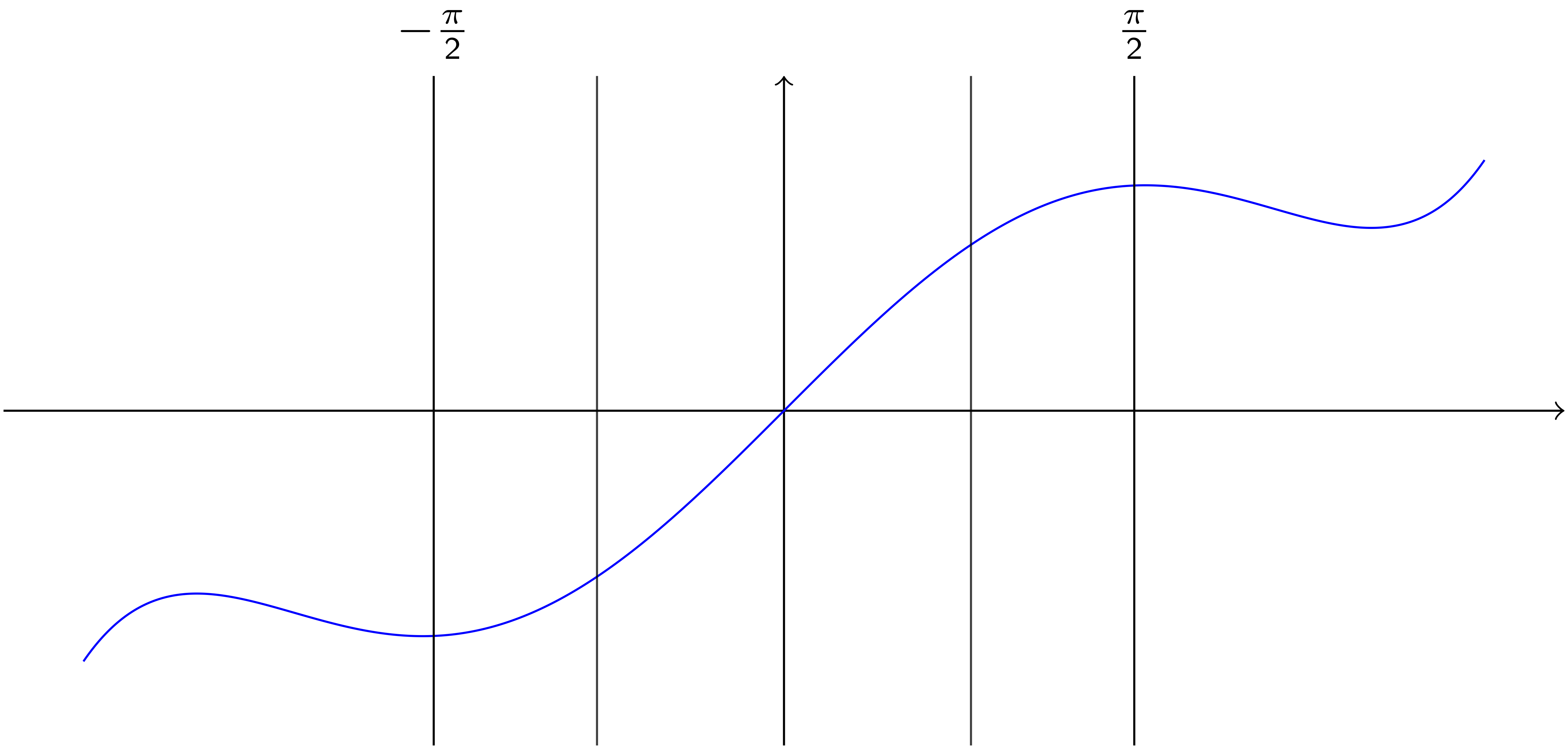
Average speedup over CBMC ~270x

Implementation



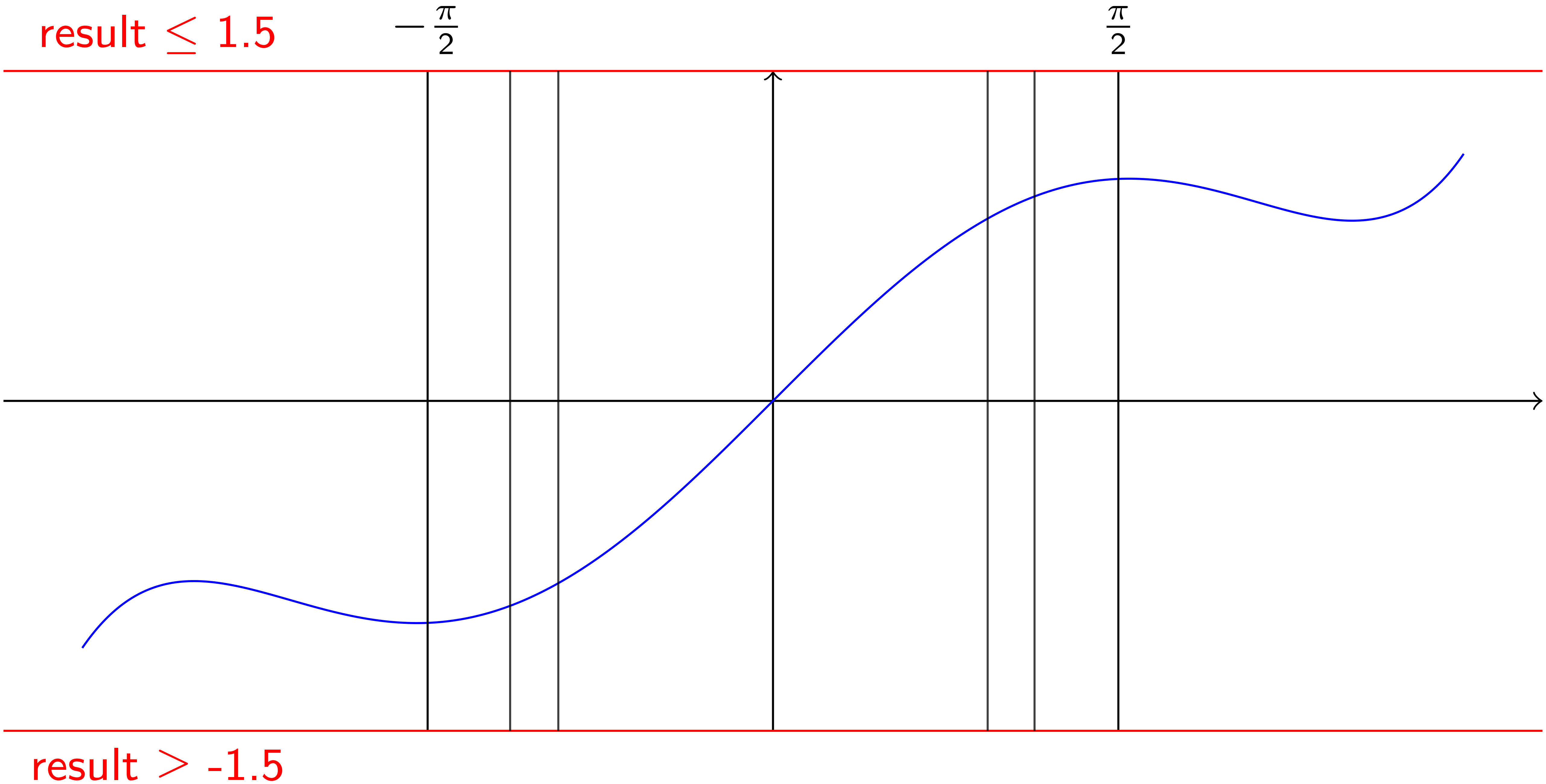
Number of partitions vs. tightness of bound

result $\leq$ 2.0

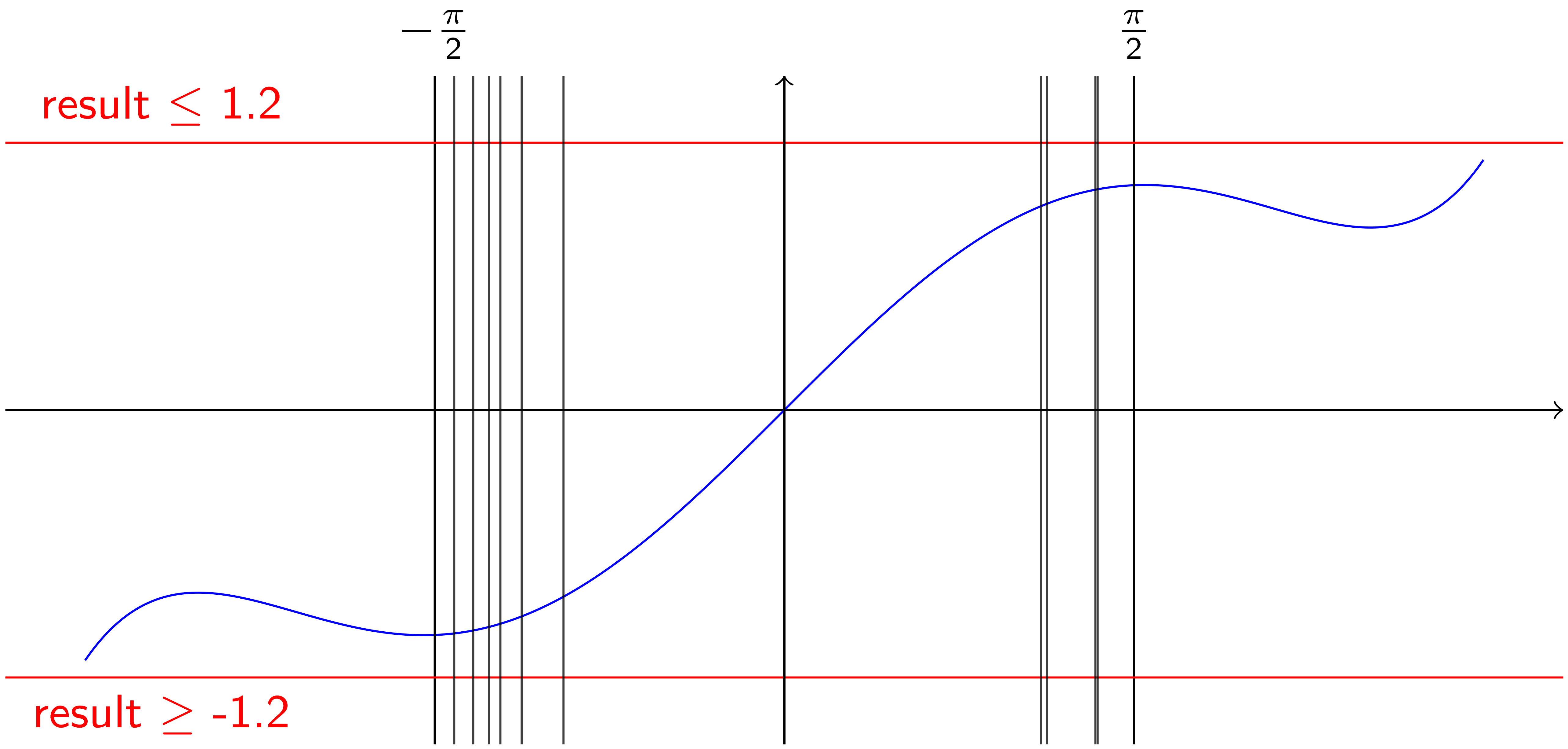


result $\geq$ -2.0

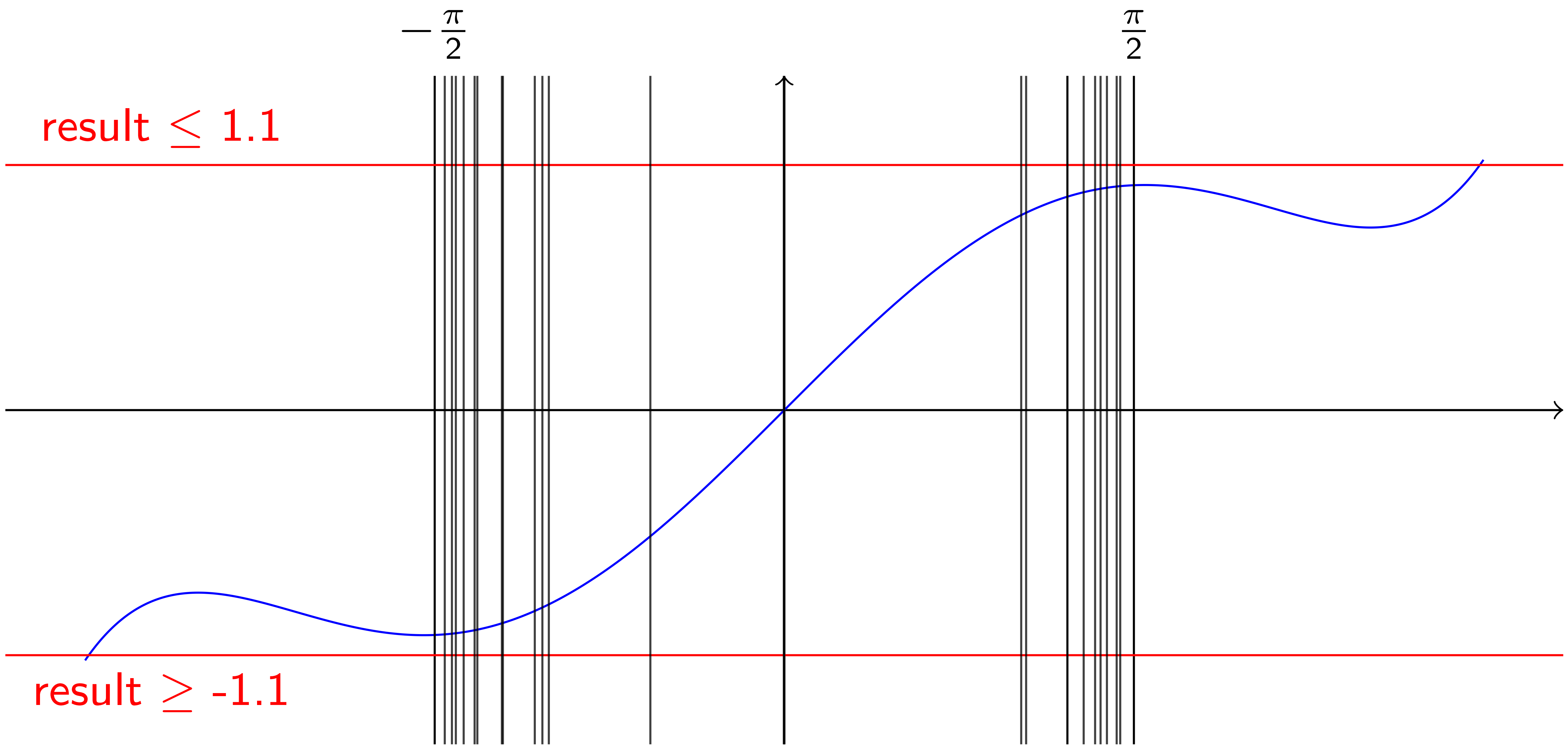
Number of partitions vs. tightness of bound



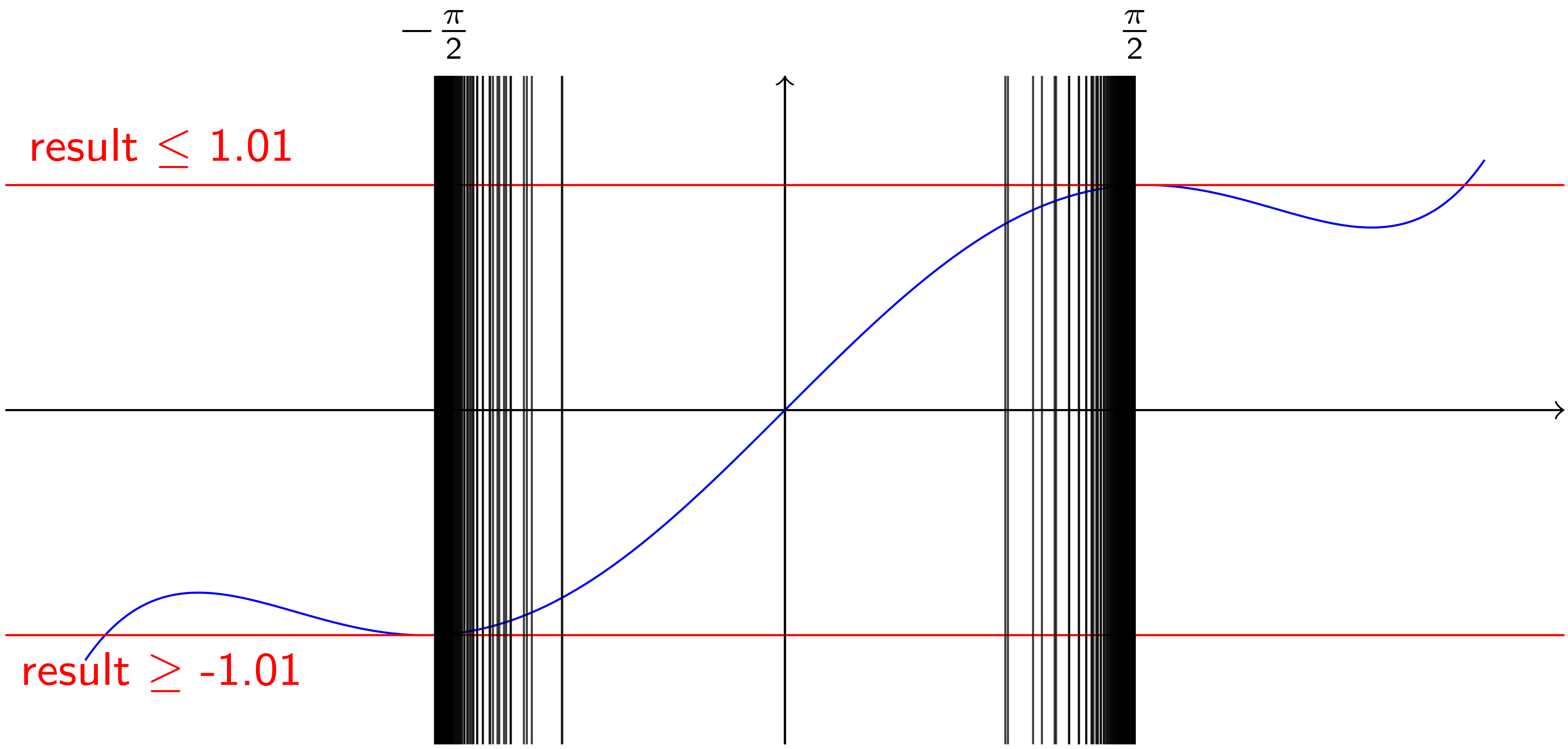
Number of partitions vs. tightness of bound



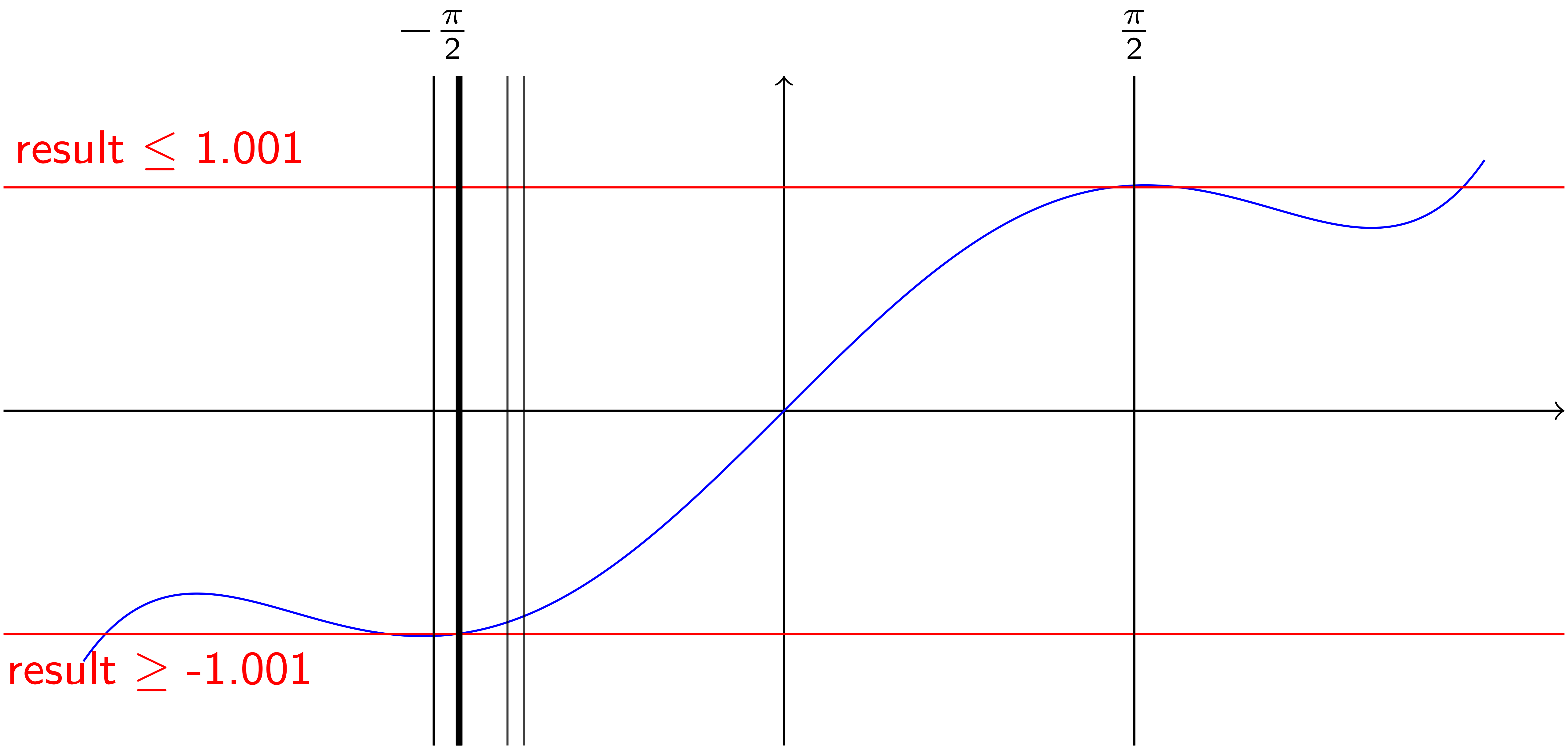
Number of partitions vs. tightness of bound



Number of partitions vs. tightness of bound



Number of partitions vs. tightness of bound



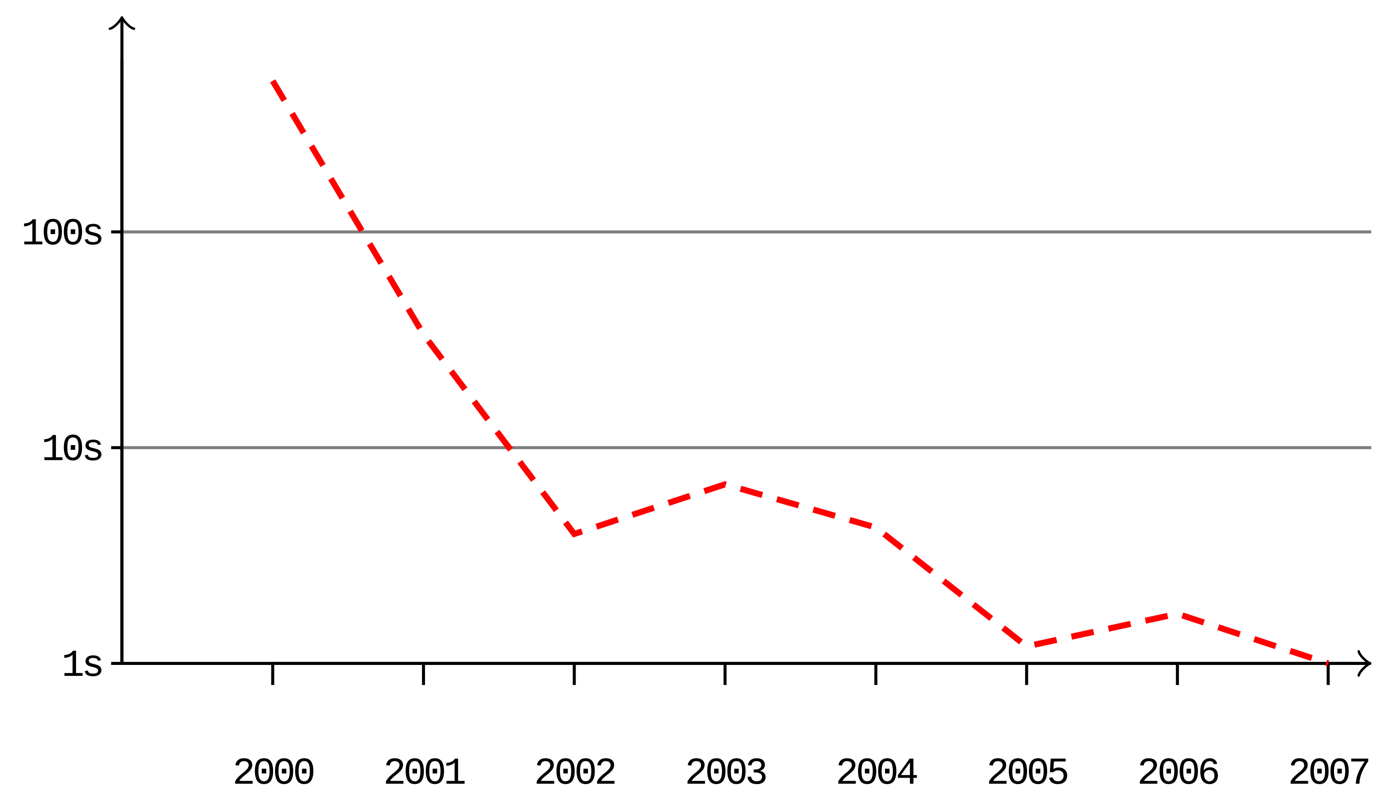
Current and Future Work

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

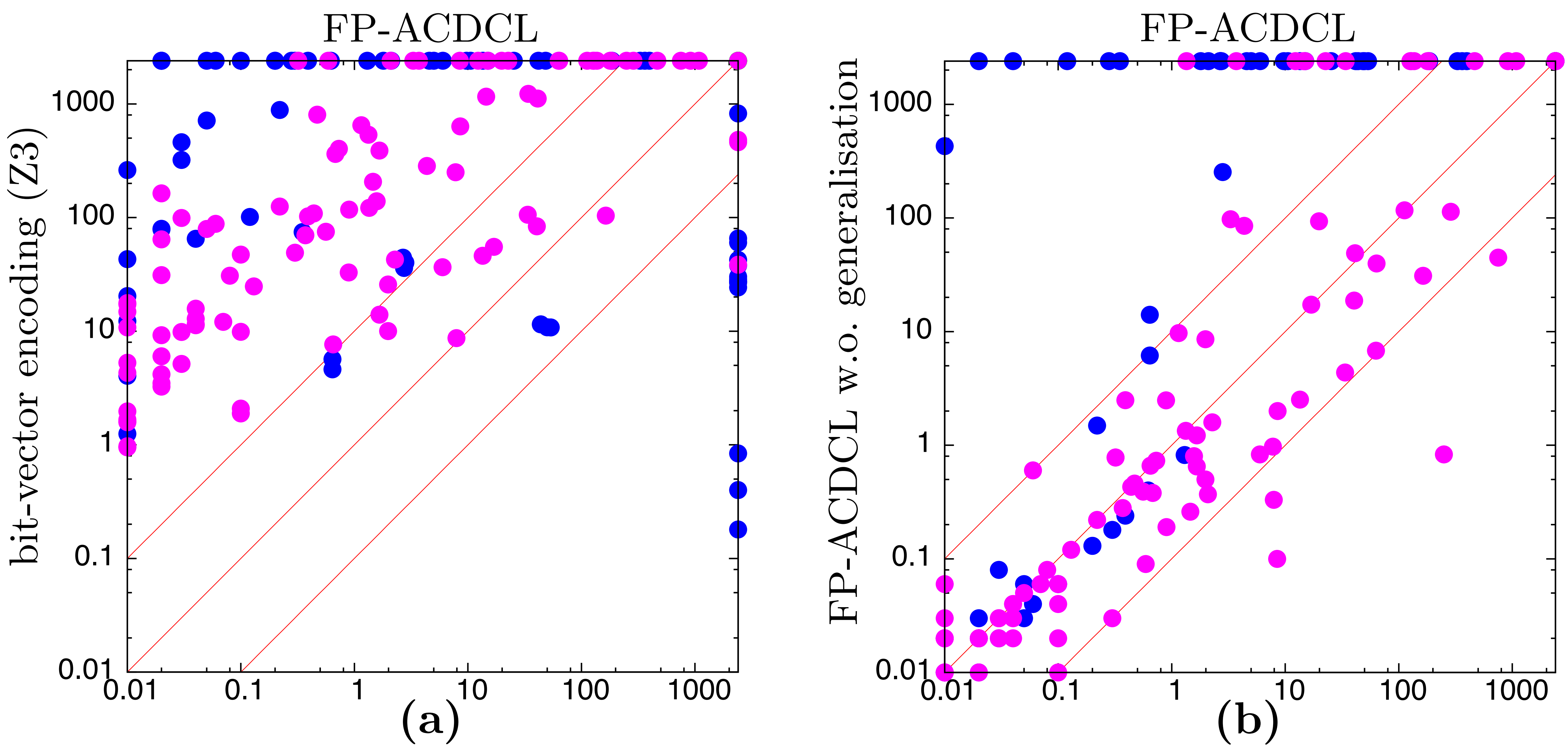
Current and Future Work

- 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



MathSAT + ACDCL

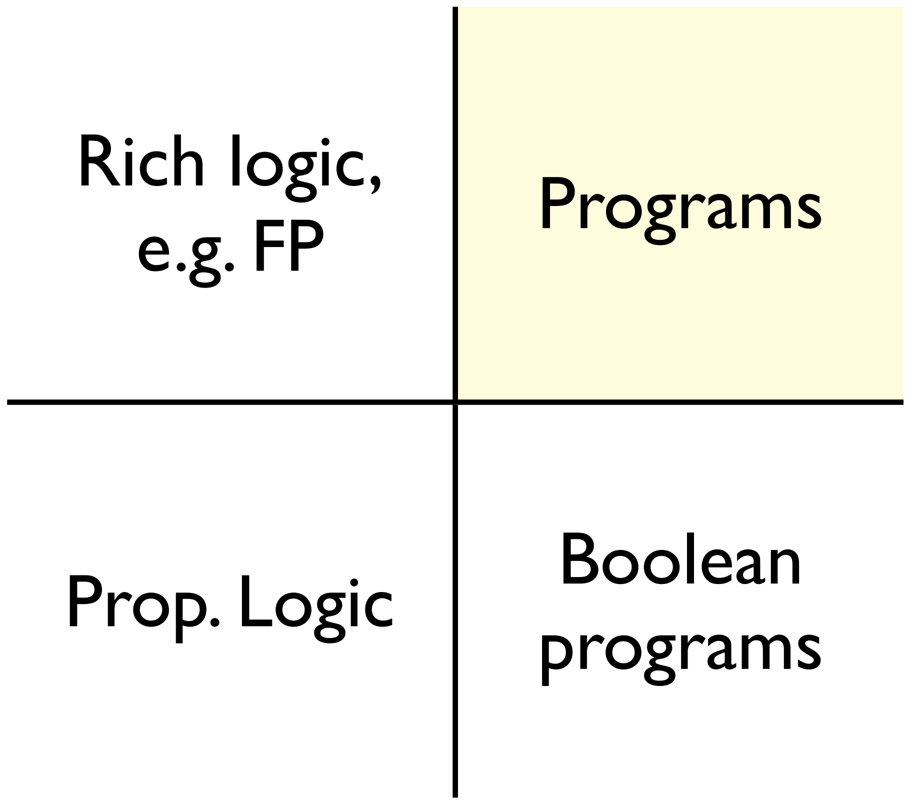


Current and Future Work

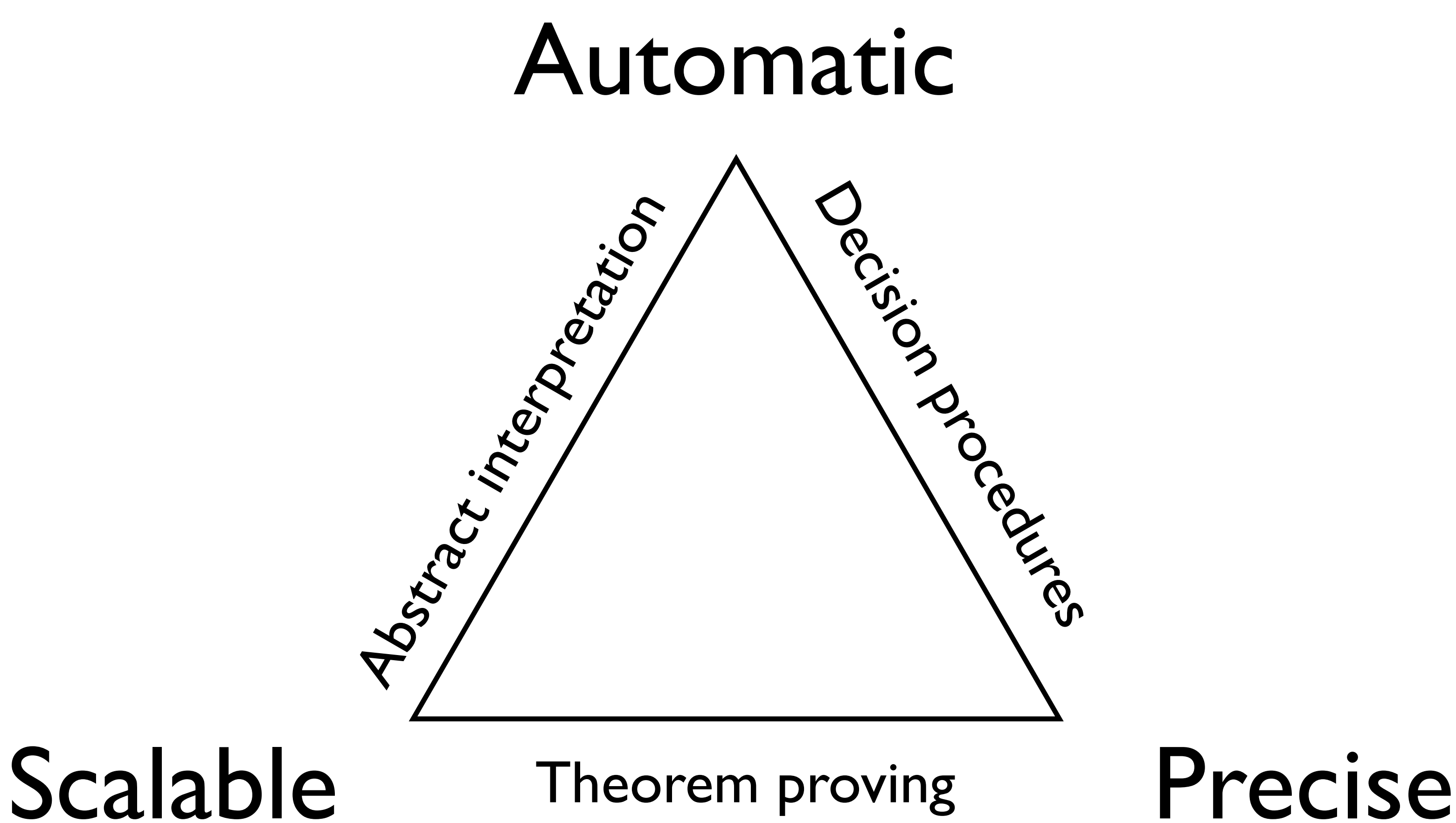
- Reengineer prototype into a tool for floating point verification
  - Significantly improved efficiency
  - Generic interface for integrating abstract domains
  - Development and generalisation of heuristics and learning strategies

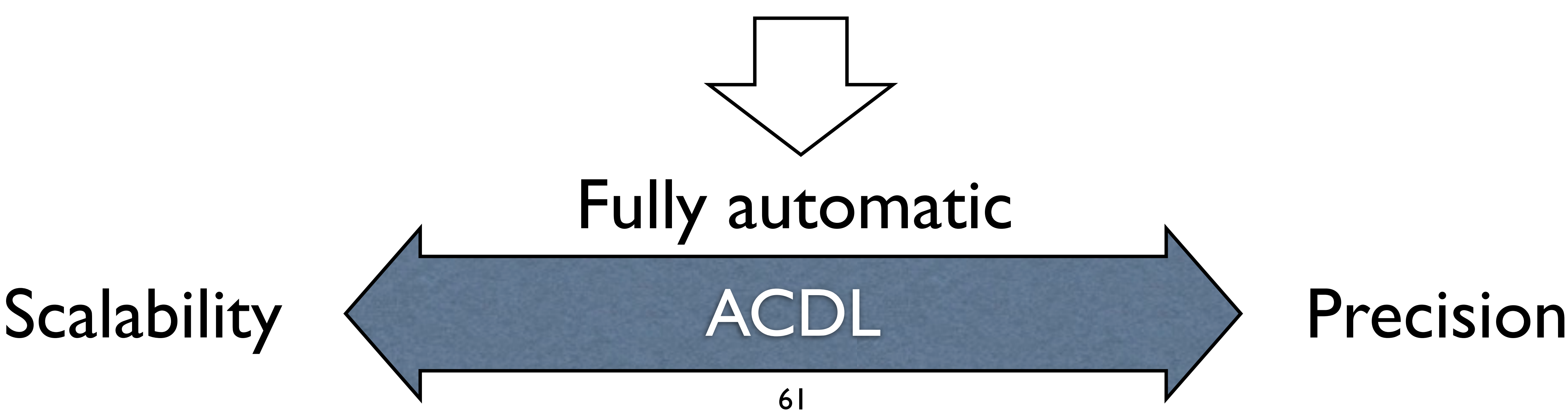
Current and Future Work

- 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



Conclusion - Part II





Thank you for your attention