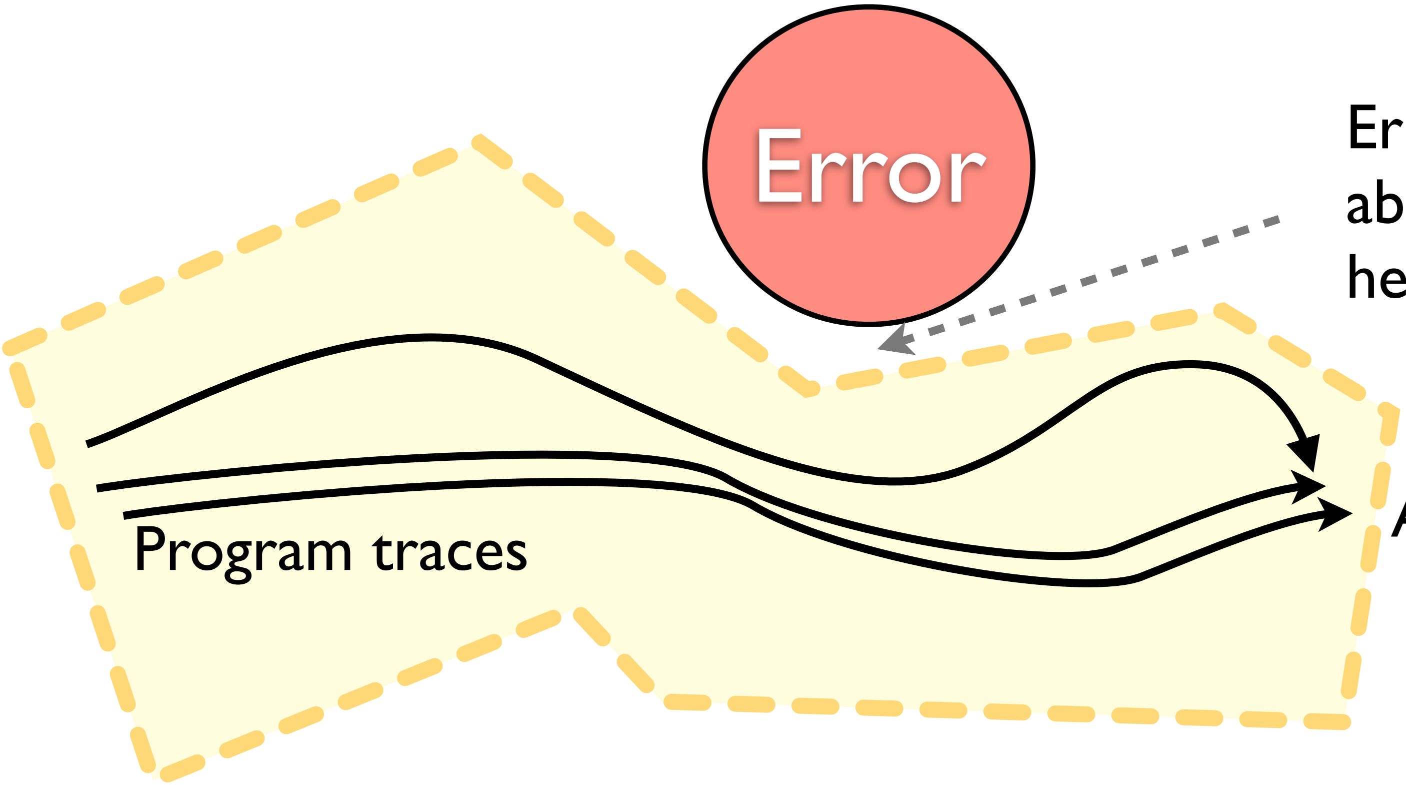
Abstract Satisfaction

Vijay D’Silva, <u>Leopold Haller</u>, Daniel Kroening

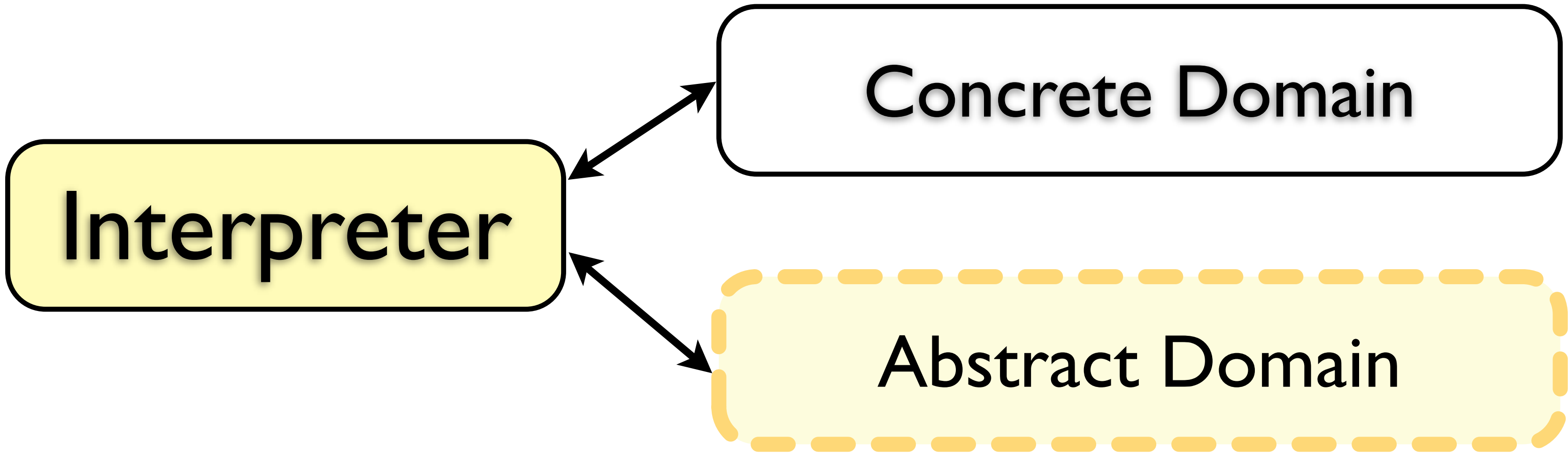
A Tale of Two Software Verification DPhils

Abstract Interpretation based Program Analysis



Error states do not overlap abstract representation, hence program is safe

Abstract representation



Interpreter

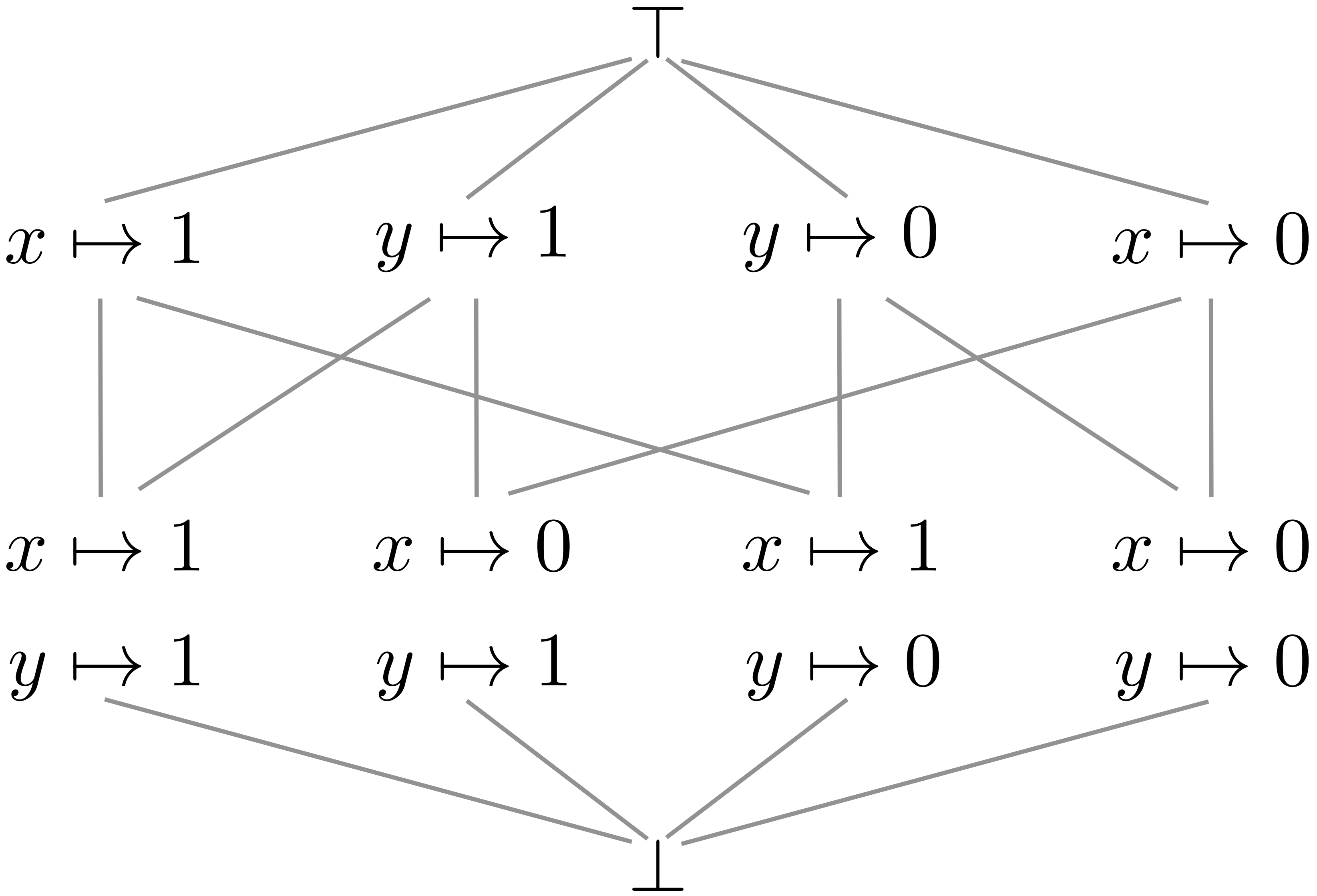
Concrete Domain

Abstract Domain

Test for all inputs and memory states

Abstract analysis

Lattice of Boolean Constants



$\top$

$x \mapsto 1$ $y \mapsto 1$ $y \mapsto 0$ $x \mapsto 0$

$x \mapsto 1$, $y \mapsto 1$ $x \mapsto 0$, $y \mapsto 1$ $x \mapsto 1$, $y \mapsto 0$ $x \mapsto 0$, $y \mapsto 0$

$\bot$

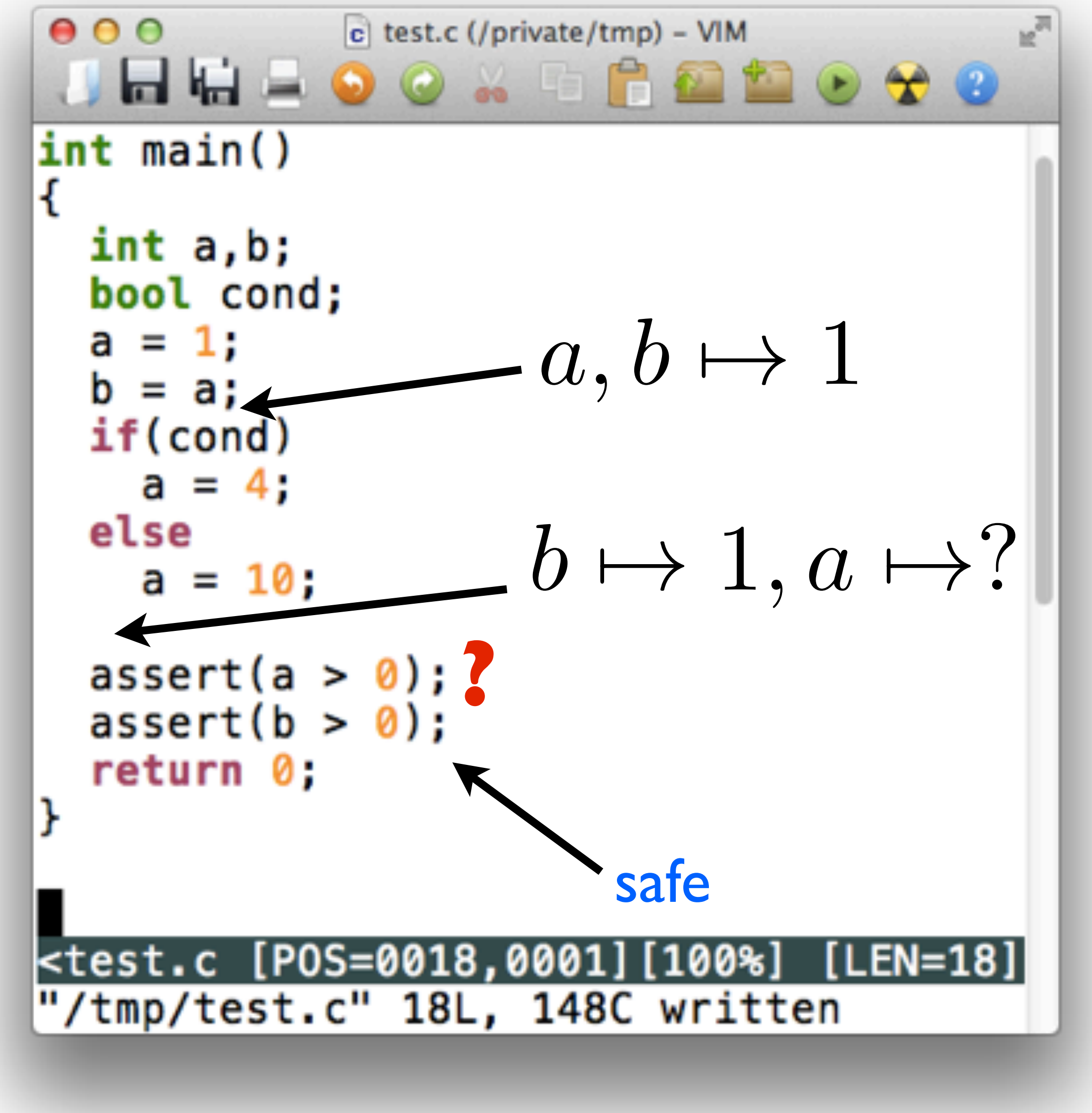
Abstract interpretation operates over lattices

Domain of Constants

Constant Propagation

$$Var \rightarrow IntVals \cup \{?\}$$

Analyse by applying abstract transformers



```
int main()
{
  int a,b;
  bool cond;
  a = 1;
  b = a;
  if(cond)
    a = 4;
  else
    a = 10;

  assert(a > 0);
  assert(b > 0);
  return 0;
}

<test.c [POS=0018,0001][100%]  [LEN=18]
"/tmp/test.c" 18L, 148C written
```

$a, b \mapsto 1$

$b \mapsto 1, a \mapsto ?$

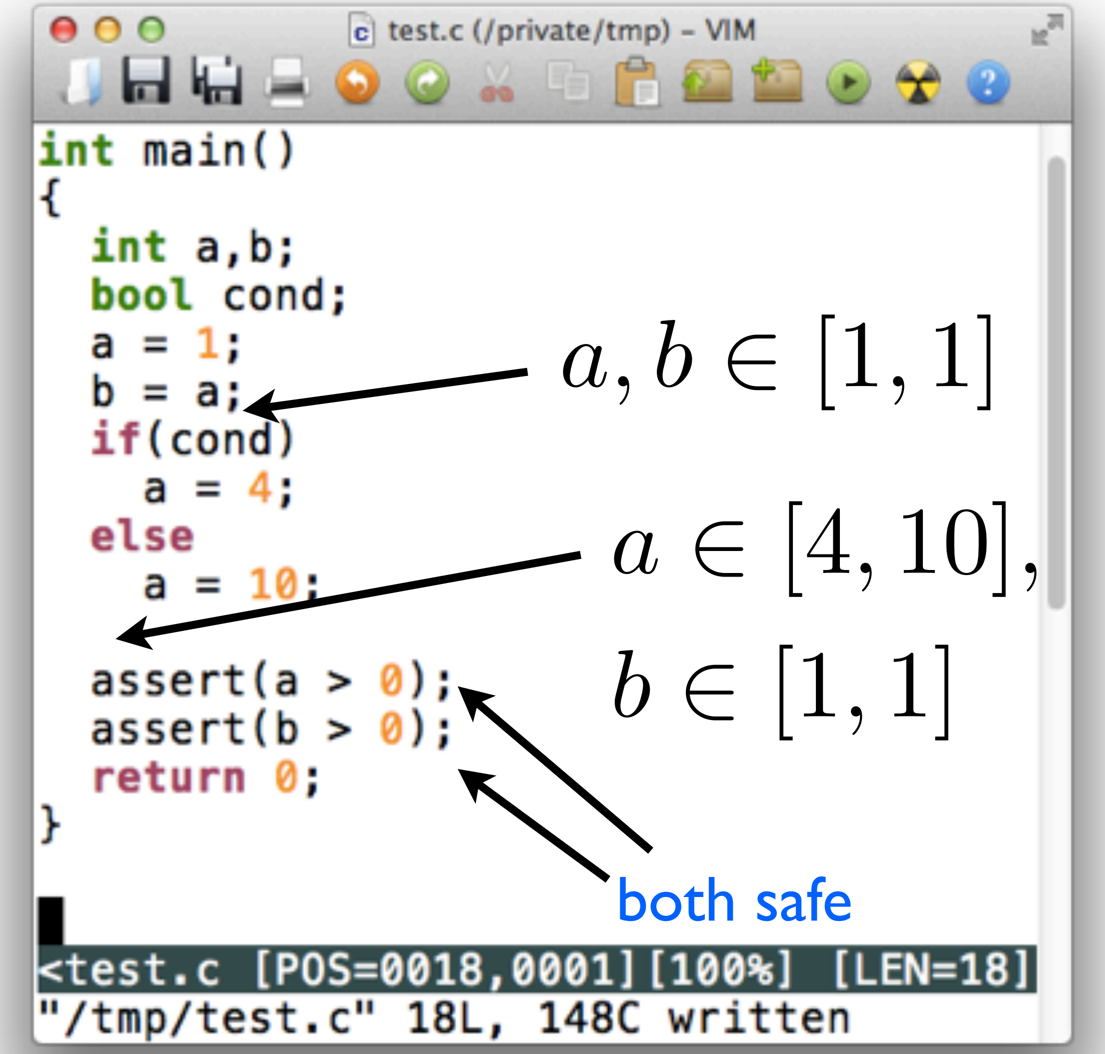
?

safe

Efficient, but imprecise

Domain of Intervals

$$Var \mapsto \{[l, u] \mid l, u \in IntVals\}$$



```
int main()
{
  int a,b;
  bool cond;
  a = 1;
  b = a;
  if(cond)
    a = 4;
  else
    a = 10;

  assert(a > 0);
  assert(b > 0);
  return 0;
}
```

```
<test.c [POS=0018,0001][100%] [LEN=18]
"/tmp/test.c" 18L, 148C written
```

$a, b \in [1, 1]$

$a \in [4, 10],$

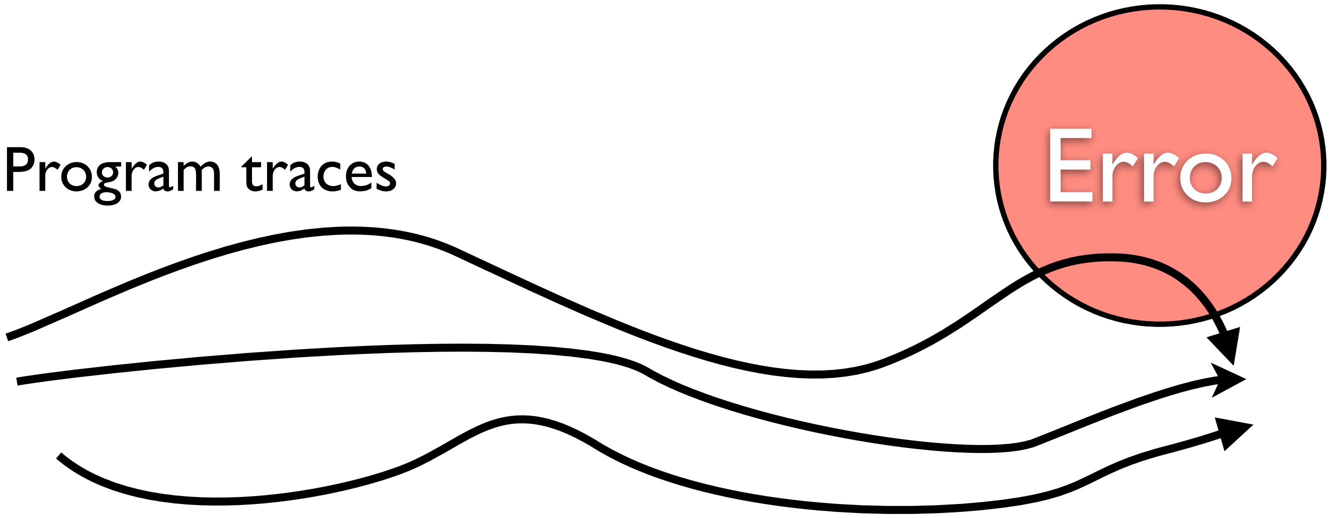
$b \in [1, 1]$

both safe

Efficient, but imprecise

How can I get abstract domains to be more precise?

SAT Solving



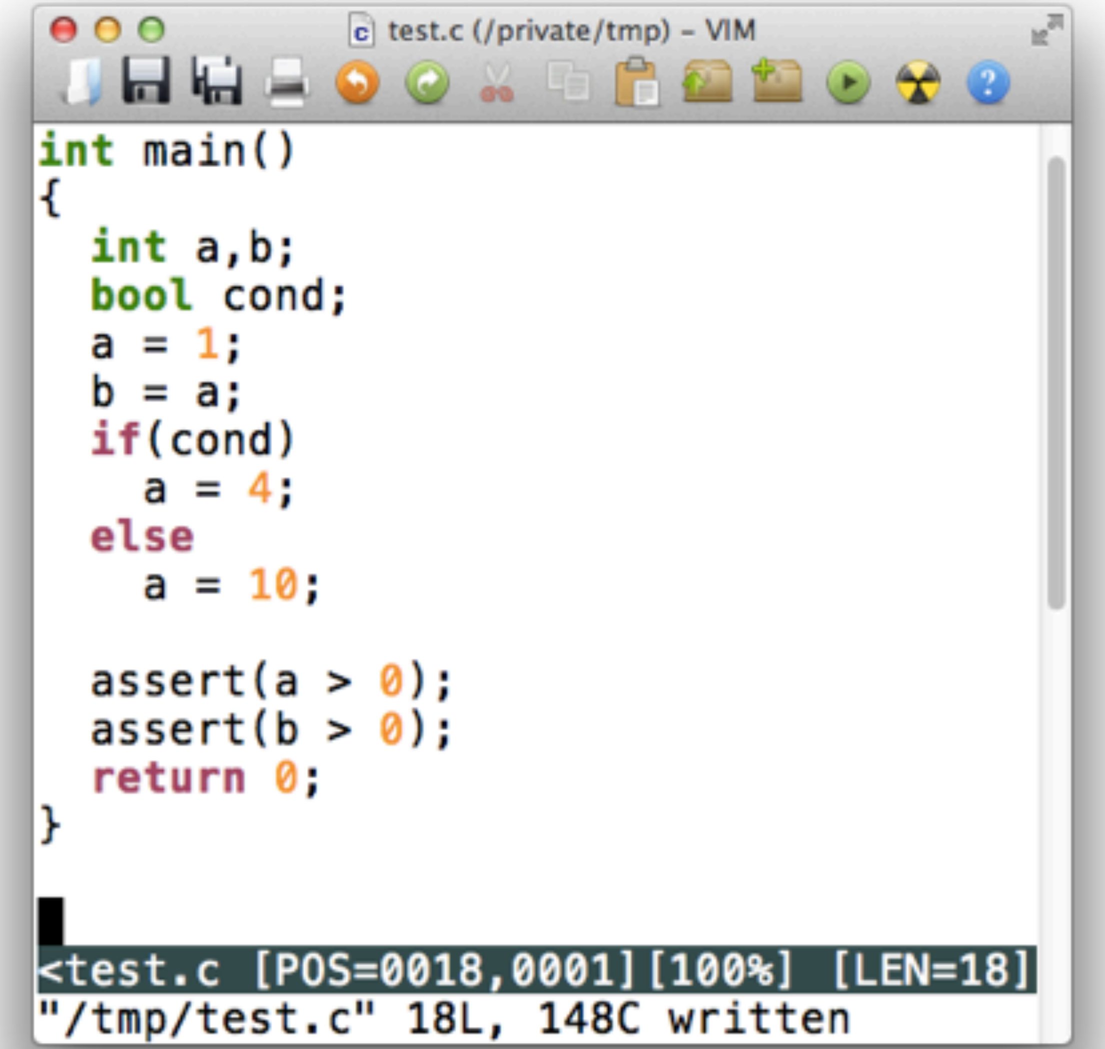
Build a logical formula that encodes program semantics

$$isTrace(t) \wedge error(t)$$

Solve satisfiability: Does there exist a *t* that makes the above formula true.

Fast *SAT solvers* exist that can solve this question.

SAT encoding

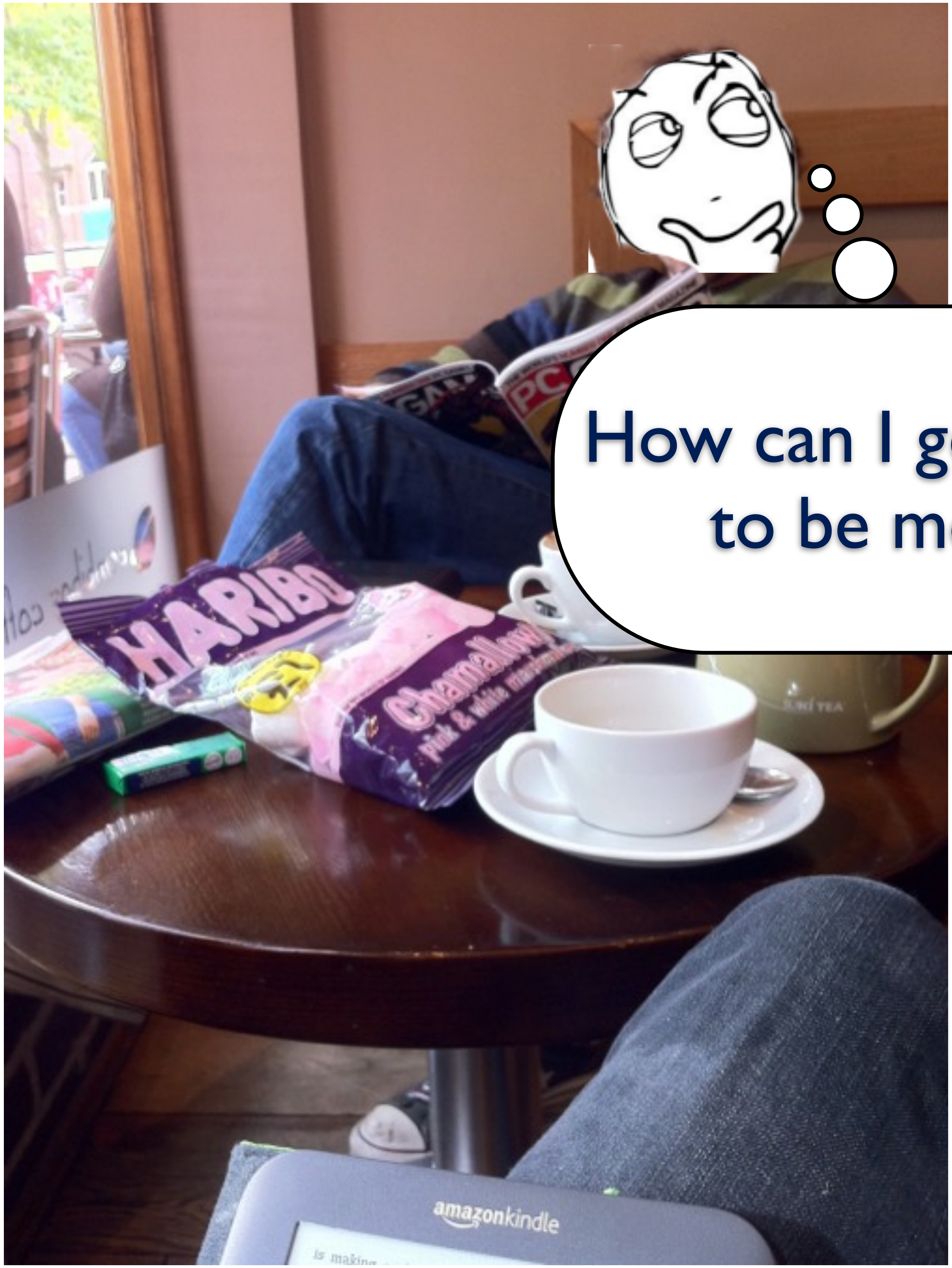


```
int main()
{
  int a,b;
  bool cond;
  a = 1;
  b = a;
  if(cond)
    a = 4;
  else
    a = 10;

  assert(a > 0);
  assert(b > 0);
  return 0;
}

<test.c  [POS=0018,0001][100%]  [LEN=18]
"/tmp/test.c" 18L, 148C written
```

$$a_0 = 1 \wedge$$
$$b_0 = a_0 \wedge$$
$$(c_0 \rightarrow a_1 = 4) \wedge$$
$$(\neg c_0 \rightarrow a_1 = 10) \wedge$$
$$(a_1 \leq 0 \vee b_0 \leq 0)$$

Translate inequalities and equalities to circuits

Precise, but not scalable



How can I get my SAT solver to be more efficient?

Our initial project

Let’s *combine* SAT solving and abstract interpretation to achieve both efficiency and precision?

Partial assignments in SAT

The main data-structure in a SAT solver is a partial assignment from variables to truth values.

This assignment is extended using *deductions* and *decisions*.

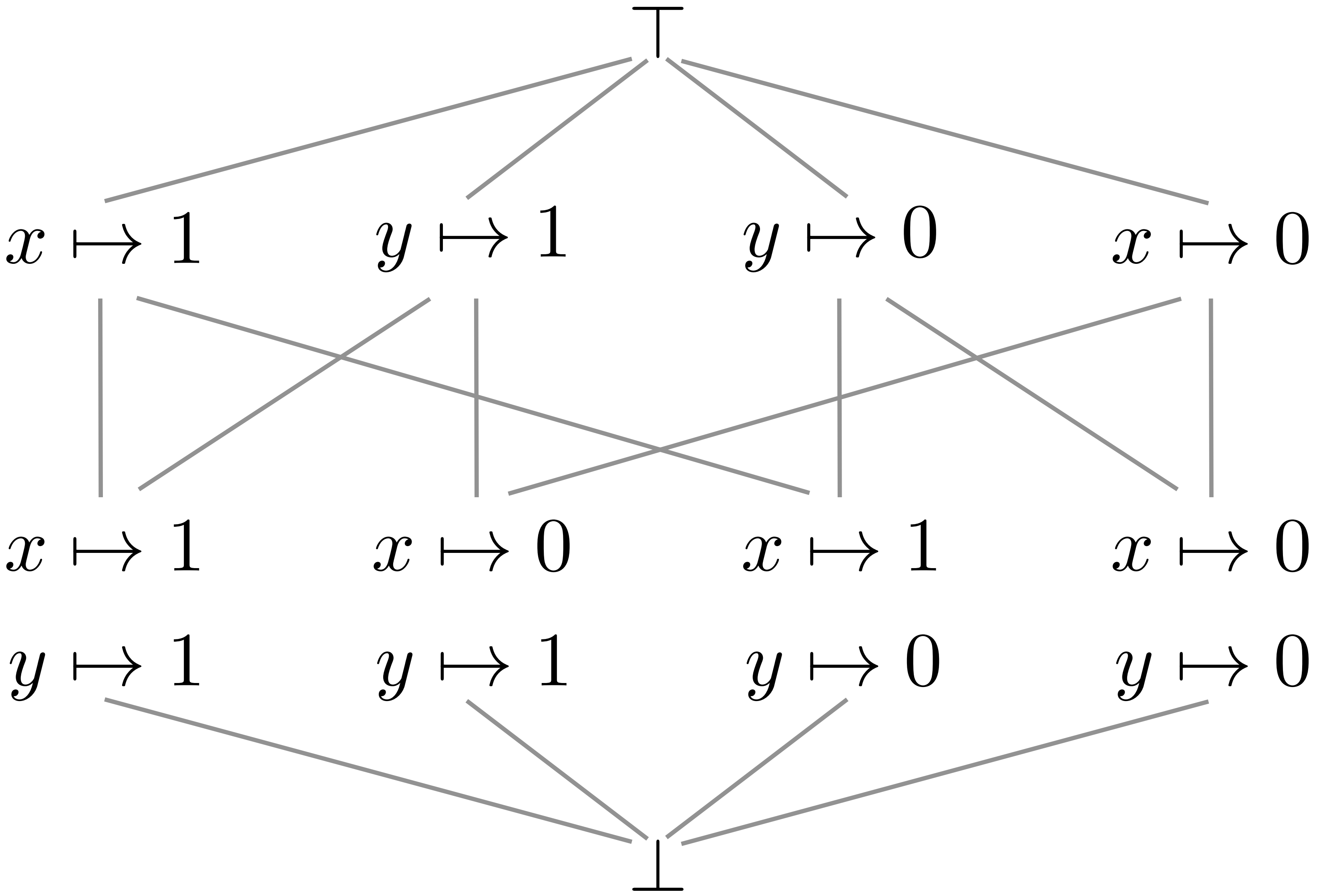
$$x \mapsto 1$$

$$y \mapsto 1 \qquad y \mapsto 0$$

$$\begin{array}{l} x \mapsto 1 \\ y \mapsto 1 \end{array} \qquad \begin{array}{l} x \mapsto 1 \\ y \mapsto 0 \end{array} \qquad x \mapsto 0$$

$$\begin{array}{l} x \mapsto 0 \\ y \mapsto 1 \end{array} \qquad \begin{array}{l} x \mapsto 0 \\ y \mapsto 0 \end{array}$$

SAT operates over a lattice



$\top$

$x \mapsto 1$ $y \mapsto 1$ $y \mapsto 0$ $x \mapsto 0$

$x \mapsto 1$ $x \mapsto 0$ $x \mapsto 1$ $x \mapsto 0$

$y \mapsto 1$ $y \mapsto 1$ $y \mapsto 0$ $y \mapsto 0$

$\bot$

SAT operates the Boolean constants lattice

The Unit Rule

$p \mapsto$ t

$q \mapsto$ f

$r \mapsto$ f

Unit Rule

$$\ldots \wedge (\neg p \vee q \vee r \vee \neg w) \wedge \ldots$$

The Unit Rule

$p \mapsto \mathsf{t}$

$q \mapsto \mathsf{f}$

$r \mapsto \mathsf{f}$

Unit Rule

$$\ldots \wedge (\neg p \vee q \vee r \vee \neg w) \wedge \ldots$$

The Unit Rule

$p \mapsto \mathsf{t}$

$q \mapsto \mathsf{f}$

$r \mapsto \mathsf{f}$

Unit Rule

$$\ldots \wedge (\neg p \vee q \vee r \vee \neg w) \wedge \ldots$$

The Unit Rule

$p \mapsto \mathsf{t}$

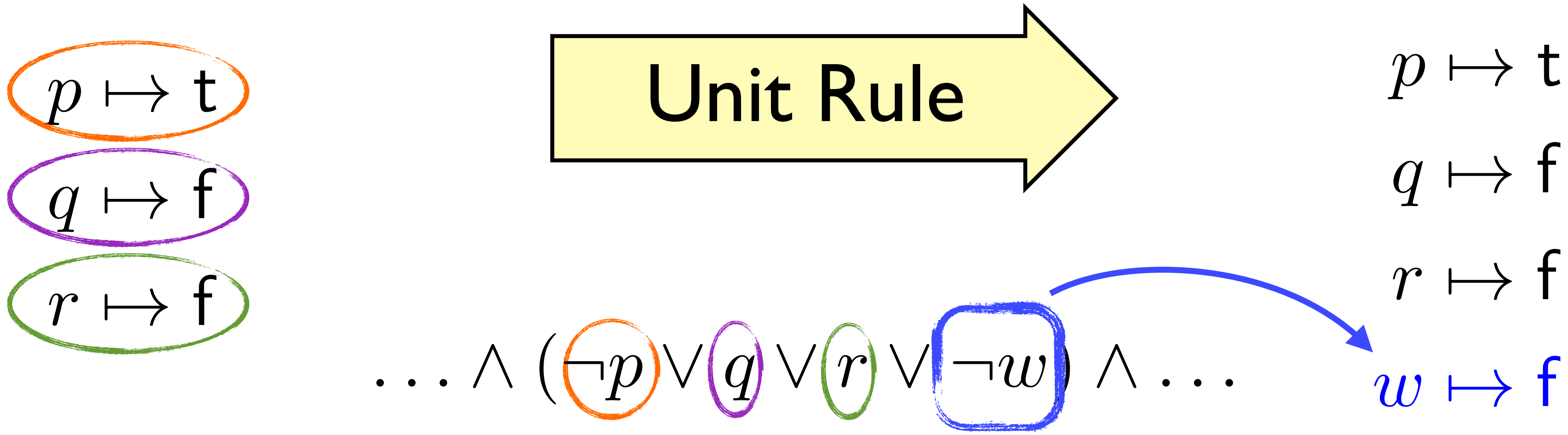
$q \mapsto \mathsf{f}$

$r \mapsto \mathsf{f}$

Unit Rule

$$\ldots \wedge (\neg p \vee q \vee r \vee \neg w) \wedge \ldots$$

The Unit Rule



$p \mapsto \mathsf{t}$

$q \mapsto \mathsf{f}$

$r \mapsto \mathsf{f}$

Unit Rule

$$\ldots \wedge (\neg p \vee q \vee r \vee \boxed{\neg w}) \wedge \ldots$$

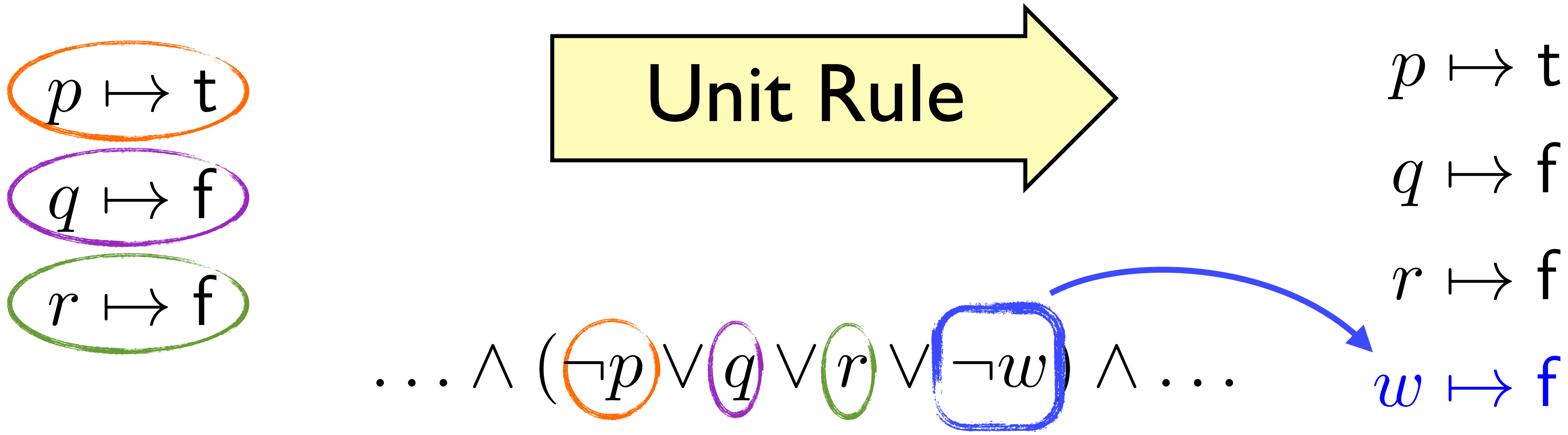
$p \mapsto \mathsf{t}$

$q \mapsto \mathsf{f}$

$r \mapsto \mathsf{f}$

$w \mapsto \mathsf{f}$

The Unit Rule



$p \mapsto \text{t}$

$q \mapsto \text{f}$

$r \mapsto \text{f}$

Unit Rule

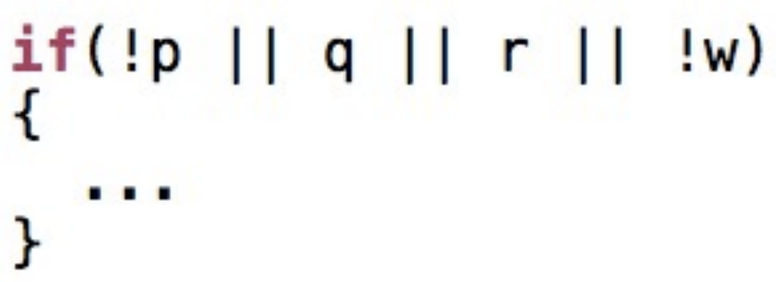
$p \mapsto \text{t}$

$q \mapsto \text{f}$

$r \mapsto \text{f}$

$w \mapsto \text{f}$

$$\ldots \wedge (\neg p \vee q \vee r \vee \boxed{\neg w}) \wedge \ldots$$



```
if(!p || q || r || !w)
{
   ...
}
```

The Unit Rule

$p \mapsto \mathsf{t}$

$q \mapsto \mathsf{f}$

$r \mapsto \mathsf{f}$

Unit Rule

$p \mapsto \mathsf{t}$

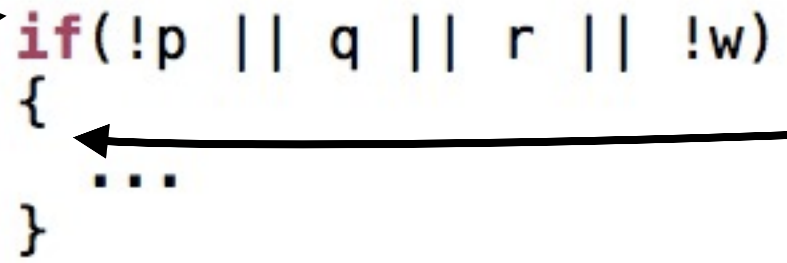
$q \mapsto \mathsf{f}$

$r \mapsto \mathsf{f}$

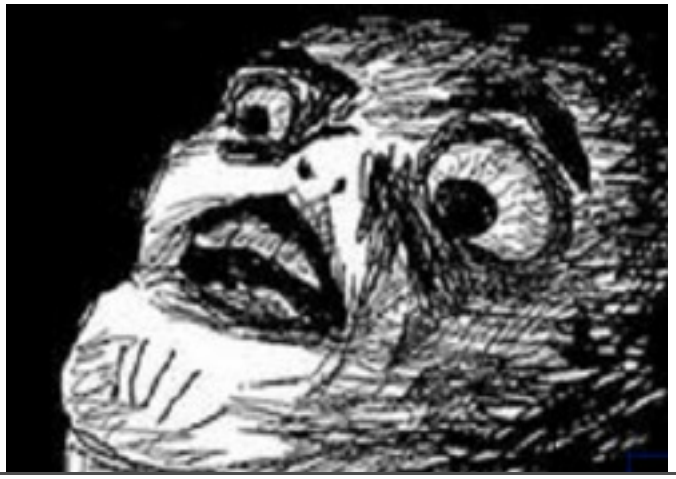
$w \mapsto \mathsf{f}$

$$\ldots \wedge (\neg p \vee q \vee r \vee \boxed{\neg w}) \wedge \ldots$$

The unit rule is the best abstract transformer over the lattice!



```
if(!p || q || r || !w)
{
   ...
}
```



Decisions

No deductions are possible on the following formula.

$$\phi = (w \vee q) \wedge (\neg w \vee q)$$

Hence a decision is made:

$$q \mapsto \text{false}$$

From which both of the following can be deduced:

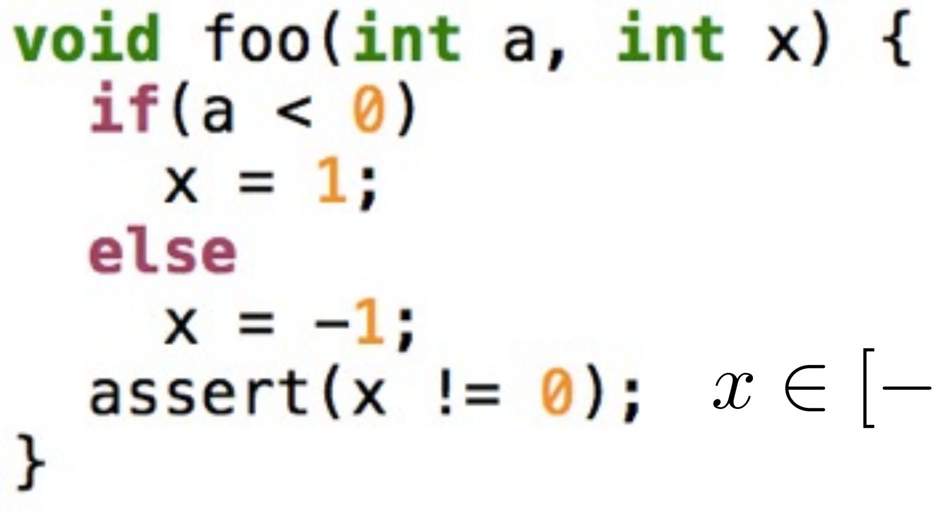
$$w \mapsto \text{true} \qquad w \mapsto \text{false}$$

The solver backtracks and learns that q must be true, essentially, we expanded the formula into

$$(q \wedge \phi) \vee (\neg q \wedge \phi)$$

Trace partitioning

Trace partitioning is an well-known refinement technique in abstract interpretation

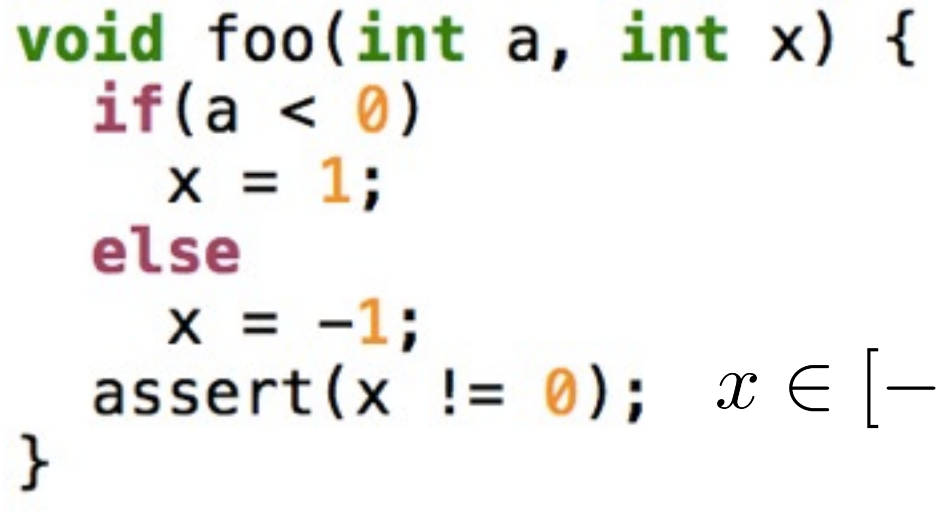


```
void foo(int a, int x) {
  if(a < 0)
    x = 1;
  else
    x = -1;
  assert(x != 0);
}
```

$x \in [-1, 1]$ too imprecise!

Trace partitioning

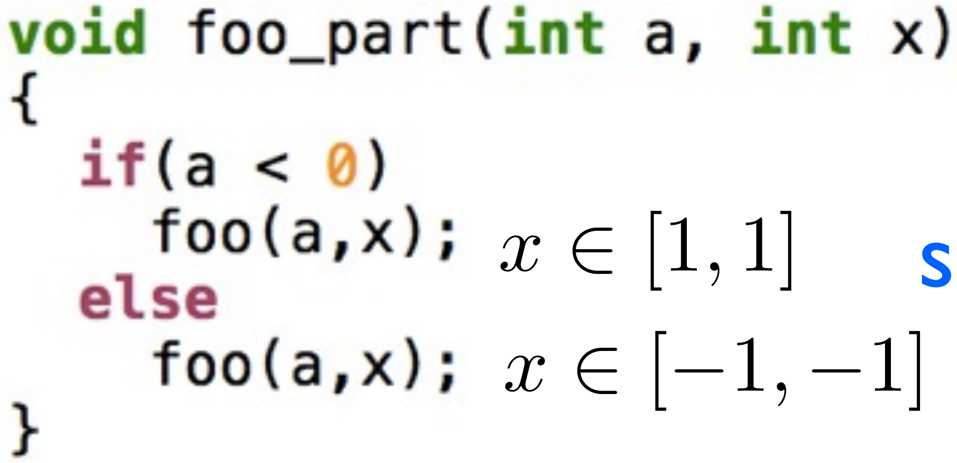
Trace partitioning is an well-known refinement technique in abstract interpretation



```
void foo(int a, int x) {
  if(a < 0)
    x = 1;
  else
    x = -1;
  assert(x != 0);
}
```

$x \in [-1, 1]$ too imprecise!

Apply partitioning:



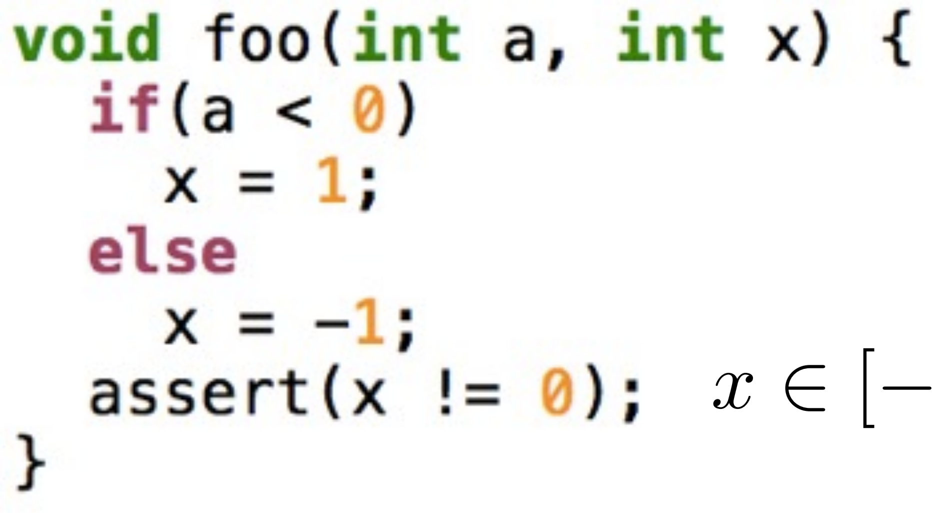
```
void foo_part(int a, int x)
{
  if(a < 0)
    foo(a,x);
  else
    foo(a,x);
}
```

$x \in [1, 1]$ safe

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

Trace partitioning

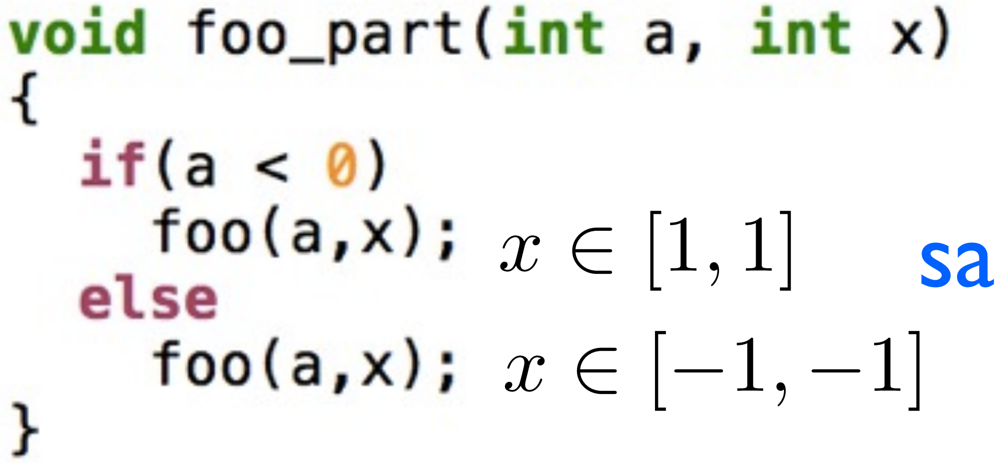
Trace partitioning is an well-known refinement technique in abstract interpretation



```
void foo(int a, int x) {
  if(a < 0)
    x = 1;
  else
    x = -1;
  assert(x != 0);
}
```

$x \in [-1, 1]$ too imprecise!

Apply partitioning:

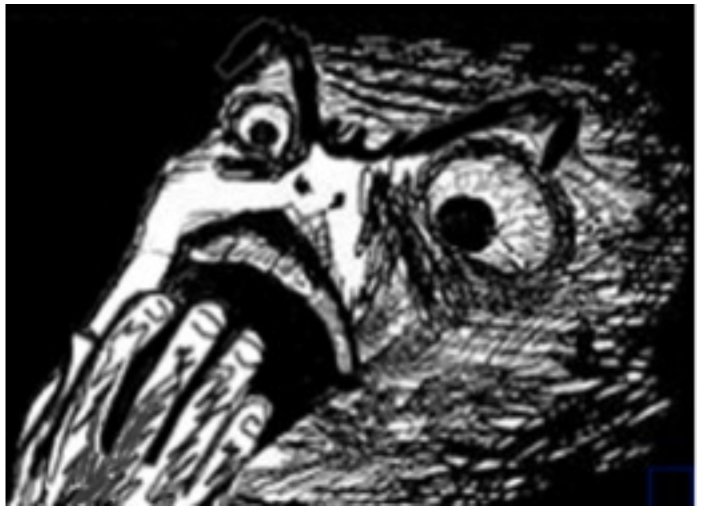


```
void foo_part(int a, int x)
{
  if(a < 0)
    foo(a,x);
  else
    foo(a,x);
}
```

$x \in [1, 1]$ safe

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

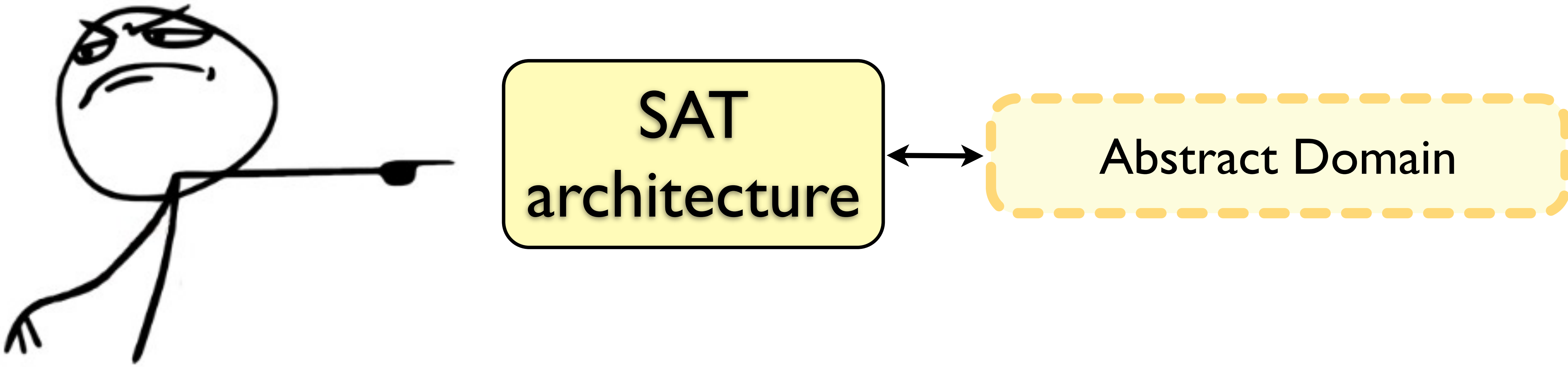
Decisions are a well-known program analysis technique!



Summary: SAT = AI

Modern SAT solvers are abstract interpreters

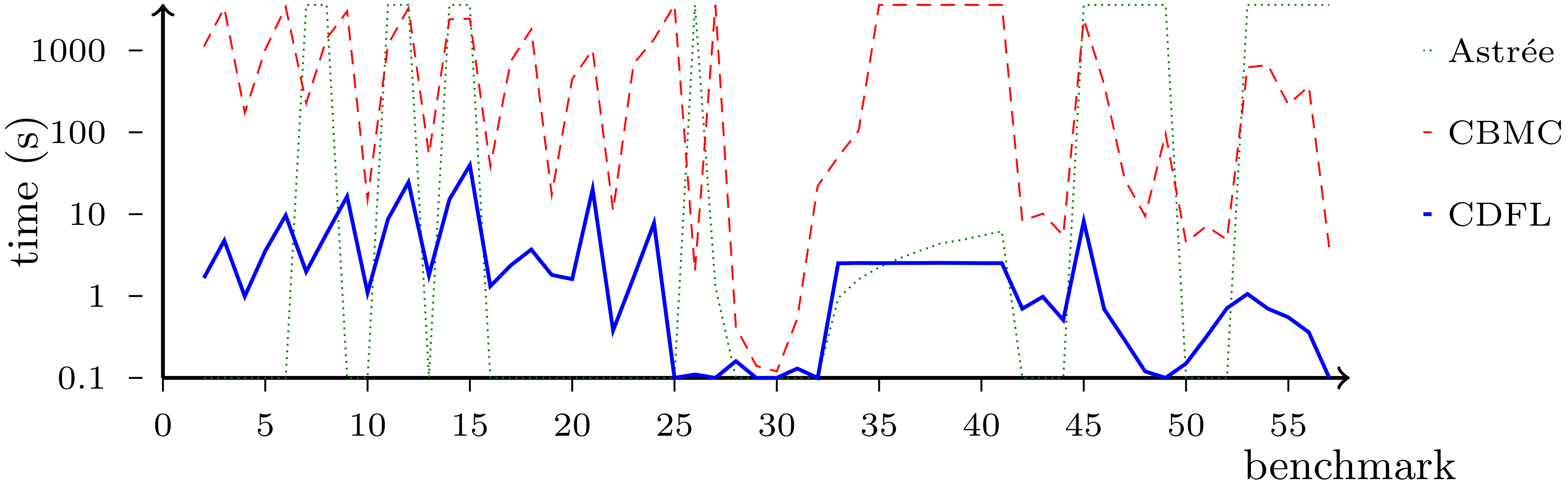
The SAT architecture is an abstract interpreter architecture that automatically and intelligently refines a base domain.



SAT architecture

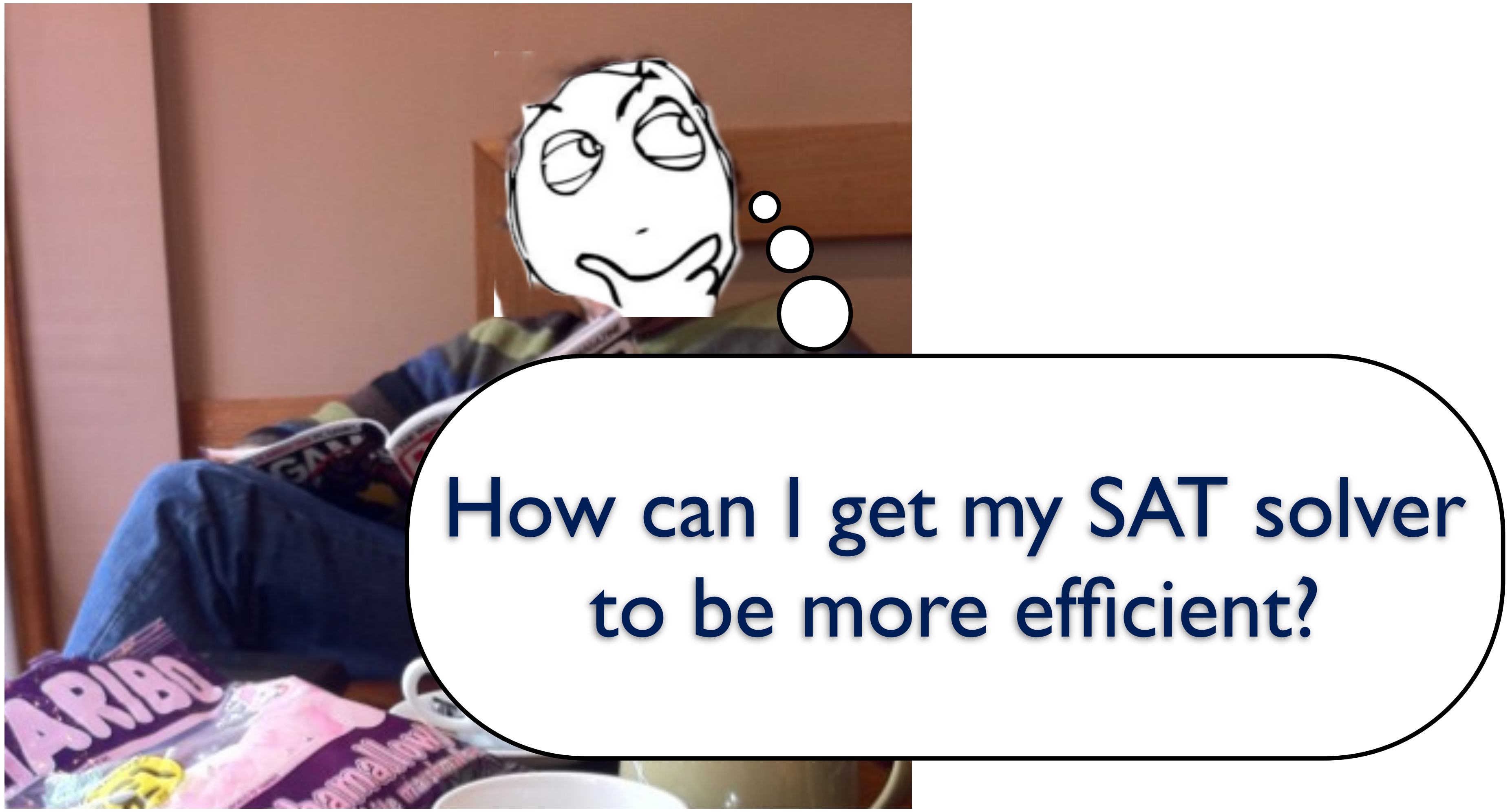
Abstract Domain

SAT over Interval Domain



Naive implementation of SAT(Intervals) applied to numeric program verification benchmarks.

On average ca. 200x faster than SAT, significantly more precise than mature abstract interpreters.



How can I get my SAT solver to be more efficient?

Choose a domain that's better suited to your problem than the Boolean constants domain!

How can I get abstract domains to be more precise?

Wrap them in the SAT architecture!

Thanks for your attention