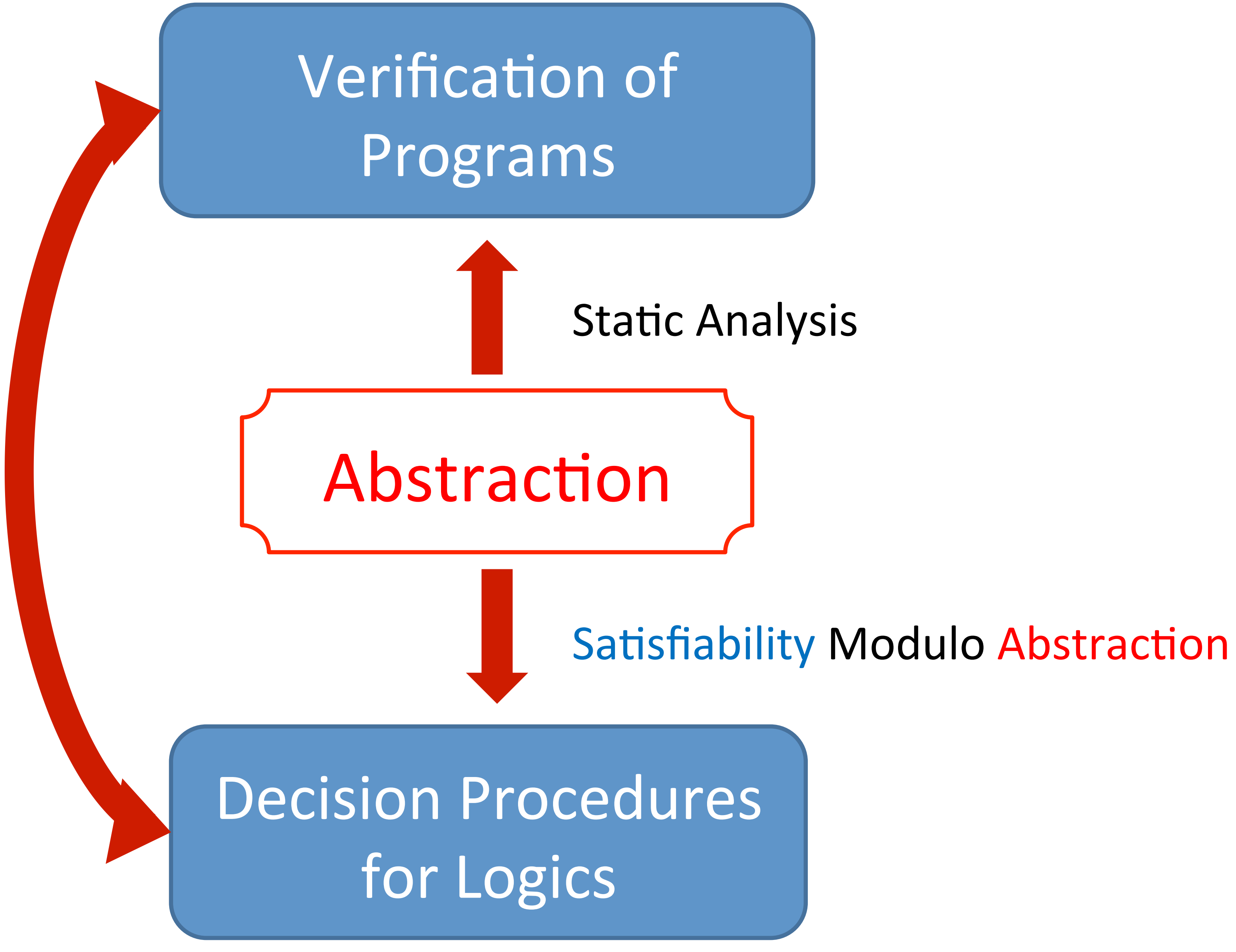
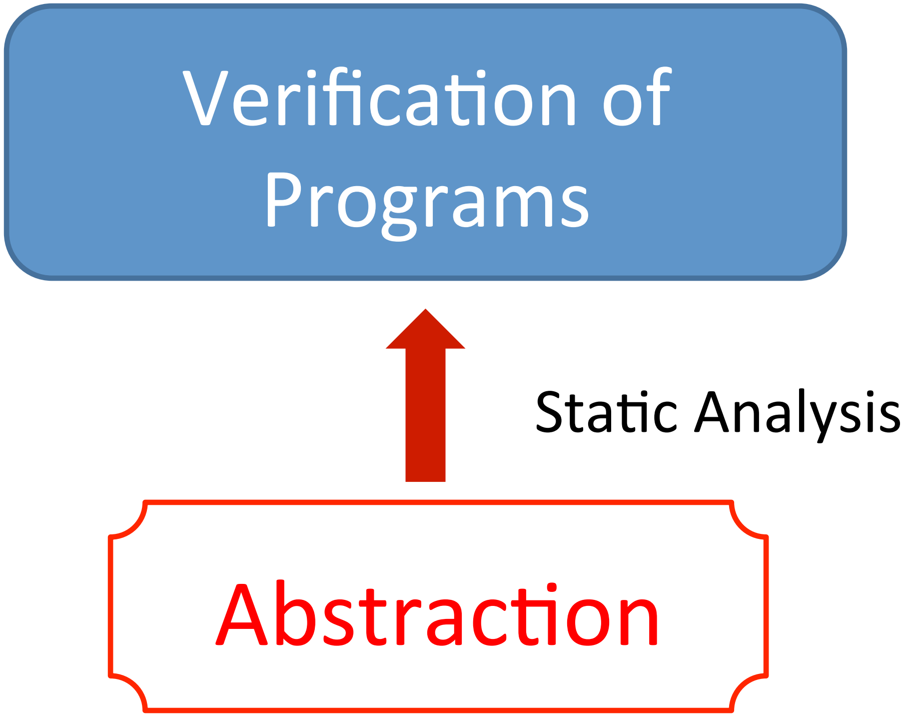
Through the Lens of Abstraction

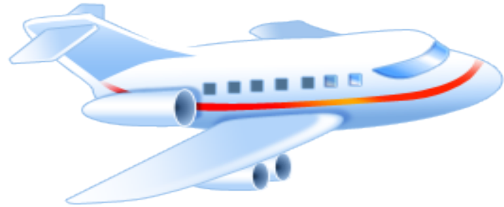
Aditya Thakur[1] Thomas Reps[1,2]

[1]University of Wisconsin–Madison

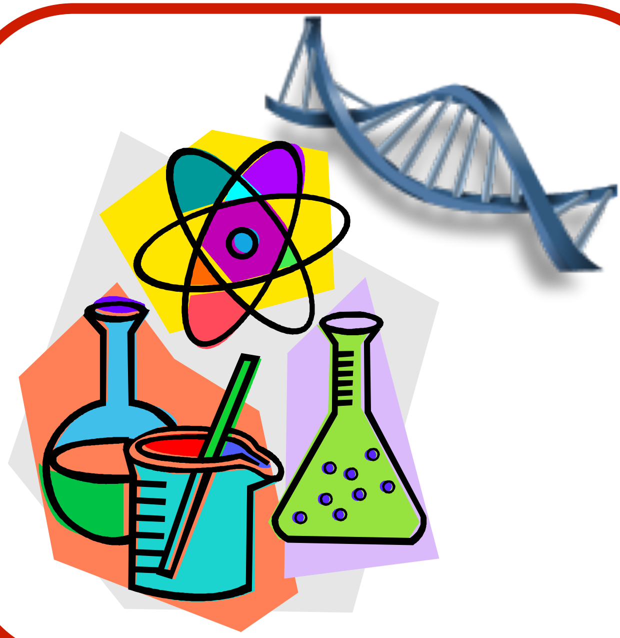
[2]GrammaTech, Inc.

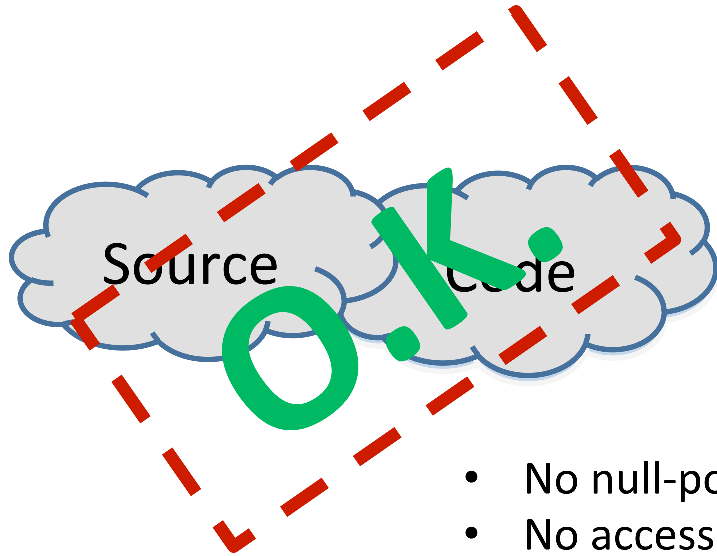

Verification of Programs
Static Analysis
Abstraction
Satisfiability Modulo Abstraction
Decision Procedures for Logics


Verification of Programs
Static Analysis
Abstraction

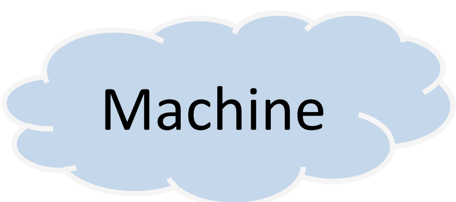


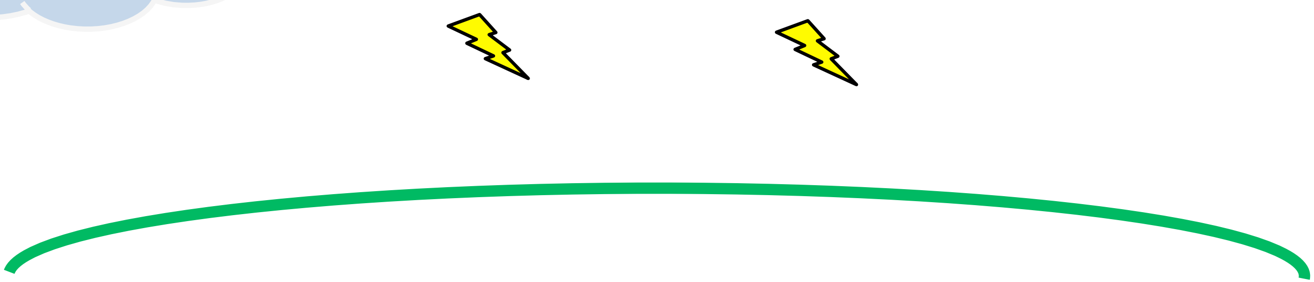

WALL ST





- No null-pointer dereferences
- No accesses outside array bounds
- No division by zero
- No stack smashing
- Data-structure invariants verified





From program paths to formulas

```
if (a0 < b0)  {
    if (a0 < c0)  {
        if (b0 < a1 || c0 < a1)  {
            ERROR:
        }
    }
}
```

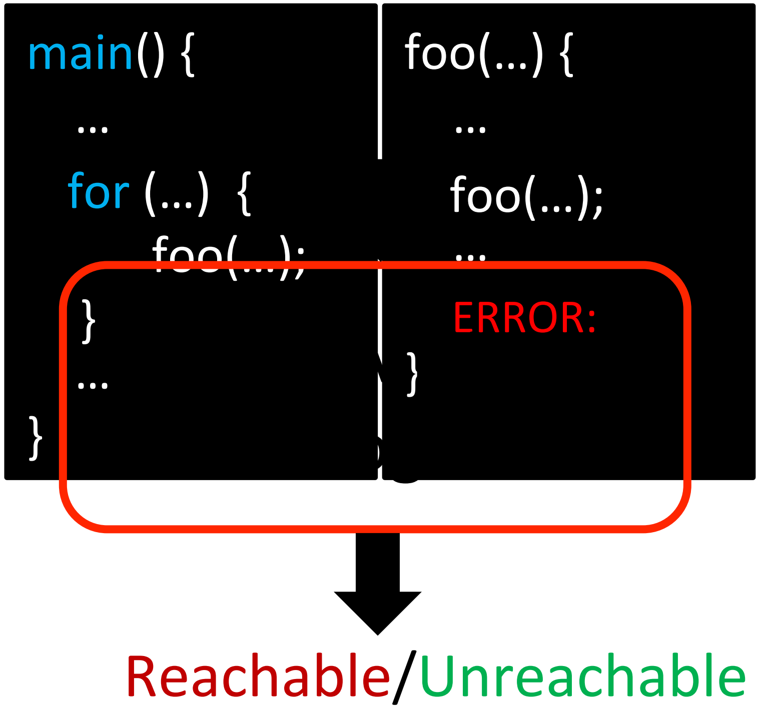
ERROR statement is reachable in program if and if

$$(a_0 < b_0) \wedge (a_0 < c_0) \wedge ((b_0 < a_1) \vee (c_0 < a_1))$$

is satisfiable

```
main() {
    …
    for (…)  {
          foo(…);
    }
    …
}
```

```
foo(…) {
    …
    foo(…);
    …
    ERROR:
}
```



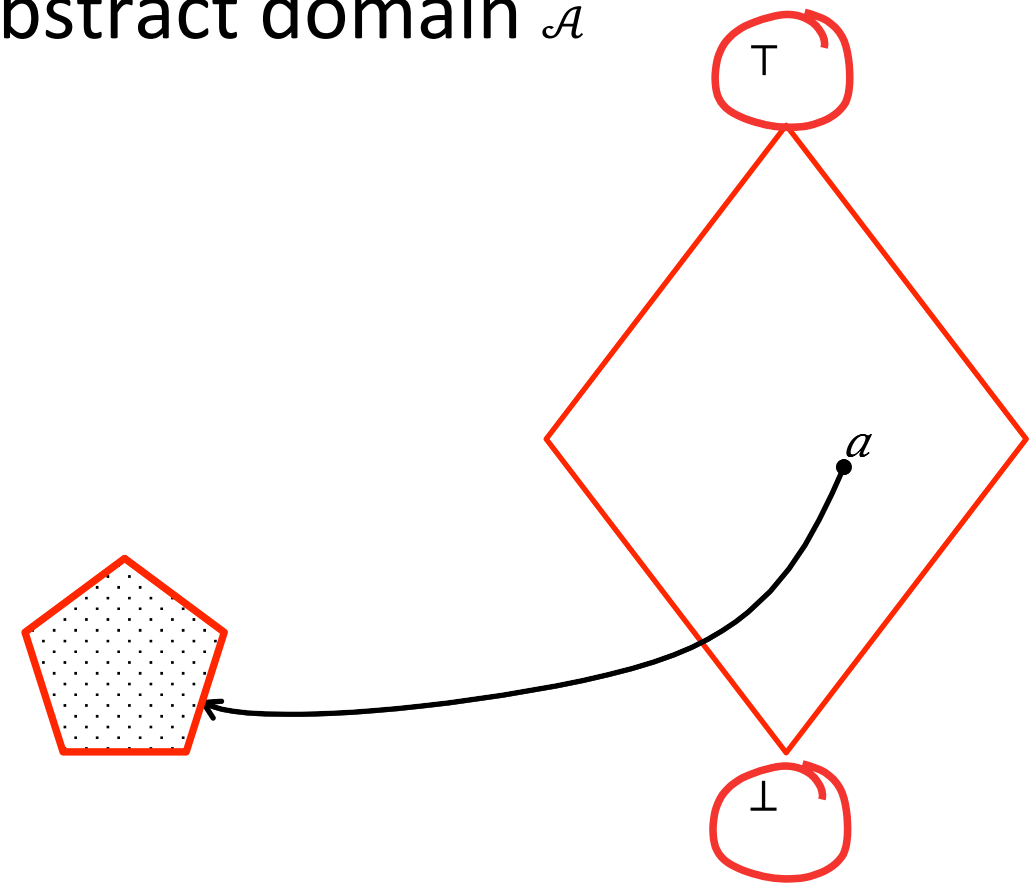
Implementing correct, precise, and scalable analyses is challenging.

Who watches the watchmen?

Abstract Interpretation [Cousot&Cousot'77]

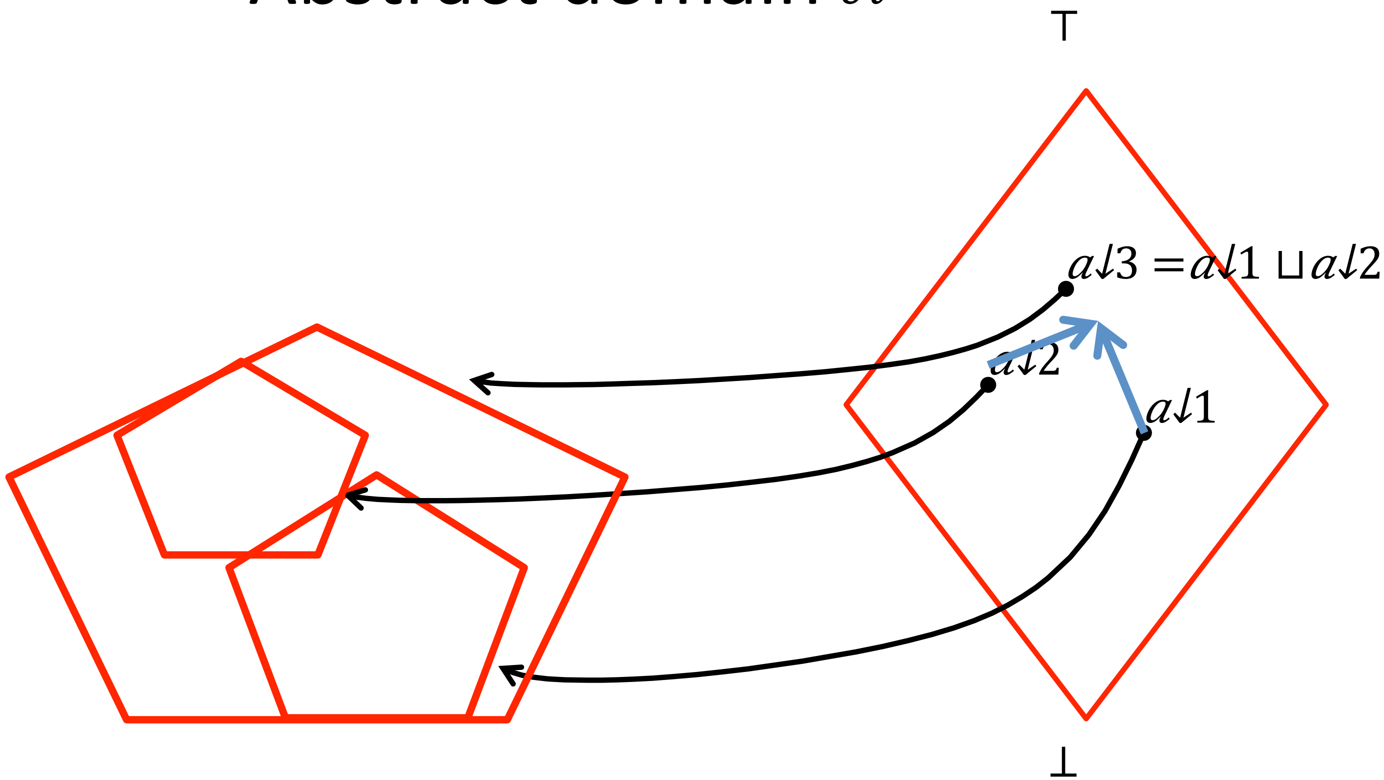
Abstract Interpretation

Abstract domain $\mathcal{A}$



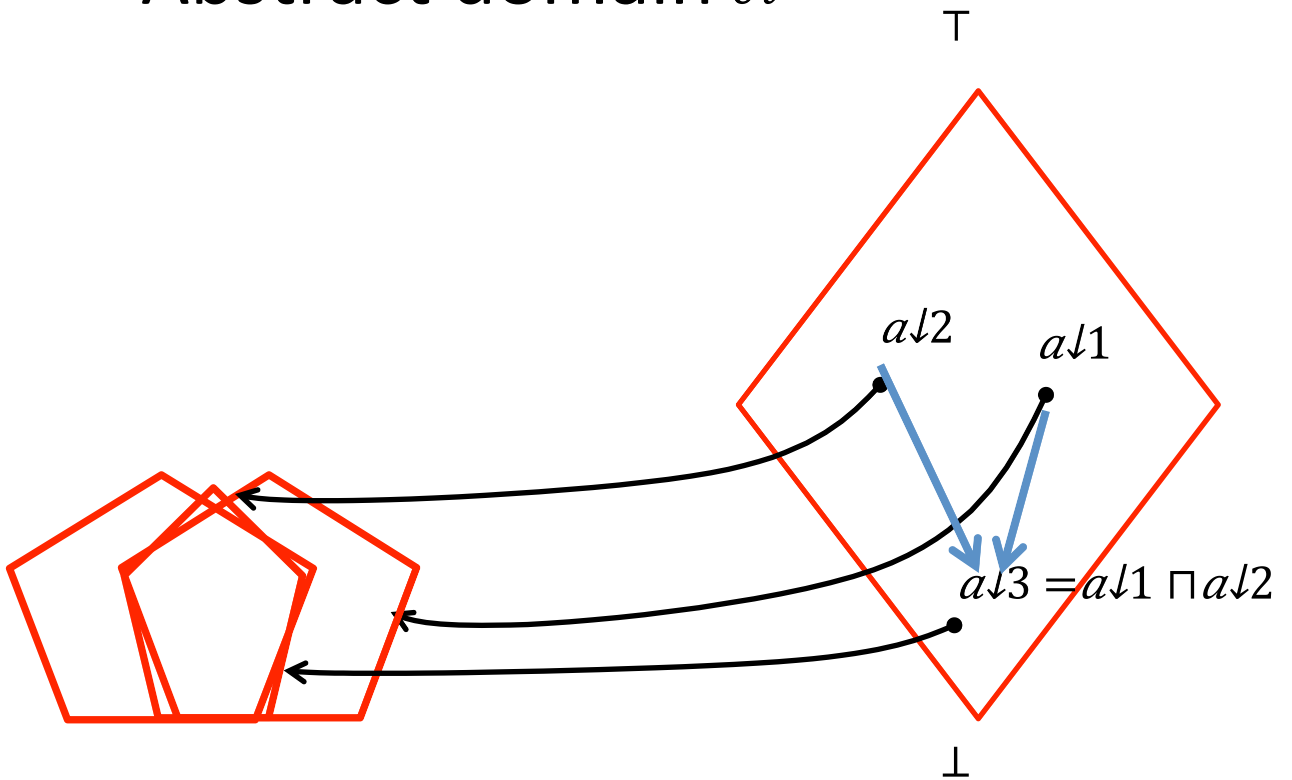
Abstract Interpretation

Abstract domain $\mathcal{A}$



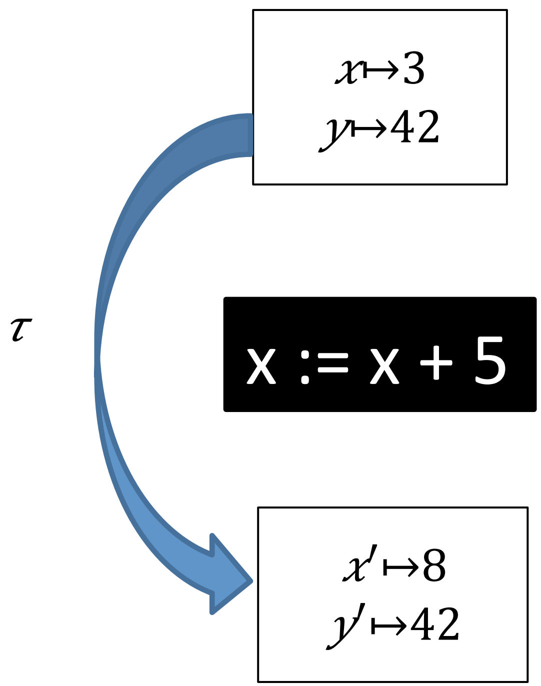
Abstract Interpretation

Abstract domain $\mathcal{A}$



Abstract Interpretation

Concrete Interpretation



Input concrete state

Concrete semantics

Output concrete state

Primed variables represent values in post-state.

Abstract Interpretation

$x \in [3,10]$
$y \in [0,420]$

Input abstract value

$\tau^{\#}$

x := x + 5

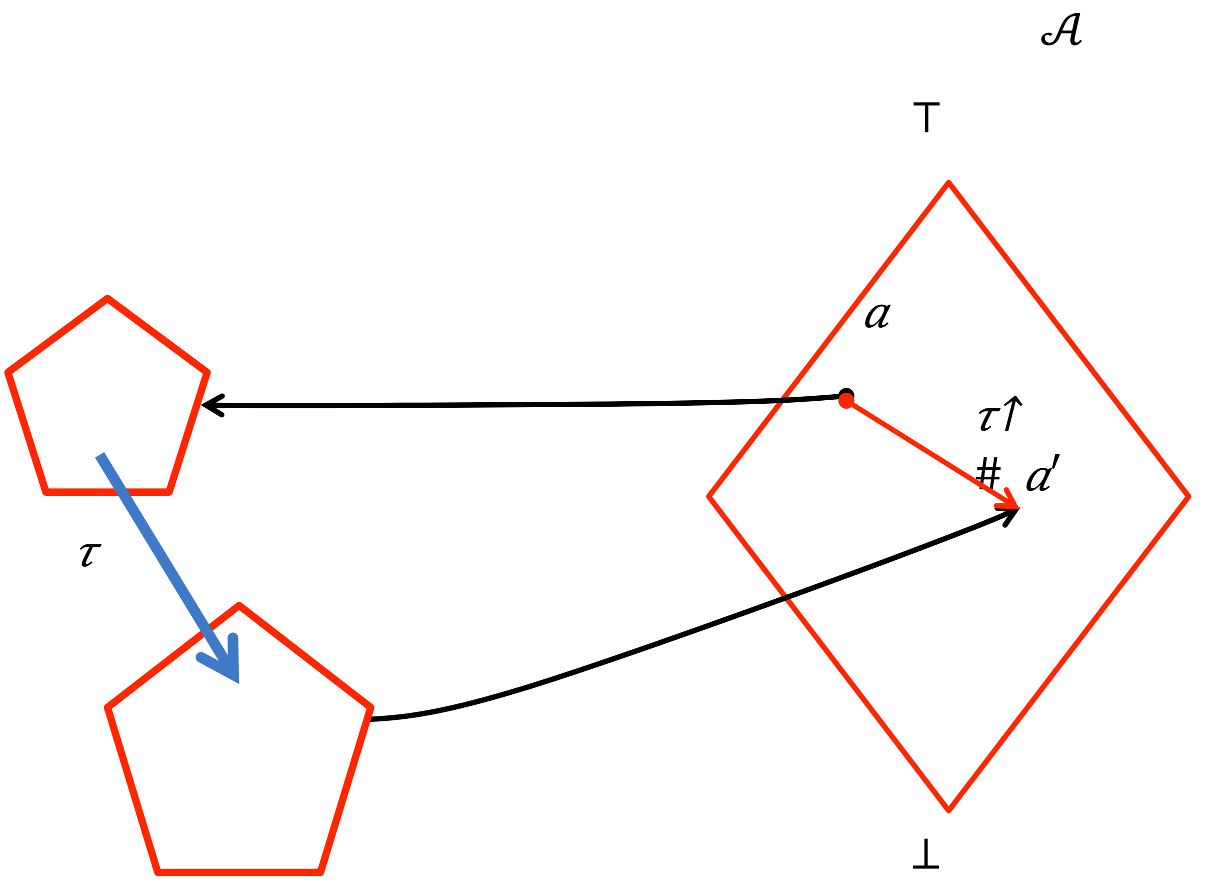
Abstract semantics

$x' \in [8,15]$
$y' \in [0,420]$

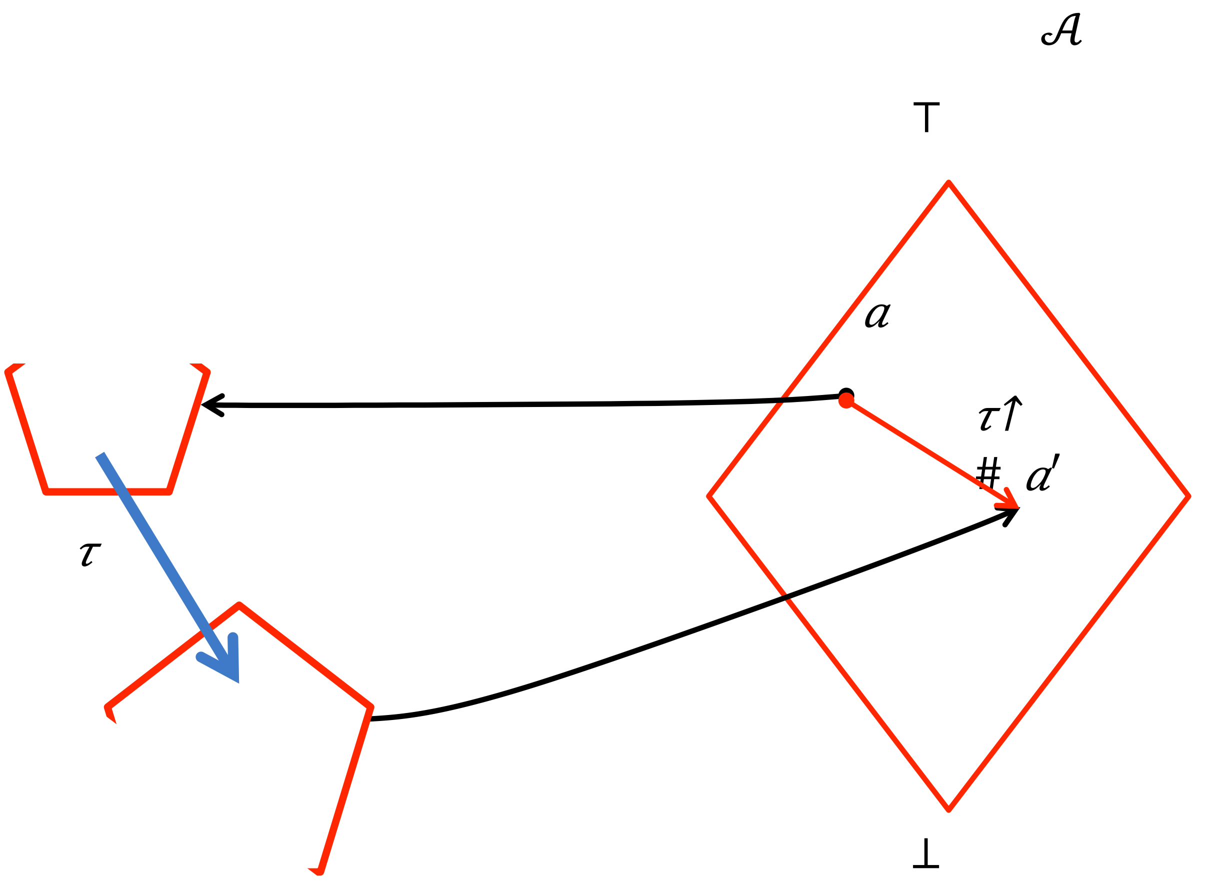
Output abstract value

Primed variables represent values in post-state.

Best Abstract Transformer [Cousot&Cousot'79]



Best Abstract Transformer [Cousot&Cousot'79]

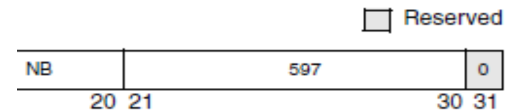


Abstract Interpretation

DAS—Decimal Adjust AL after Subtraction

Iswi

Opcode	Instruction	Description
2F	DAS	Decimal adjust AL after subtraction



Description

This instruction adjusts the result of the subtraction of two packed BCD values to create a packed BCD result. The AL register is the implied source and destination operand. The DAS instruction is only
one 2-digit, packed
instruction then ac
BCD result. If a de

Operation

IF (AL AND 0FH) >
  THEN
    AL ← AL –
    CF ← CF (
    AF ← 1;
  ELSE AF ← 0;
FI;
IF ((AL > 9FH) or (
  THEN
    AL ← AL –
    CF ← 1;
  ELSE CF ← 0;
FI;

FCOMI/FCOMIP/FUCOMI/FUCOMIP—Compare Real and Set EFLAGS (Continued)

Intel Architecture Compatibility

The FCOMI/FCOMIP/FUCOMI/FUCOMIP instructions were introduced to the Intel Architecture in the Pentium® Pro processor family and are not available in earlier Intel Architecture processors.

Operation

```
CASE (relation of operands) OF
    ST(0) > ST(i):      ZF, PF, CF ← 000;
    ST(0) < ST(i):      ZF, PF, CF ← 001;
    ST(0) = ST(i):      ZF, PF, CF ← 100;
ESAC;
IF instruction is FCOMI or FCOMIP
    THEN
        IF ST(0) or ST(i) = NaN or unsupported format
            THEN
                #IA
                IF FPUControlWord.IM = 1
                    THEN
```

– 1.

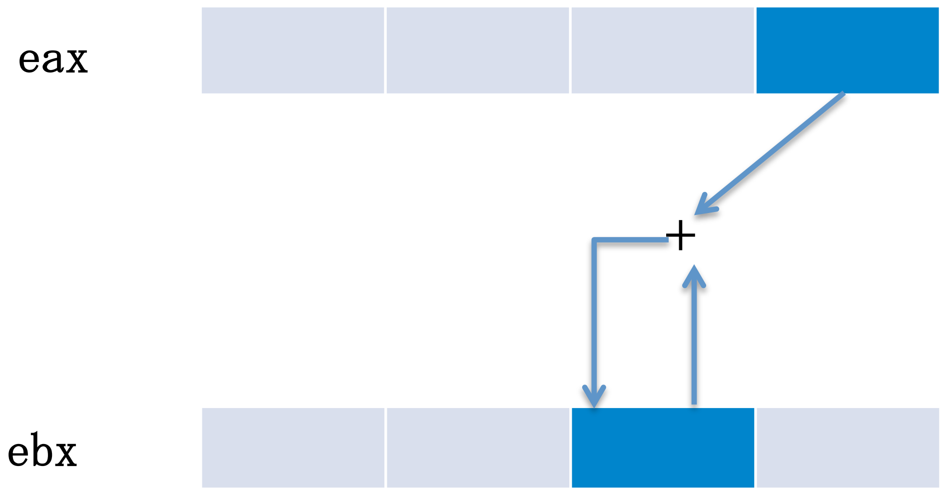
s around to r0 if required. If
rder byte(s) of that register

which rA = 0, the instruction
ossing) the data alignment
(0x00300)."

n a sequence of individual

τ: add bh, al

Adds al, the low-order byte of 32-bit register eax, to bh, the second-to-lowest byte of 32-bit register ebx



From concrete semantics to formulas

τ: add bh, al

$$\text{ebx}' = \begin{pmatrix} (\text{ebx}\ \&\ \text{0xFFFF00FF}) \\ \mid ((\text{ebx} + 256 * (\text{eax}\ \&\ \text{0xFF}))\ \&\ \text{0xFF00}) \end{pmatrix} \begin{matrix} \wedge\, \text{eax}' = \text{eax} \\ \wedge\, \text{ecx}' = \text{ecx} \end{matrix}$$

Primed variables represent values in post-state.

Abstract transformers via reinterpretation

$\mathcal{A}$: Conjunctions of bit-vector affine equalities between registers

$$\text{ebx} = \text{ecx} \in \mathcal{A}$$

$$\text{ebx}' =^{\#} \begin{pmatrix} (\text{ebx} \ \&^{\#}\ \text{0xFFFF00FF}) \\ |^{\#} ((\text{ebx} +^{\#} 256 *^{\#} (\text{eax} \ \&^{\#}\ \text{0xFF})) \ \&^{\#}\ \text{0xFF00}) \end{pmatrix} \begin{matrix} \wedge\ \text{eax}' =^{\#} \text{eax} \\ \wedge\ \text{ecx}' =^{\#} \text{ecx} \end{matrix}$$

$$\text{eax}' = \text{eax}$$
$$\wedge\, \text{ecx}' = \text{ecx}$$
$$\wedge\, 2^{24}\ \text{ebx}' = 2^{24}\ \text{ecx}'$$

Primed variables represent values in post-state.

Transformer Specification Language (TSL) [CC'08]

- A functional language for specifying the concrete semantics of an instruction set
  - Ia32, ppc32, x64, arm, llvm
- Ability to provide reinterpretation for each concrete basetype and operator
  - This is done once *per analysis*,
    not once per (analysis, instruction set)
- Over 20 analyses implemented using this framework
  - Intervals, def-use, affine-relation analysis, etc.

Not Best Abstract Transformer

$\mathcal{A}$: Conjunctions of bit-vector affine equalities between registers

$$\text{ebx} = \text{ecx} \in \mathcal{A}$$

$$\text{ebx}' = \begin{pmatrix} (\text{ebx}\ \&\ \text{0xFFFF00FF}) \\ \mid ((\text{ebx} + 256 * (\text{eax}\ \&\ \text{0xFF}))\ \&\ \text{0xFF00}) \end{pmatrix} \begin{matrix} \wedge\ \text{eax}' = \text{eax} \\ \wedge\ \text{ecx}' = \text{ecx} \end{matrix}$$

$$\text{eax}' = \text{eax}$$
$$\wedge\ \text{ecx}' = \text{ecx}$$
$$\wedge\ 2^{24}\ \text{ebx}' = 2^{24}\ \text{ecx}'$$

Best Abstract Transformer

$\mathcal{A}$: Conjunctions of bit-vector affine equalities between registers

$$\mathrm{ebx} = \mathrm{ecx} \in \mathcal{A}$$

$$\mathrm{ebx}' = \begin{pmatrix} (\mathrm{ebx}\ \&\ \mathrm{0xFFFF00FF}) \\ \mid ((\mathrm{ebx} + 256 * (\mathrm{eax}\ \&\ \mathrm{0xFF}))\ \&\ \mathrm{0xFF00}) \end{pmatrix} \begin{matrix} \wedge\ \mathrm{eax}' = \mathrm{eax} \\ \wedge\ \mathrm{ecx}' = \mathrm{ecx} \end{matrix}$$

$$\mathrm{eax}' = \mathrm{eax}$$
$$\wedge\, \mathrm{ecx}' = \mathrm{ecx}$$
$$\wedge\, 2^{24}\ \mathrm{ebx}' = 2^{24}\ \mathrm{ecx}'$$
$$\wedge\, 2^{16}\ \mathrm{ebx}' = 2^{16}\ \mathrm{ecx}' + 2^{24}\ \mathrm{e}$$

Abstract transformers for instruction sequence

$\mathcal{A}$: Conjunctions of bit-vector affine equalities between registers

$\top$

τ_1 : push ebx — $\tau_1^{\#}$

$\top$

τ_2 : pop eax — $\tau_2^{\#}$

$\top$

Abstract transformers for instruction sequence

$\mathcal{A}$: Conjunctions of bit-vector affine equalities between registers

$\top$

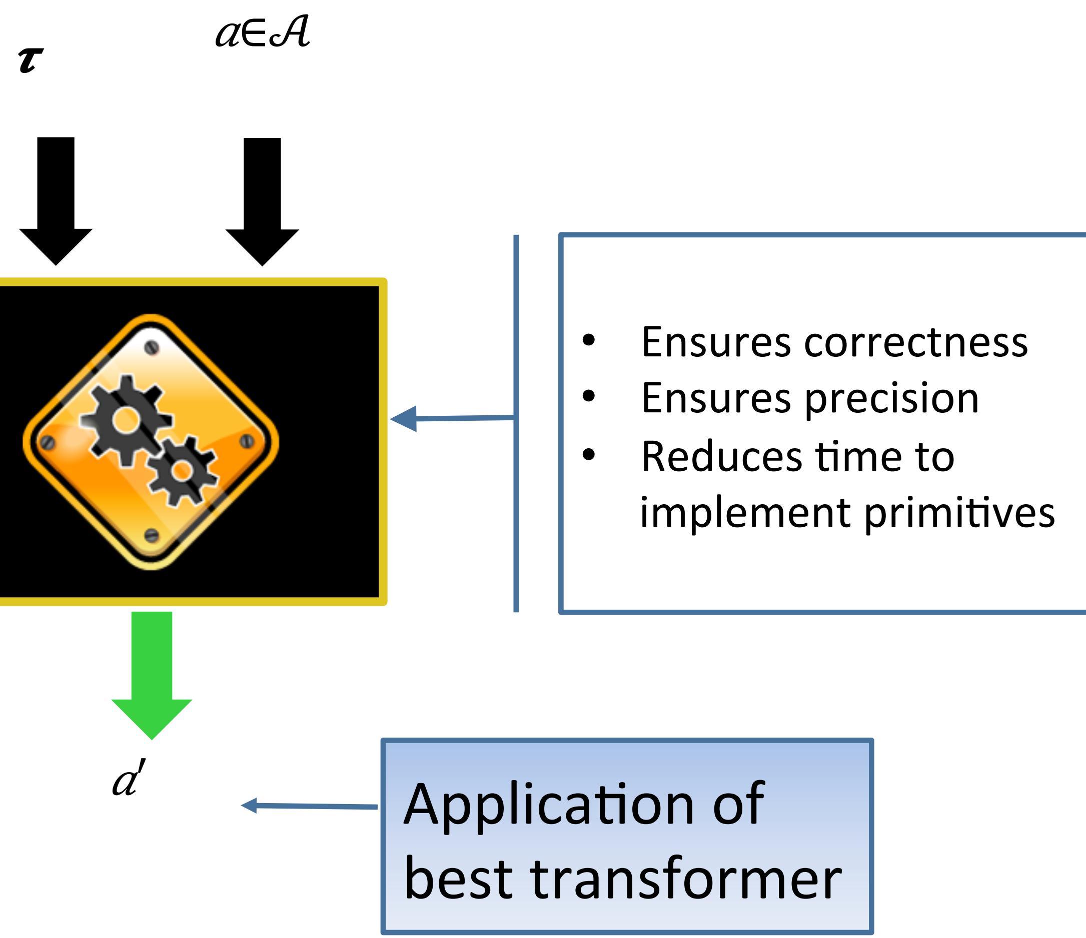
τ_1 : push ebx

$\tau_{1,2}^{\#}$

τ_2 : pop eax

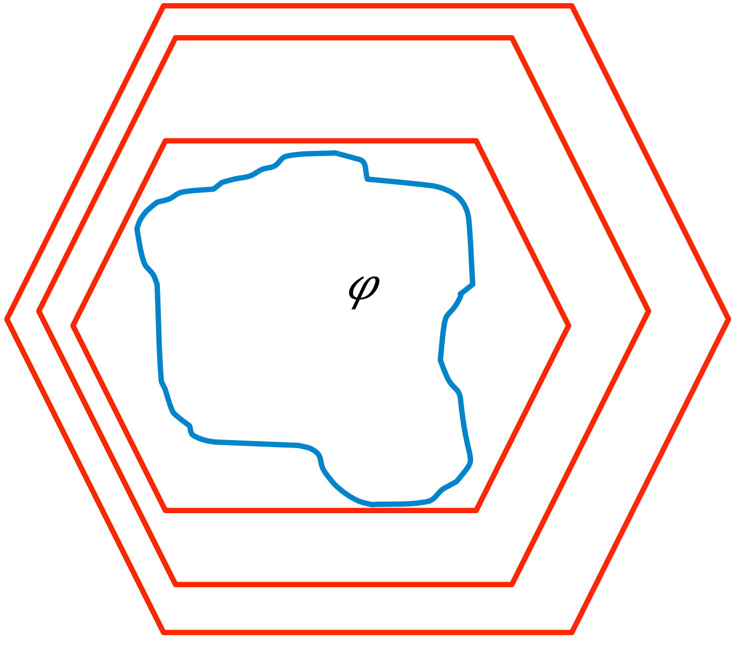
$ebx' = eax' \in \mathcal{A}$

Automation of best transformer



Symbolic Abstraction α

Given $\varphi \in \mathcal{L}$ and abstract domain $\mathcal{A}$, $\alpha(\varphi)$ is the *strongest consequence* of φ expressible in $\mathcal{A}$

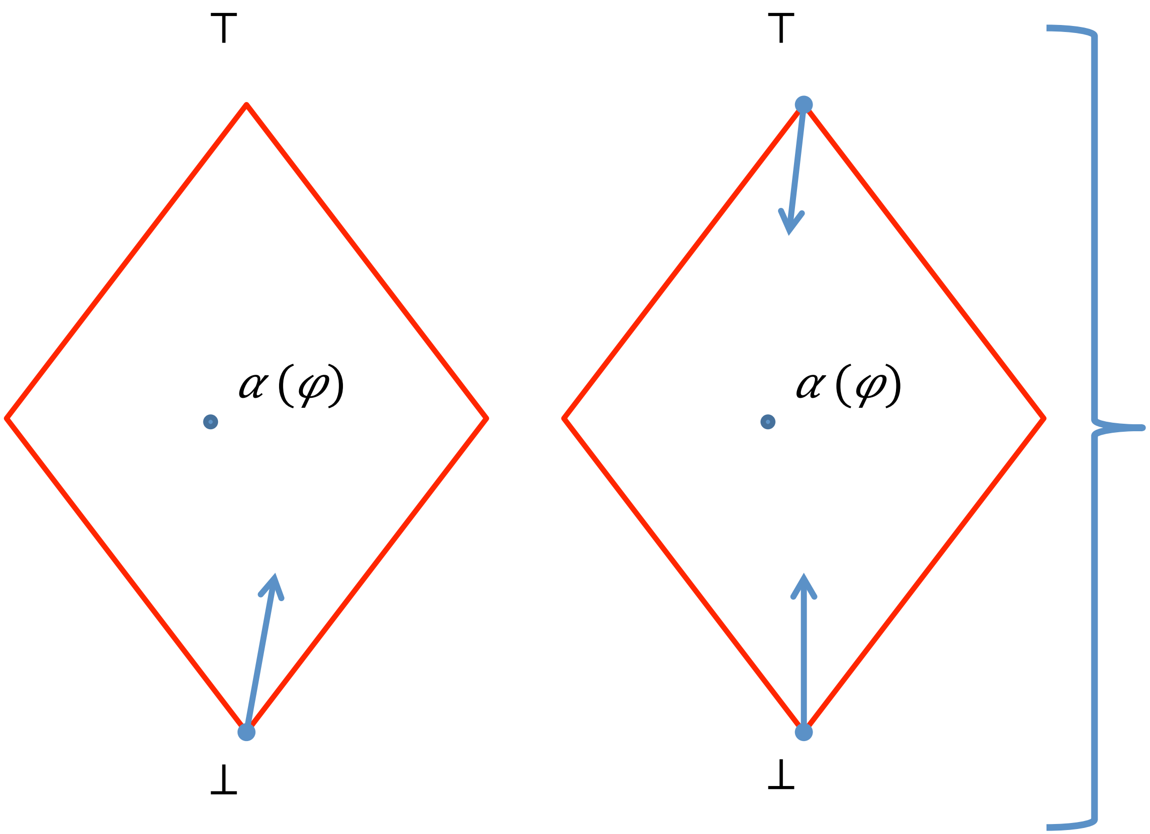


Symbolic Abstraction α

Given $\varphi \in \mathcal{L}$ and abstract domain $\mathcal{A}$, $\alpha(\varphi)$ is the *strongest consequence* of φ expressible in $\mathcal{A}$

$\alpha(a \wedge \varphi \downarrow \tau)$ gives the best abstract transformer

Frameworks for symbolic abstraction



- Inductive learning
- Abstract domain provides inductive bias
- Related to classical machine learning algorithms

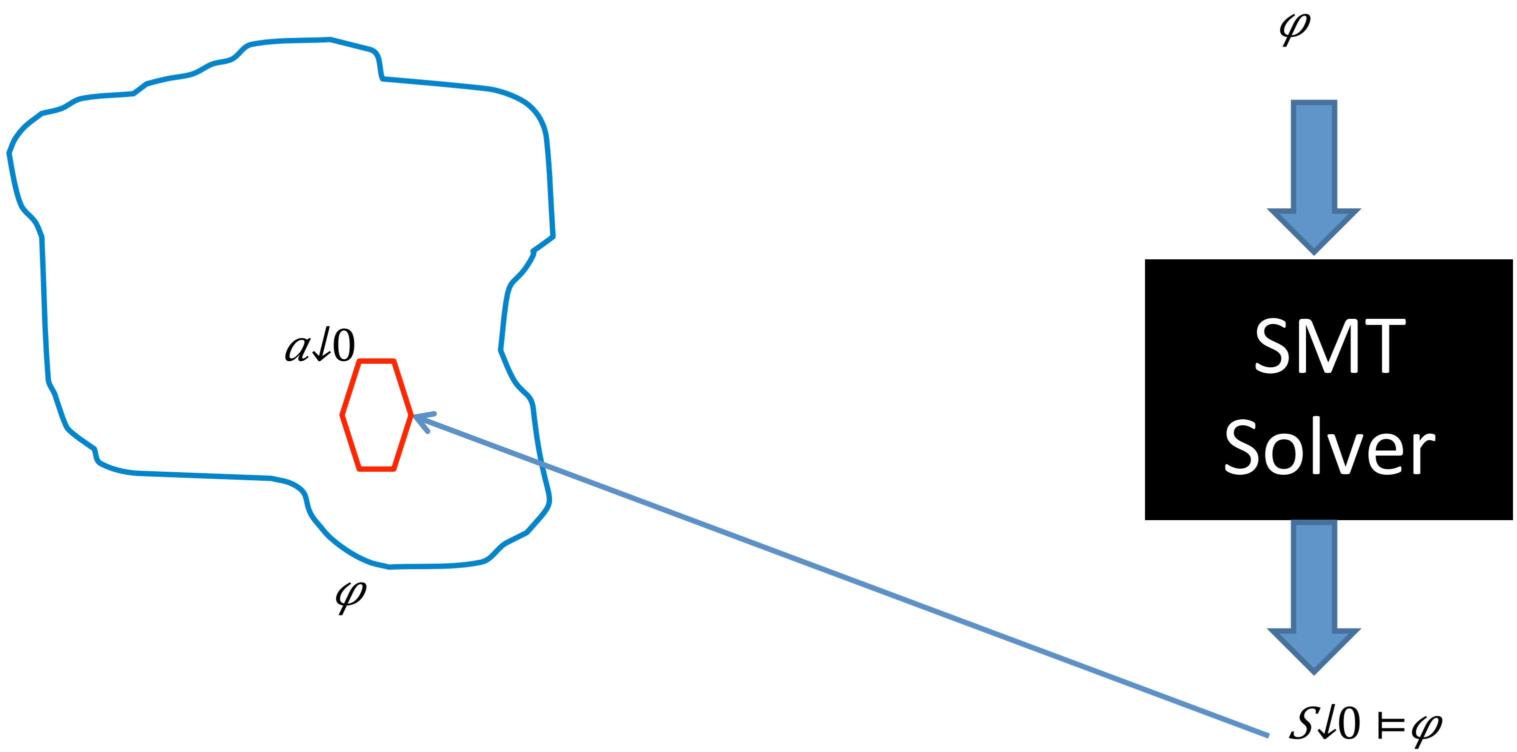
α-from-below
[VMCAI 2004]
$\approx$

Find-S algorithm

Bilateral α
[SAS 2012]
$\approx$

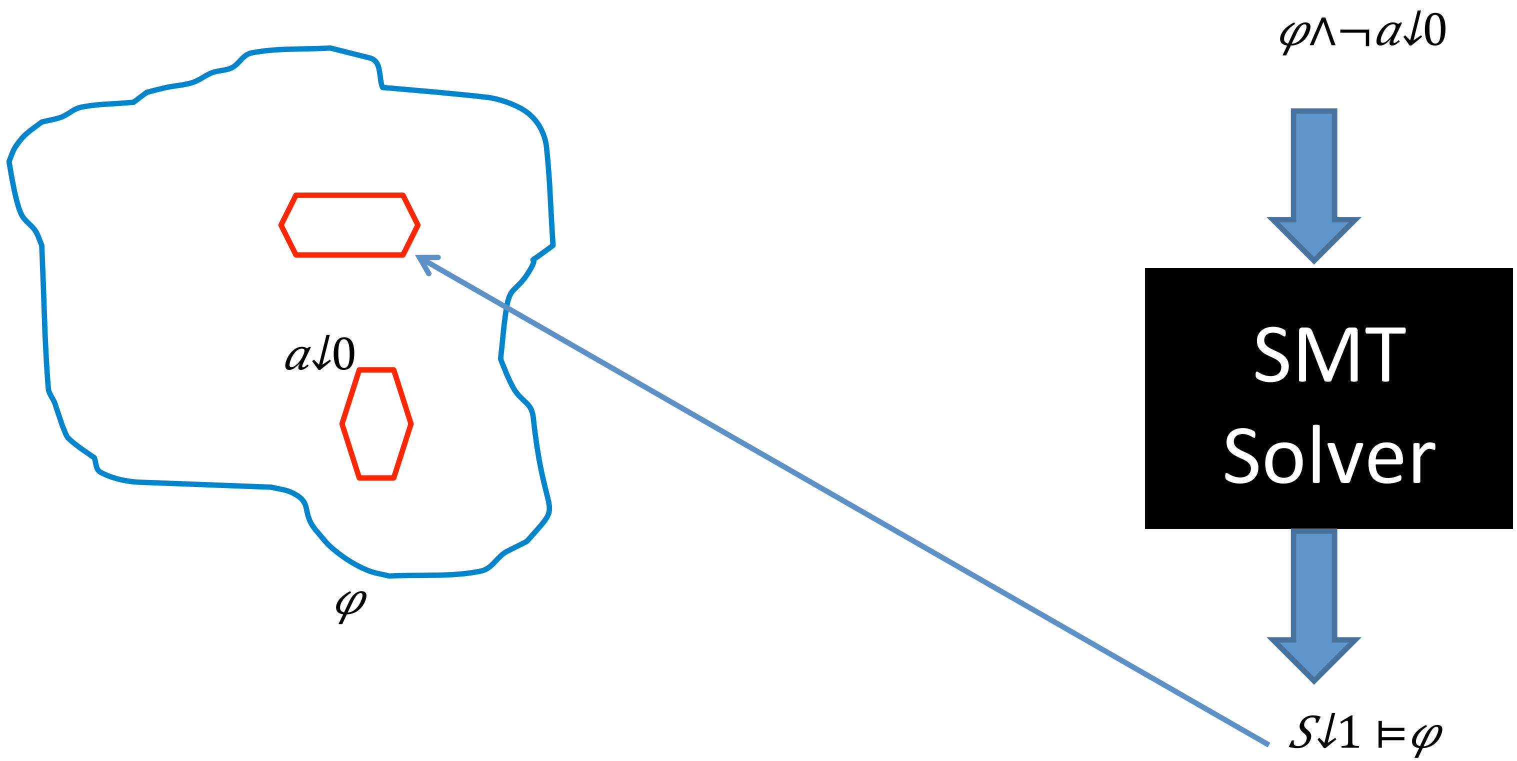
Candidate-elimination algorithm

α-from-below

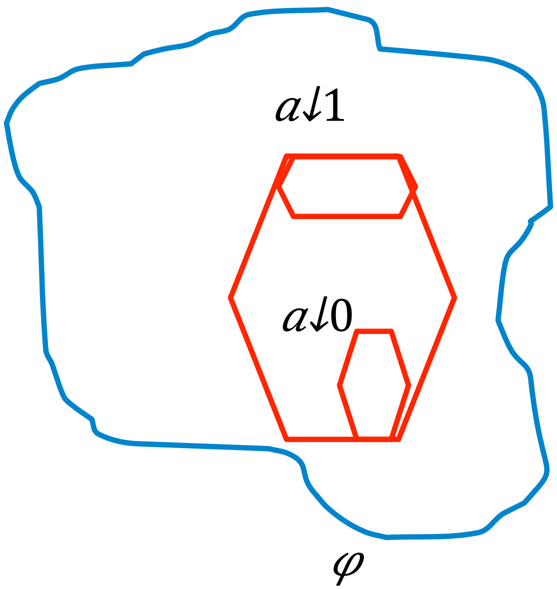


SMT: Satisfiability Modulo Theory

α-from-below

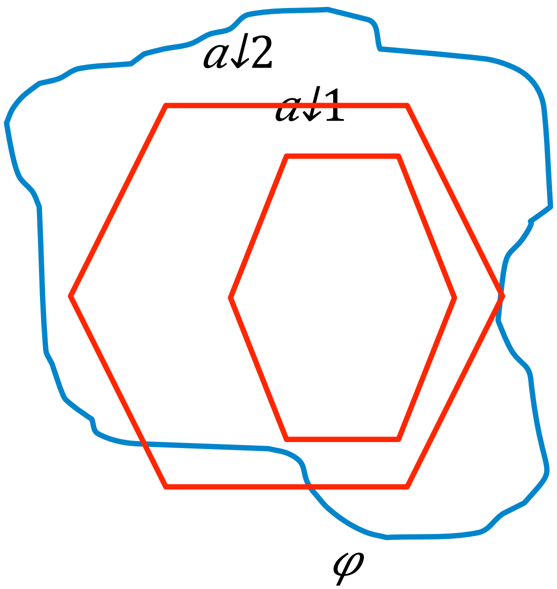


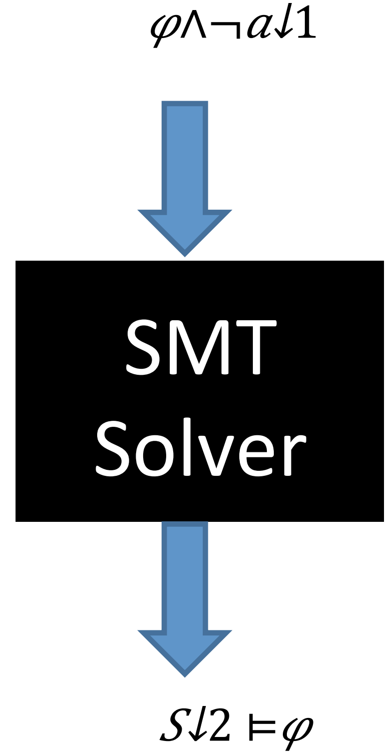
α-from-below



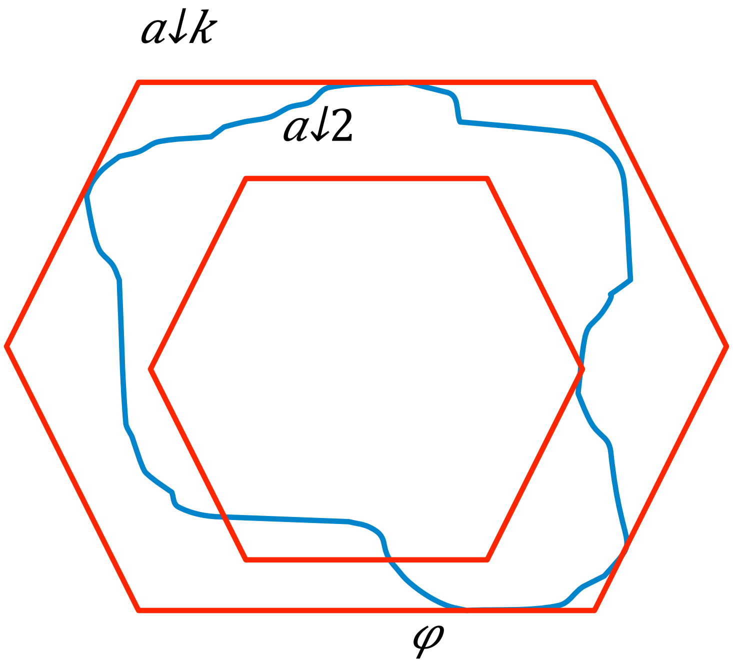
SMT Solver

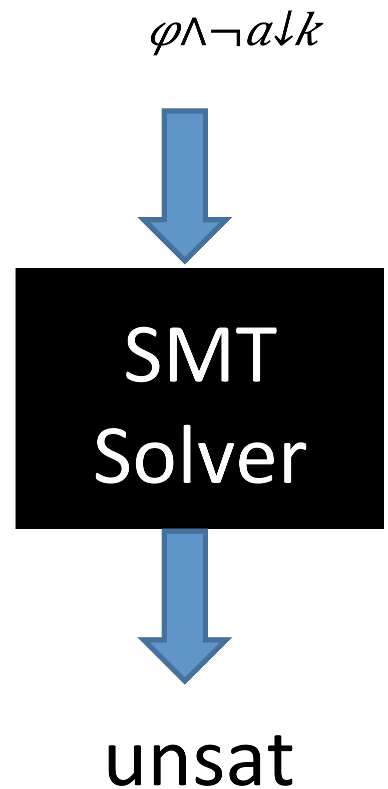
α-from-below



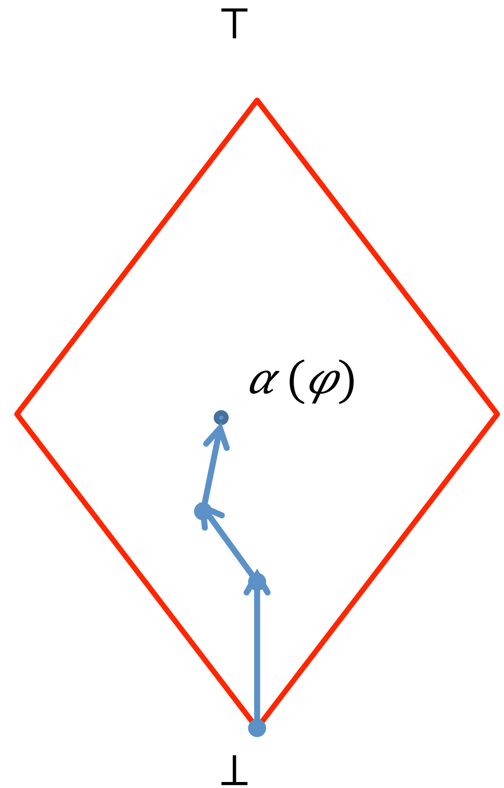


α-from-below

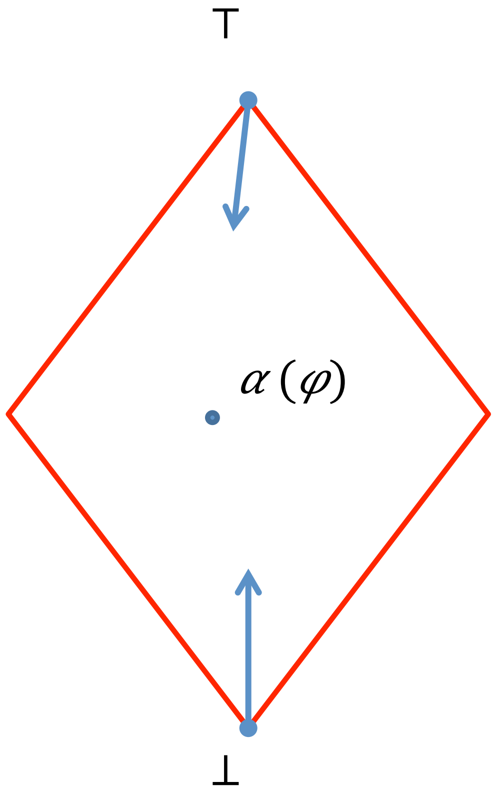




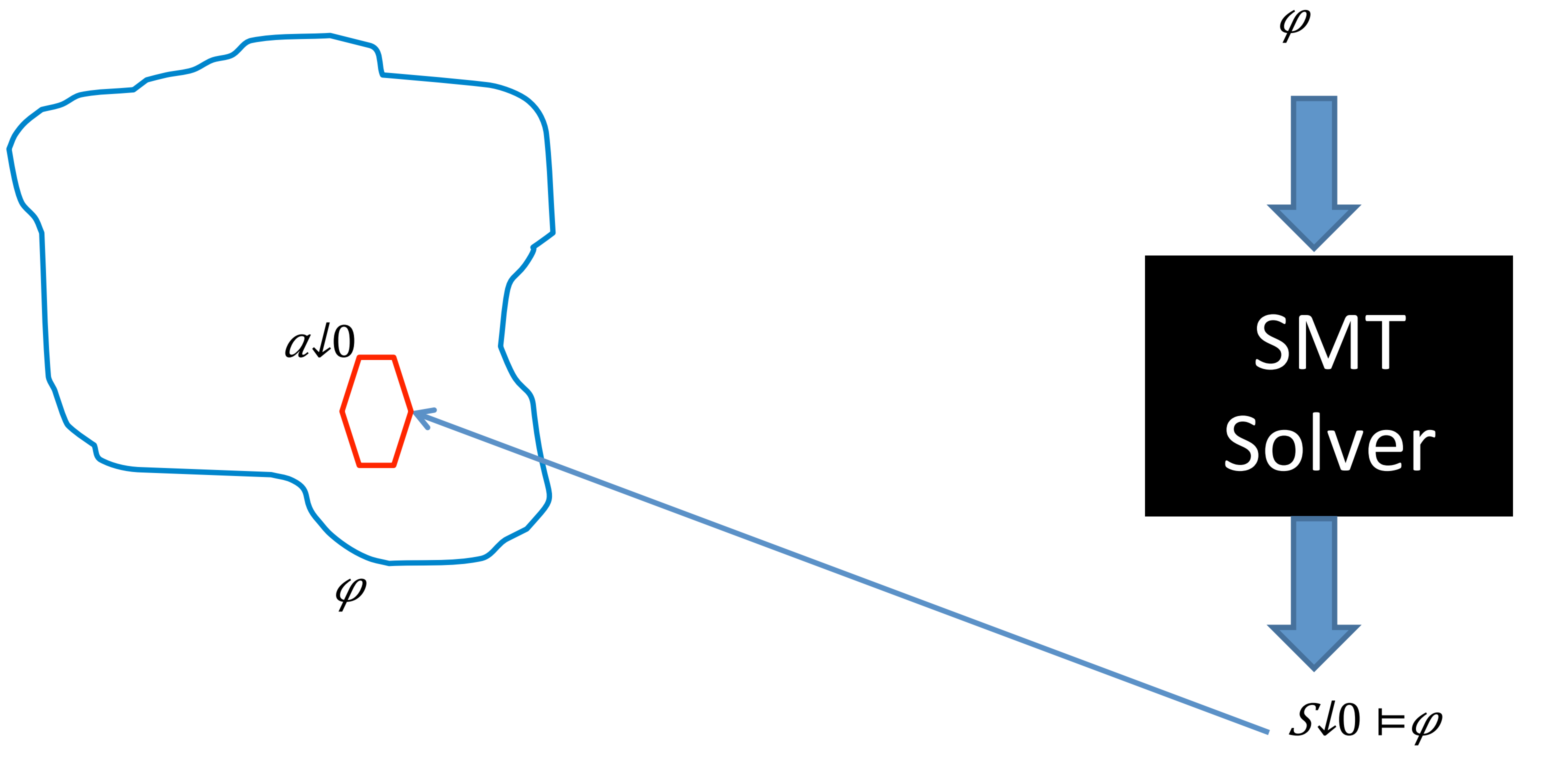
α-from-below



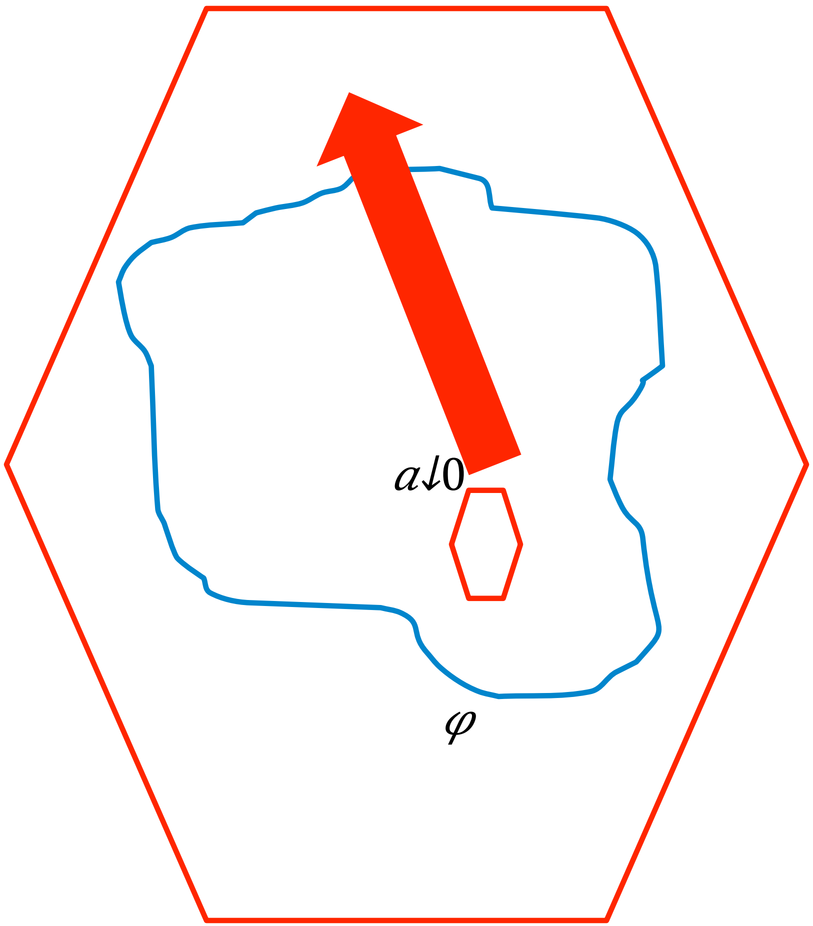
Bilateral α

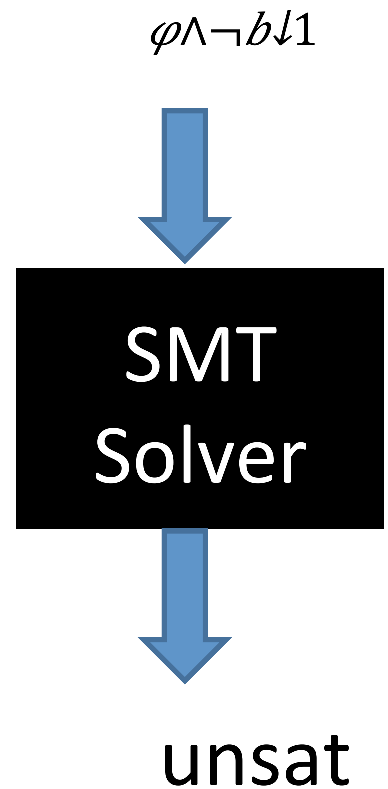


Bilateral α

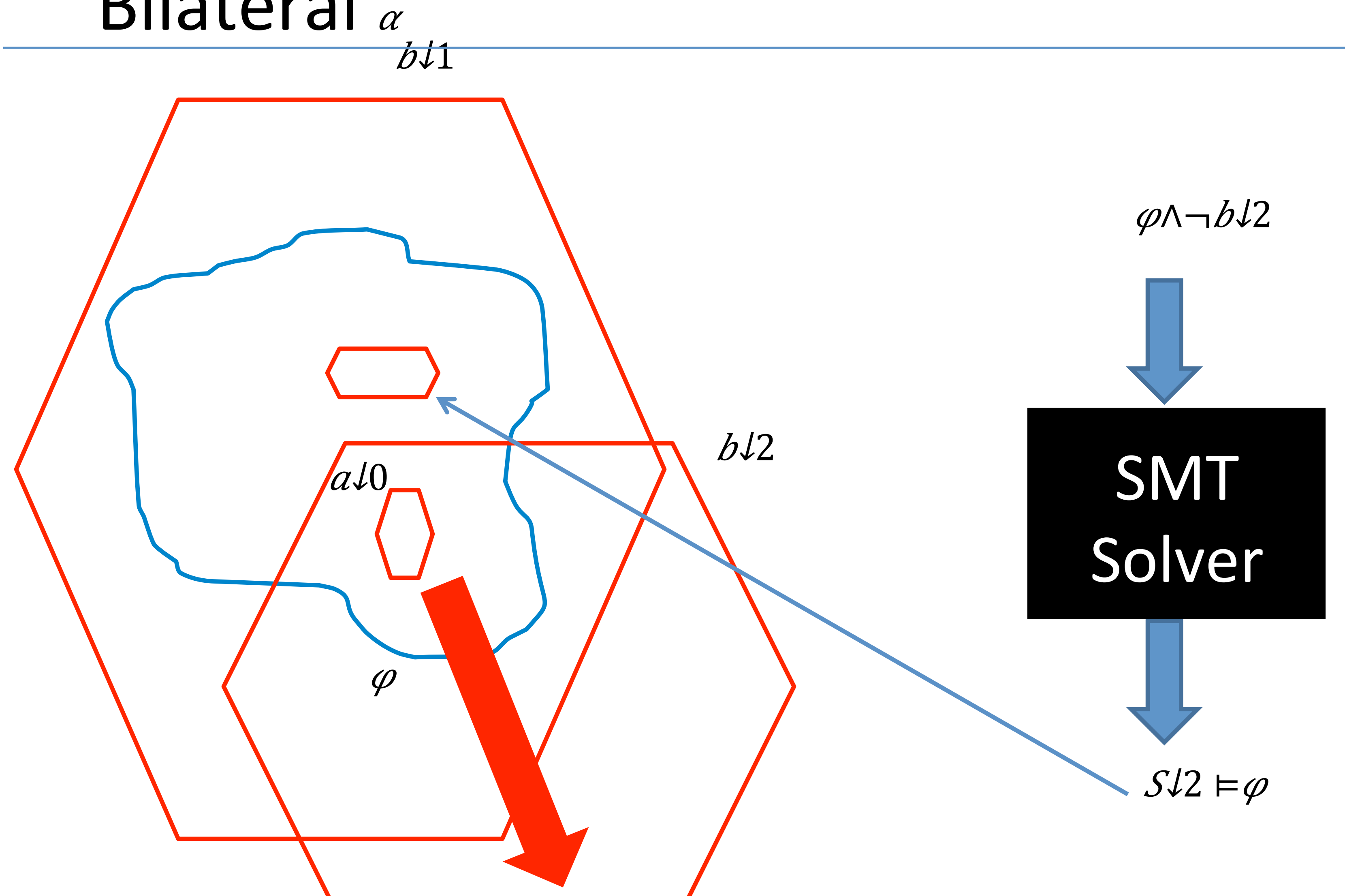


Bilateral α $b\downarrow 1$



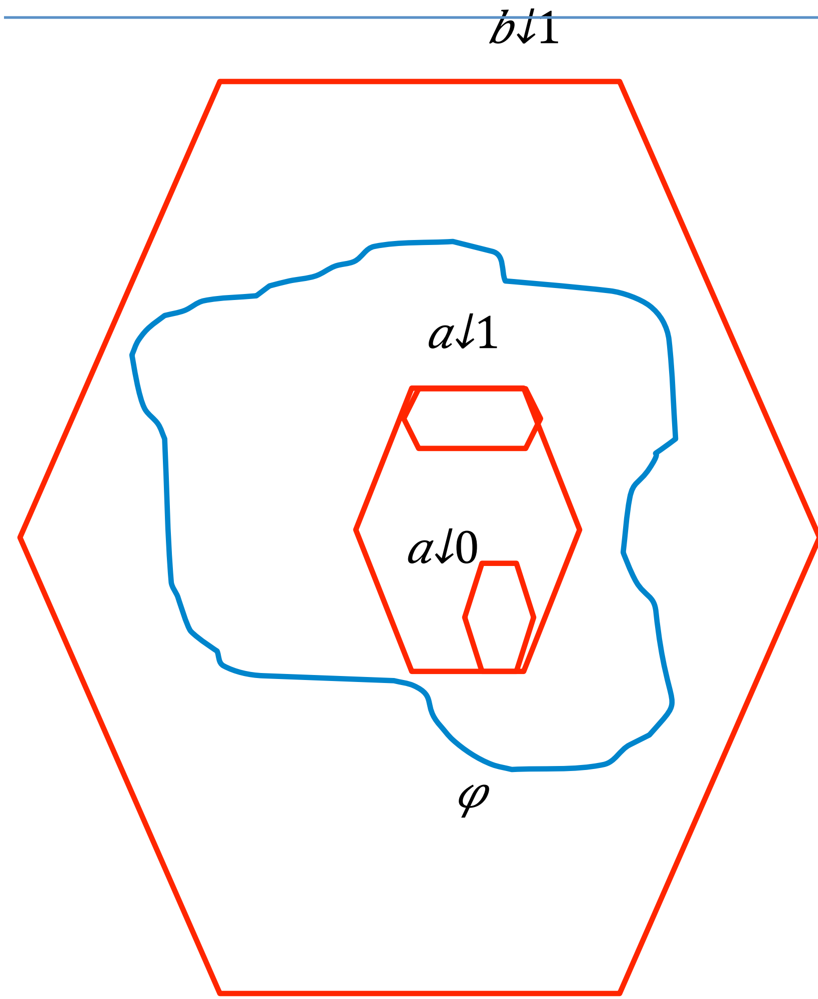


Bilateral



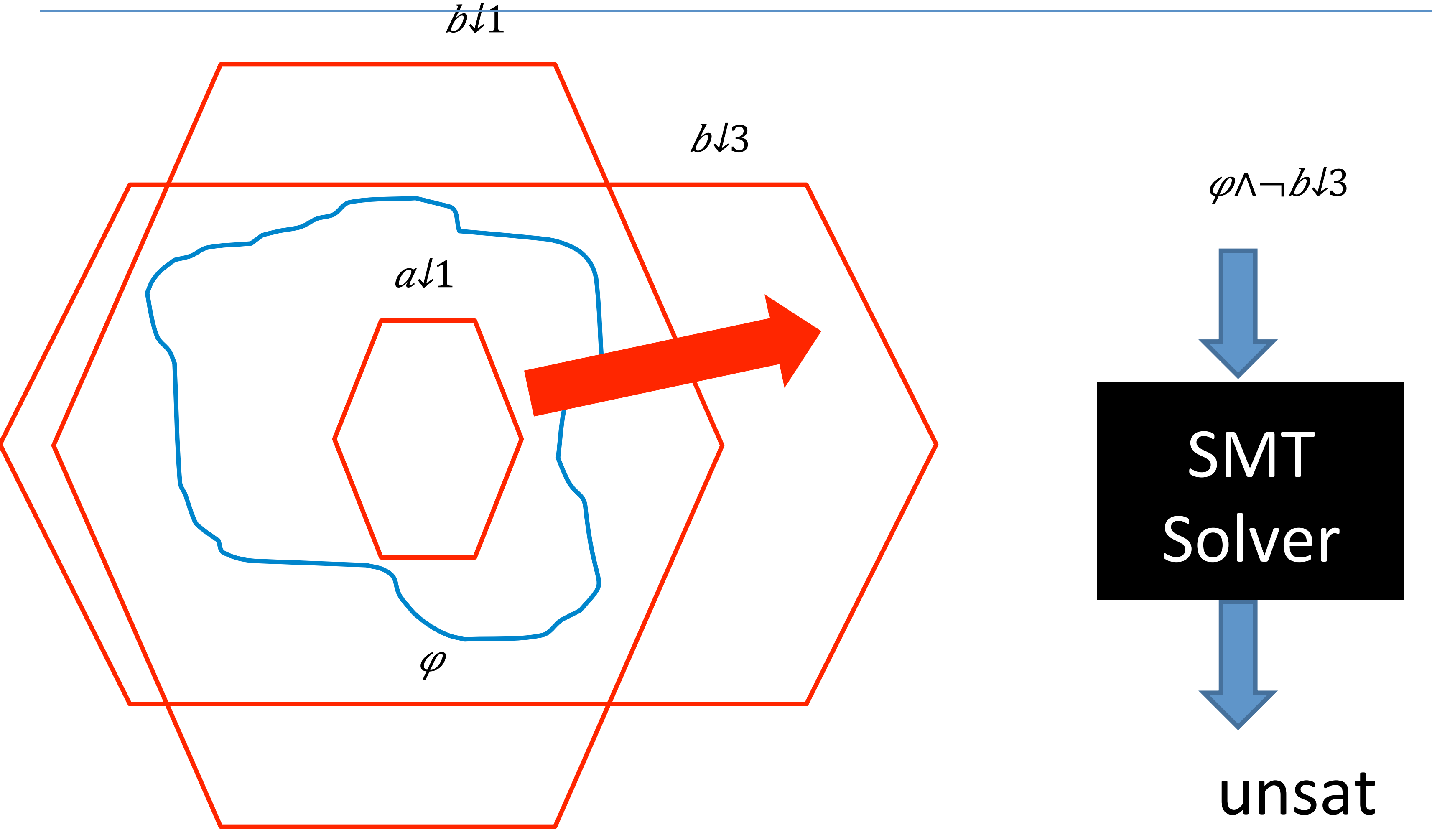
α
b_1
b_2
a_0
φ
$\varphi \wedge \neg b_2$
SMT Solver
$S_2 \models \varphi$

Bilateral α

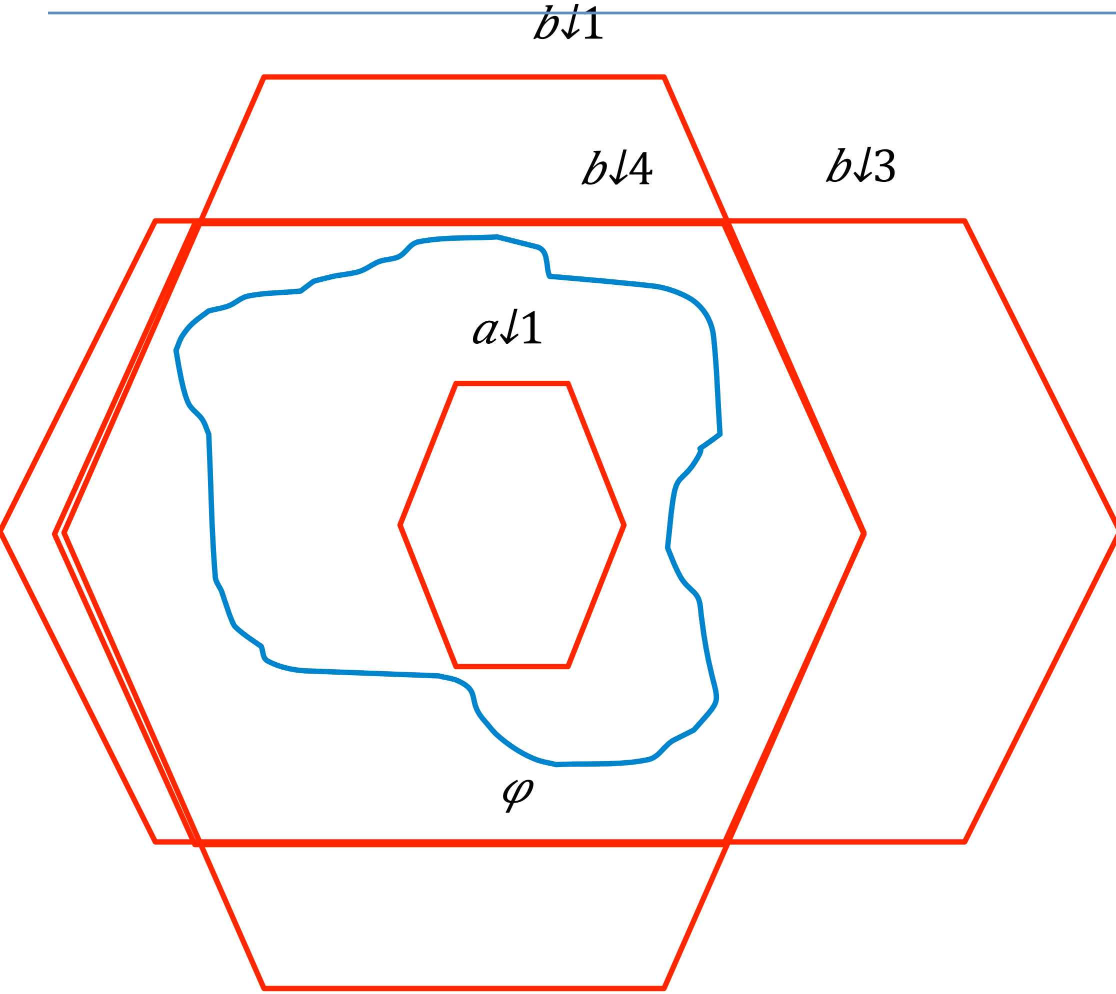


SMT Solver

Bilateral α

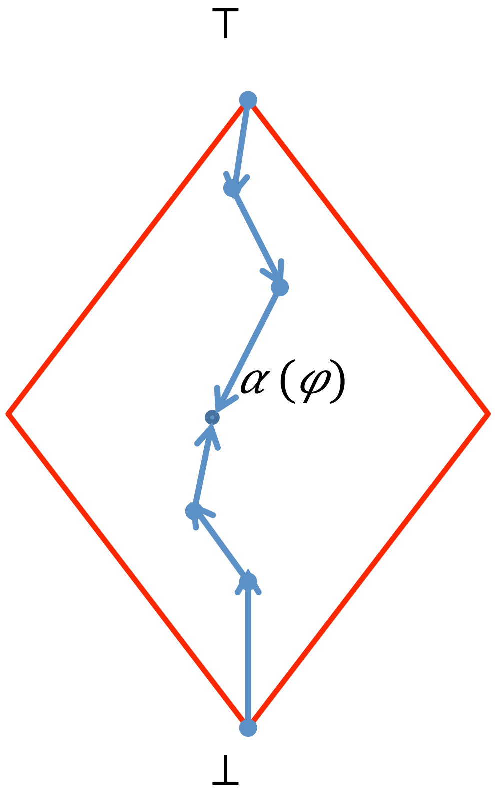


Bilateral α

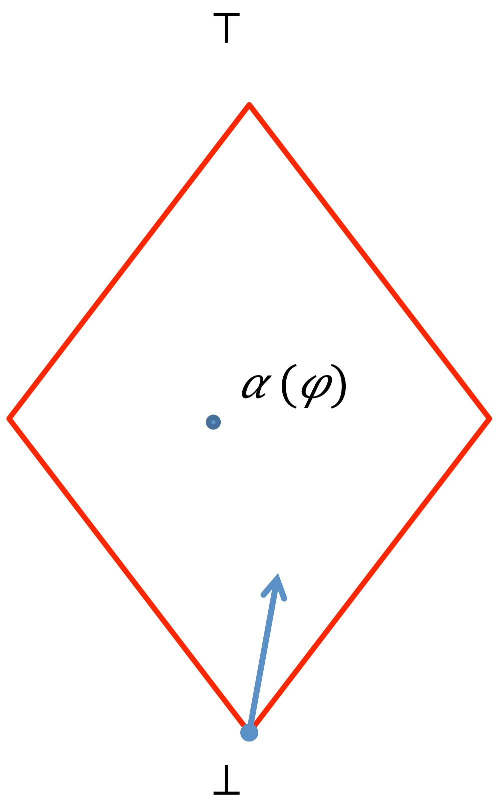


SMT Solver

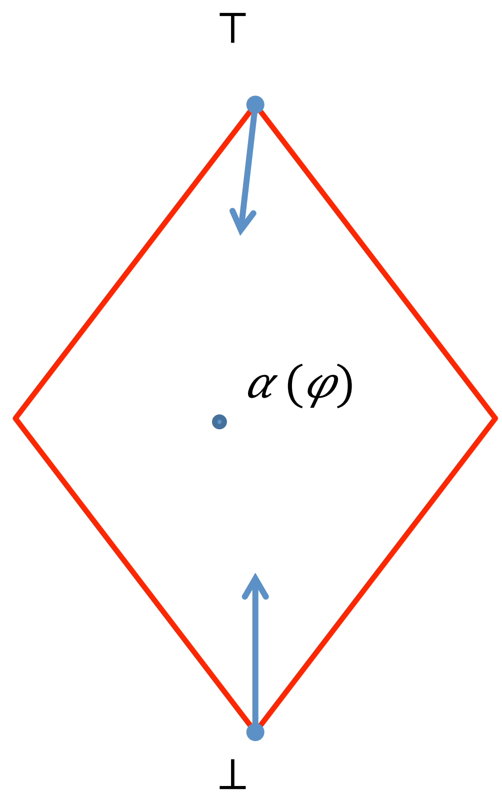
Bilateral α



Machine-code analysis

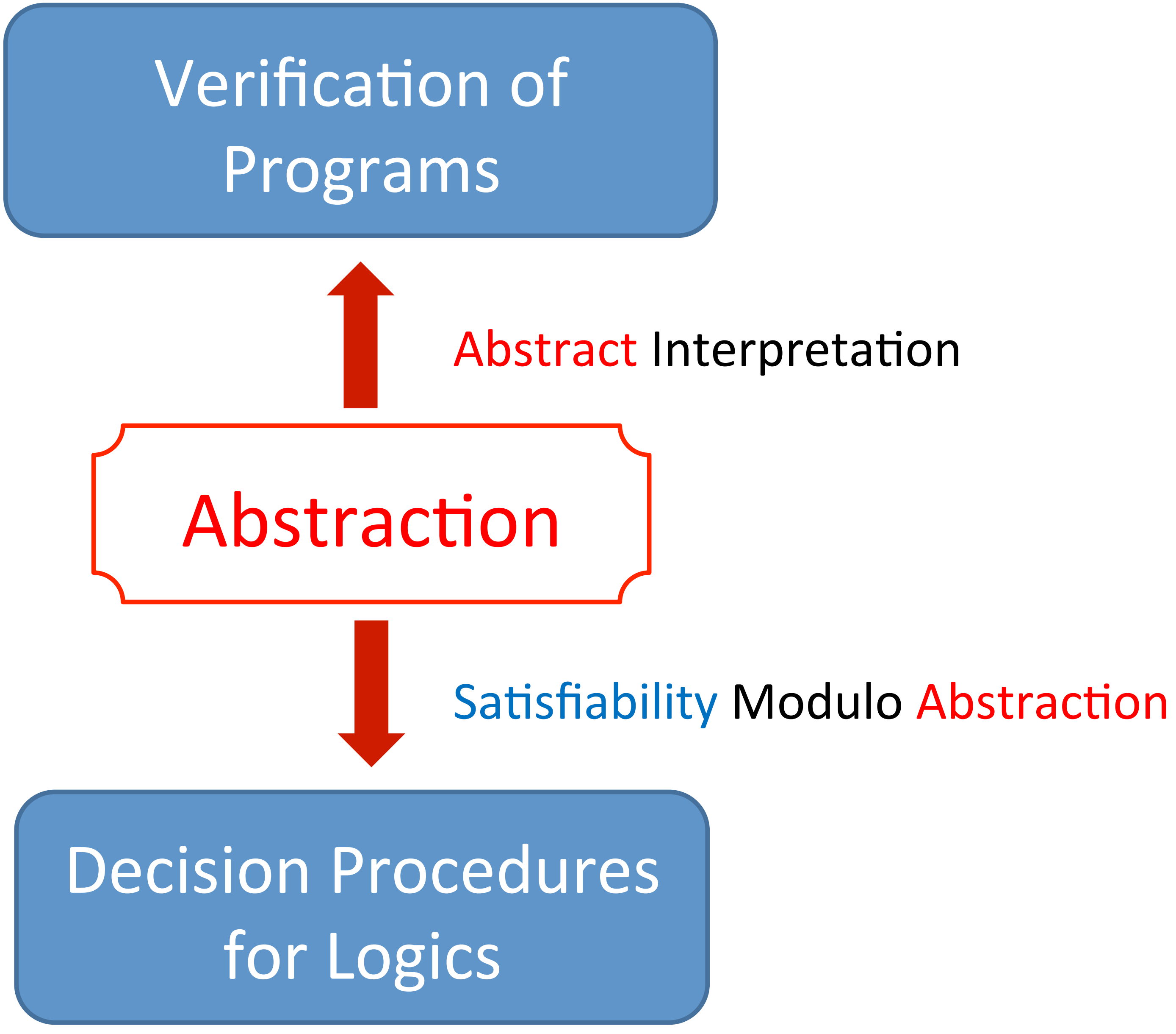


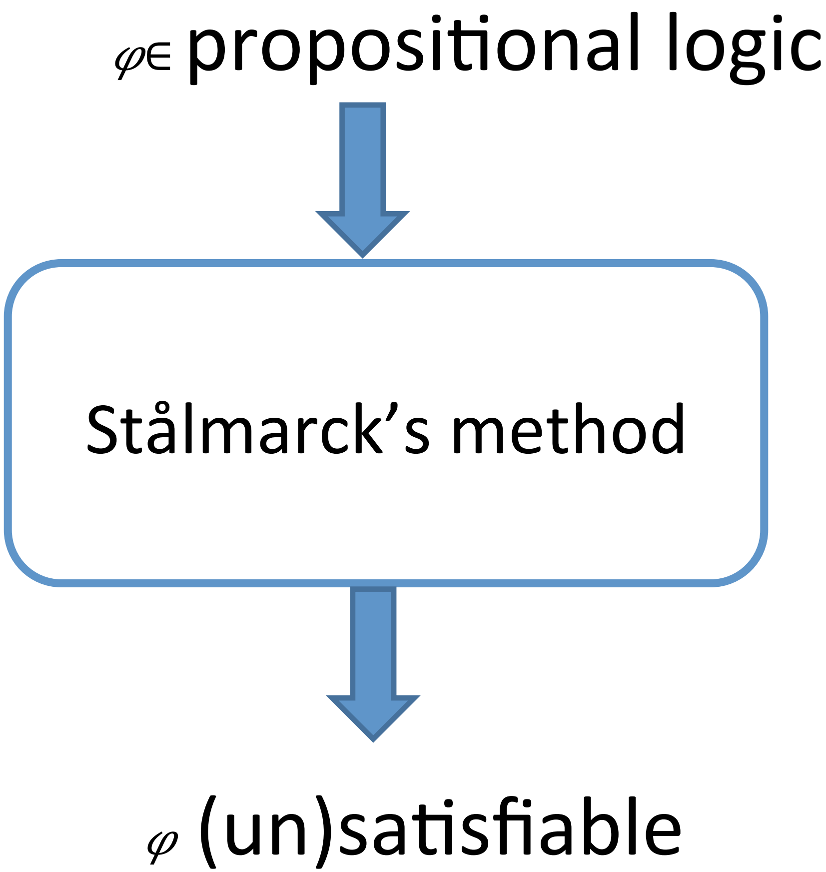
α-from-below
[VMCAI 2004]



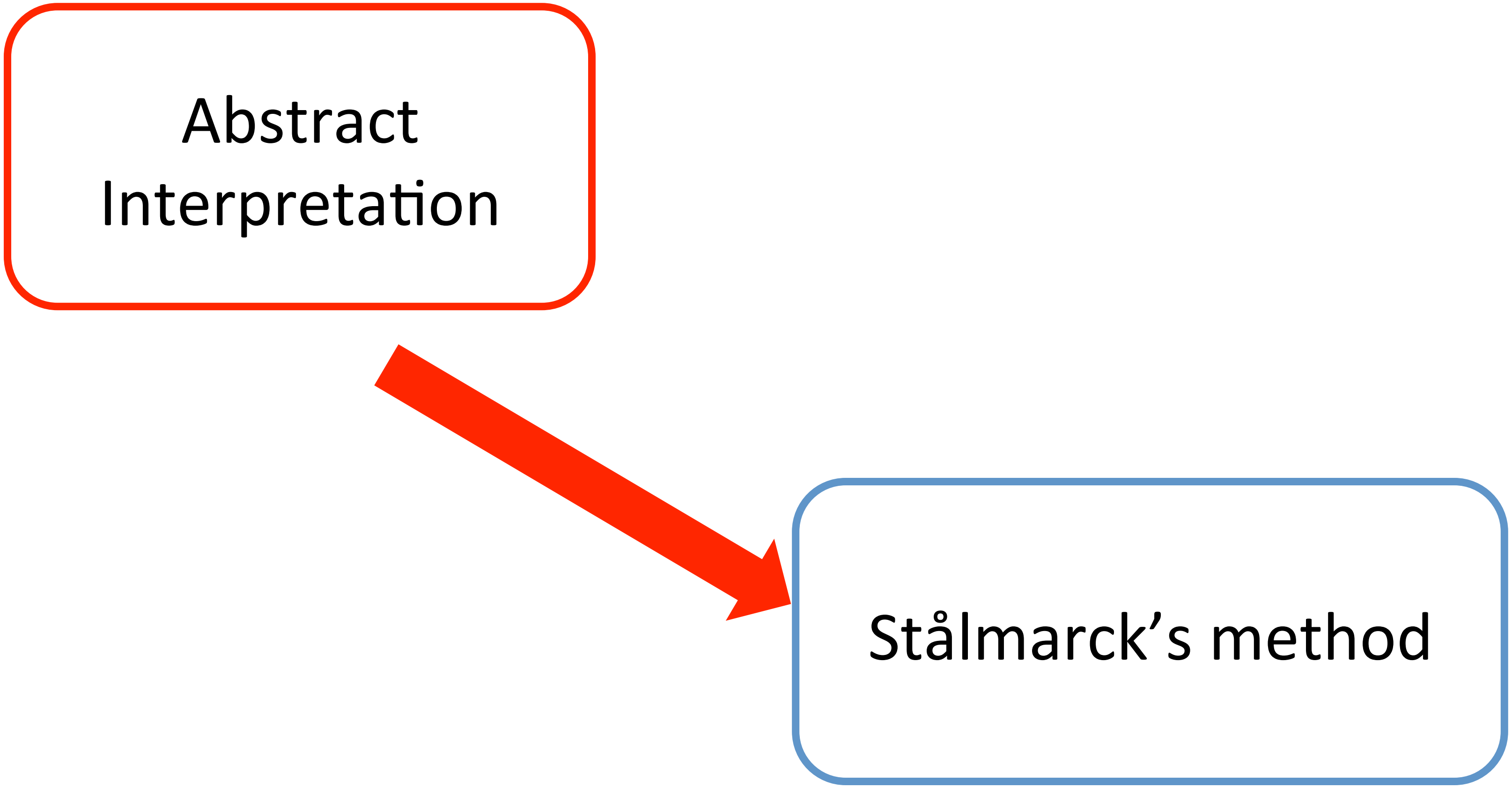
Bilateral α
[SAS 2012]

10x faster than
α-from-below


Verification of Programs
Abstract Interpretation
Abstraction
Satisfiability Modulo Abstraction
Decision Procedures for Logics


$\varphi \in$ propositional logic
Stålmarck's method
φ (un)satisfiable

Stålmarck's method


Abstract Interpretation
Stålmarck's method

Abstract Interpretation

Synthesize operations for abstract interpreters

Stålmarck's method

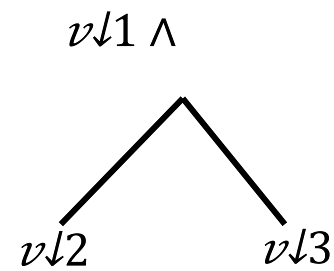
- New SAT algorithms
- Generalize to SMT
- Computes $\alpha(\varphi)$

Satisfiability Modulo Abstraction (SMA) solver

Stålmarck's method

Propagation Rules

$$R=\{v_1 = True, v_2 = *, v_3 = *\}$$

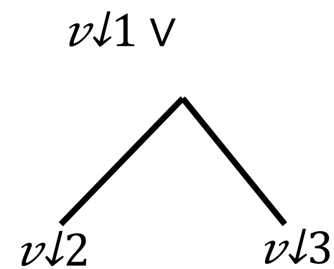


$$R' = R \cup \{v_1 = *, v_2 = True, v_3 = True\}$$

Stålmarck's method

Propagation Rules

$$R = \{v_1 = False, v_2 = *, v_3 = *\}$$

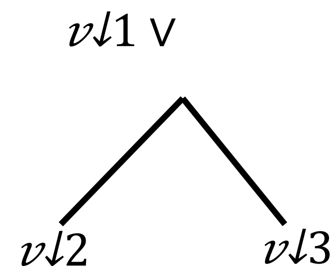


$$R' = R \cup \{v_2 = False, v_3 = False\}$$

Stålmarck's method

Propagation Rules

$$R = \{v_1 = True, v_2 = *, v_3 = *\}$$

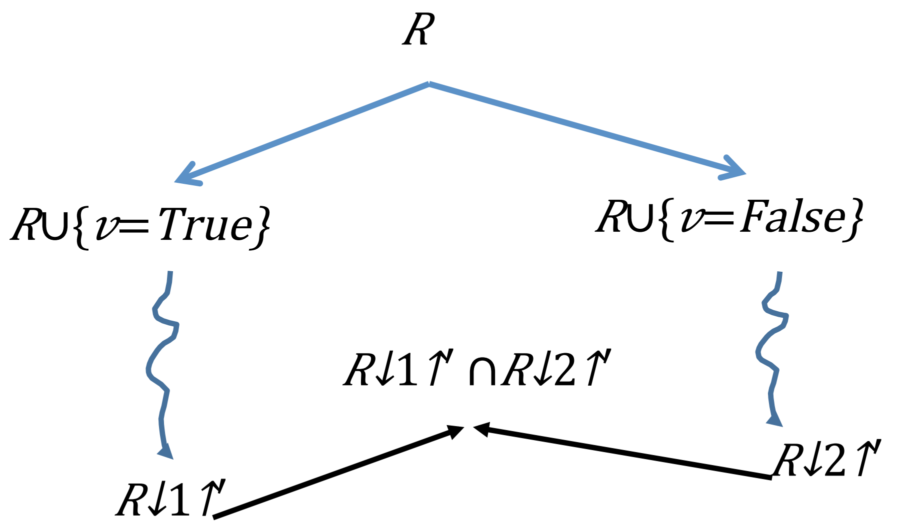


$$R' = R$$

Stålmarck's method

Dilemma Rule

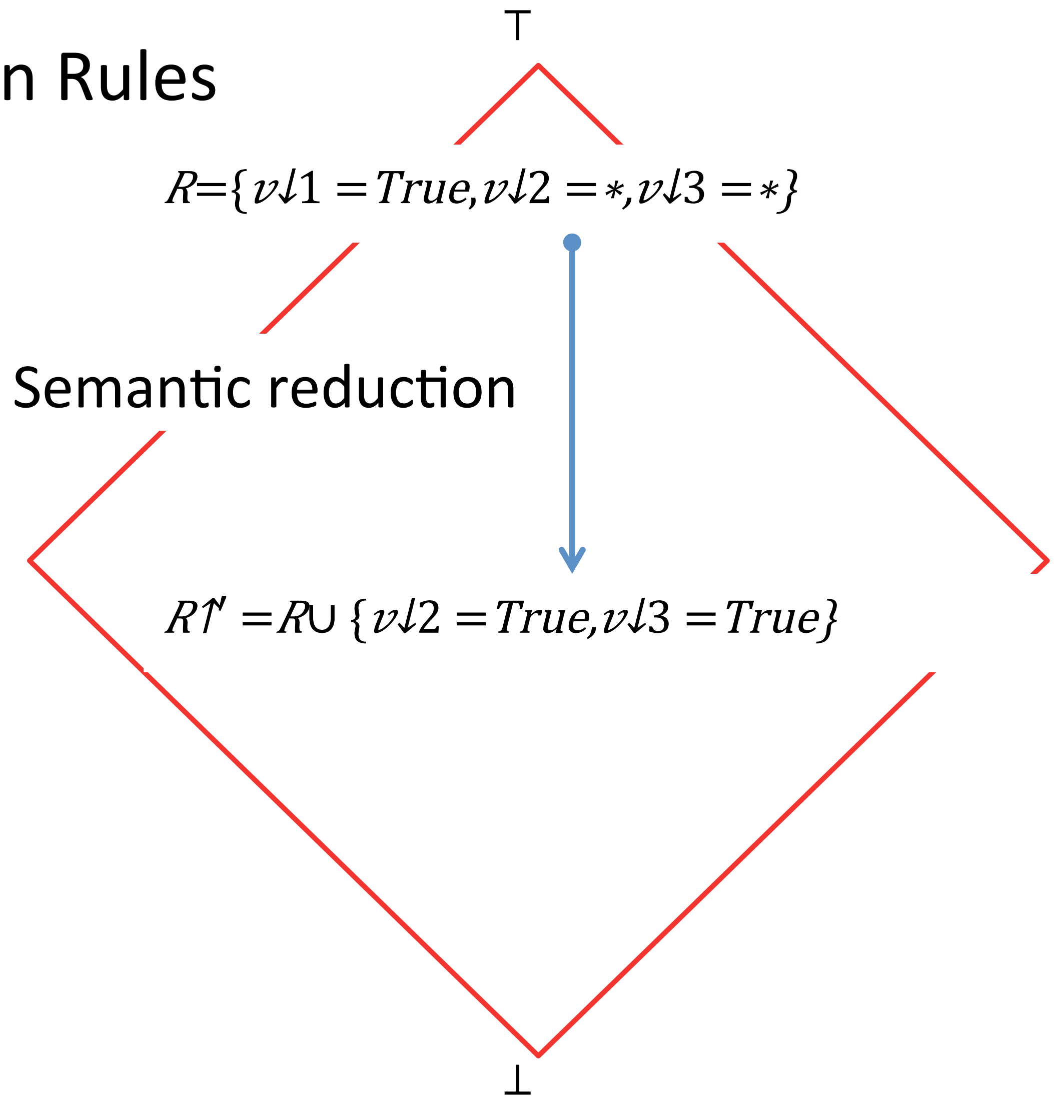
- Split
- Propagate
- Merge

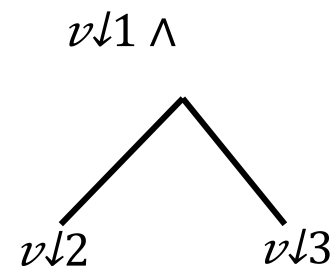


Stålmarck's method

Generalized Stålmarck's method

Propagation Rules



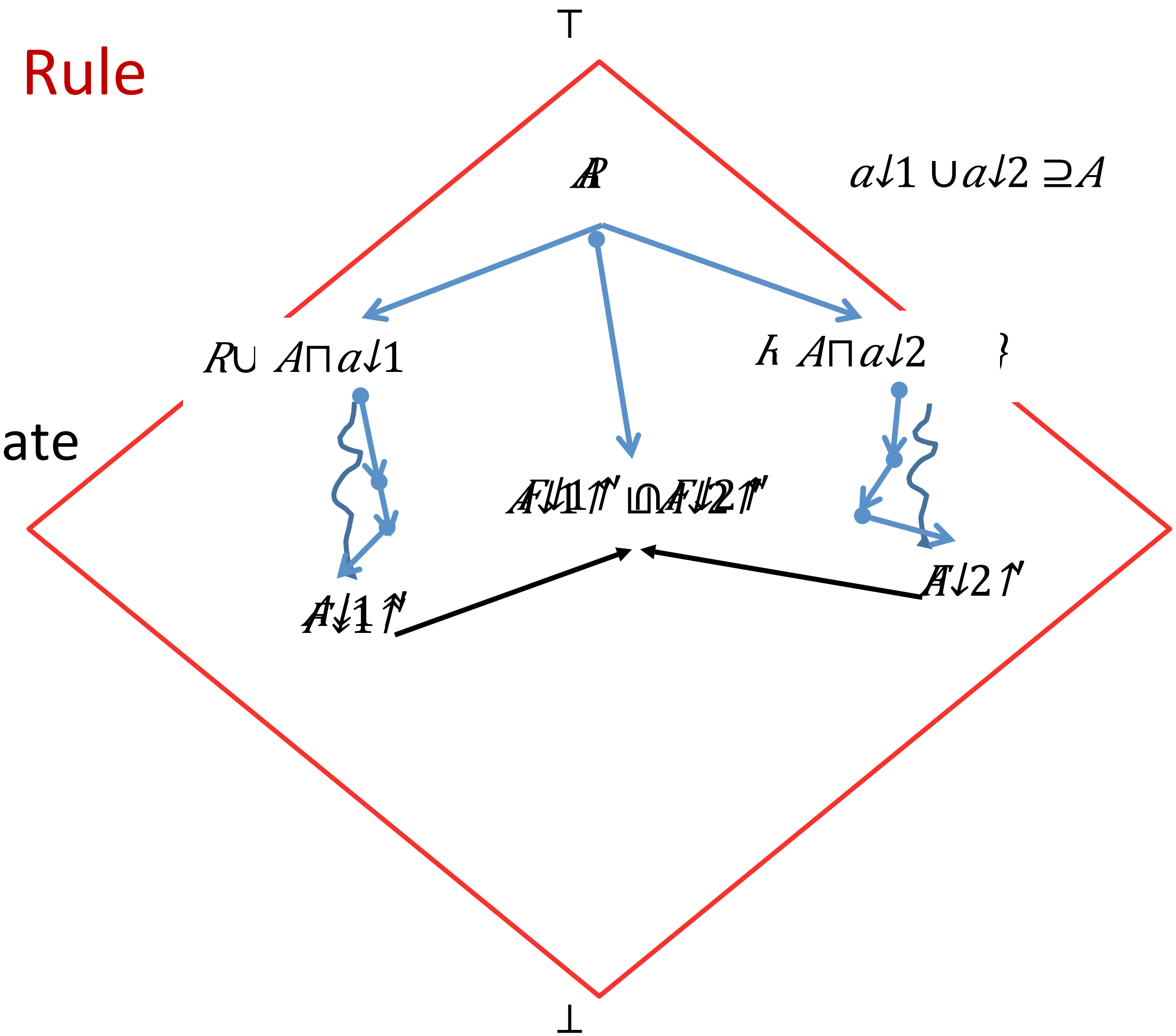


Generalized Stålmarck's method

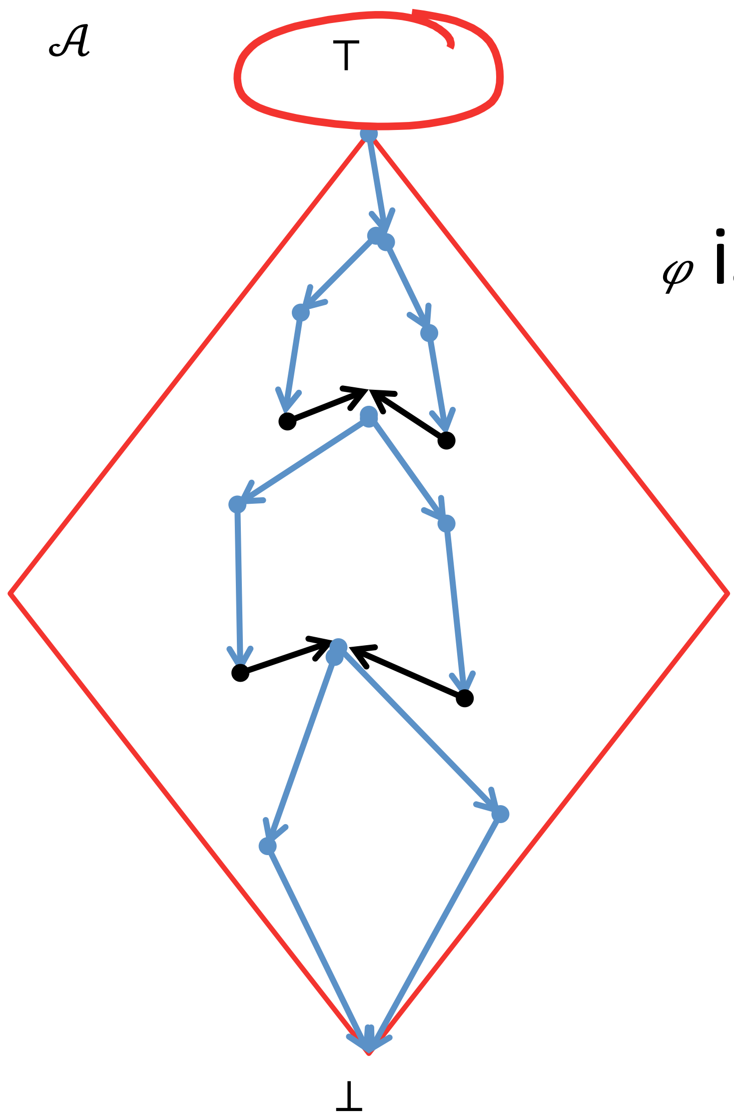
Stålmarck's method

Dilemma Rule

- Split
- Propagate
- Merge



Generalized Stålmarck's method



φ is unsatisfiable

Generalized Stålmarck's method

propositional logic

Stålmarck$_{[Implication]}(\varphi)$ $x \rightarrow y$

Stålmarck$_{[Equivalence]}(\varphi)$ $x \leftrightarrow y$

Stålmarck$_{[Cartesian]}(\varphi)$ $x=1, y=0$

[SAS 2012]

Generalized Stålmarck's method

richer logic

Stålmarck$_{[\text{richer domain}]}(\varphi)$

Generalized Stålmarck's method

QF_LRA logic

Stålmarck$_{[\text{Boolean, Polyhedral}]}(\varphi)$

QF_BV quantifier-free linear
rational arithmetic

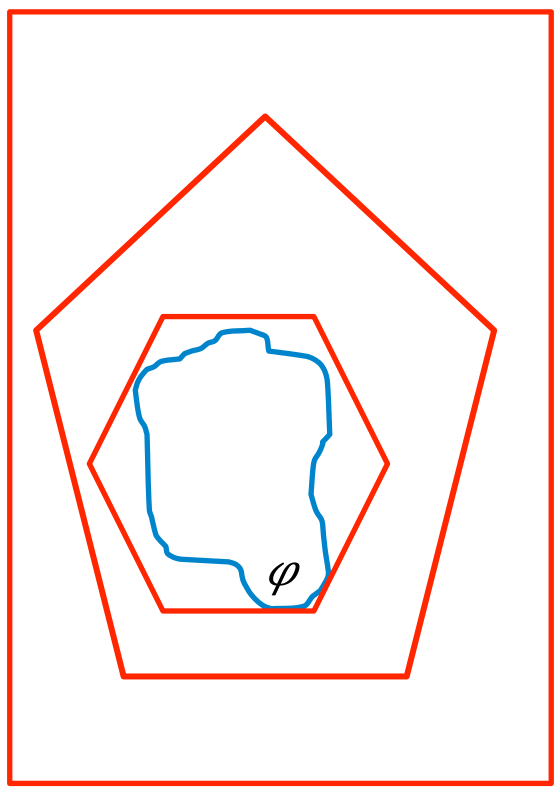
Generalized Stålmarck's method

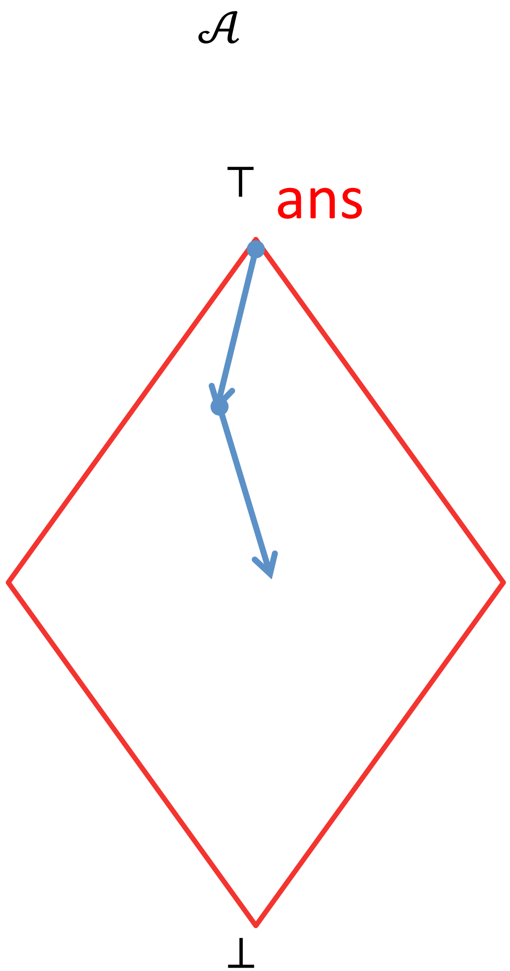
QF_BV logic

Stålmarck$[\text{Boolean, Bitvector domain}](\varphi)$

QF_BV quantifier-free bitvector logic

Generalized Stålmarck's method computes $\alpha(\varphi)$





Symbolic Abstraction $\widehat{\alpha}$

$\widehat{\alpha}(\varphi) = \bot$ implies φ is unsatisfiable

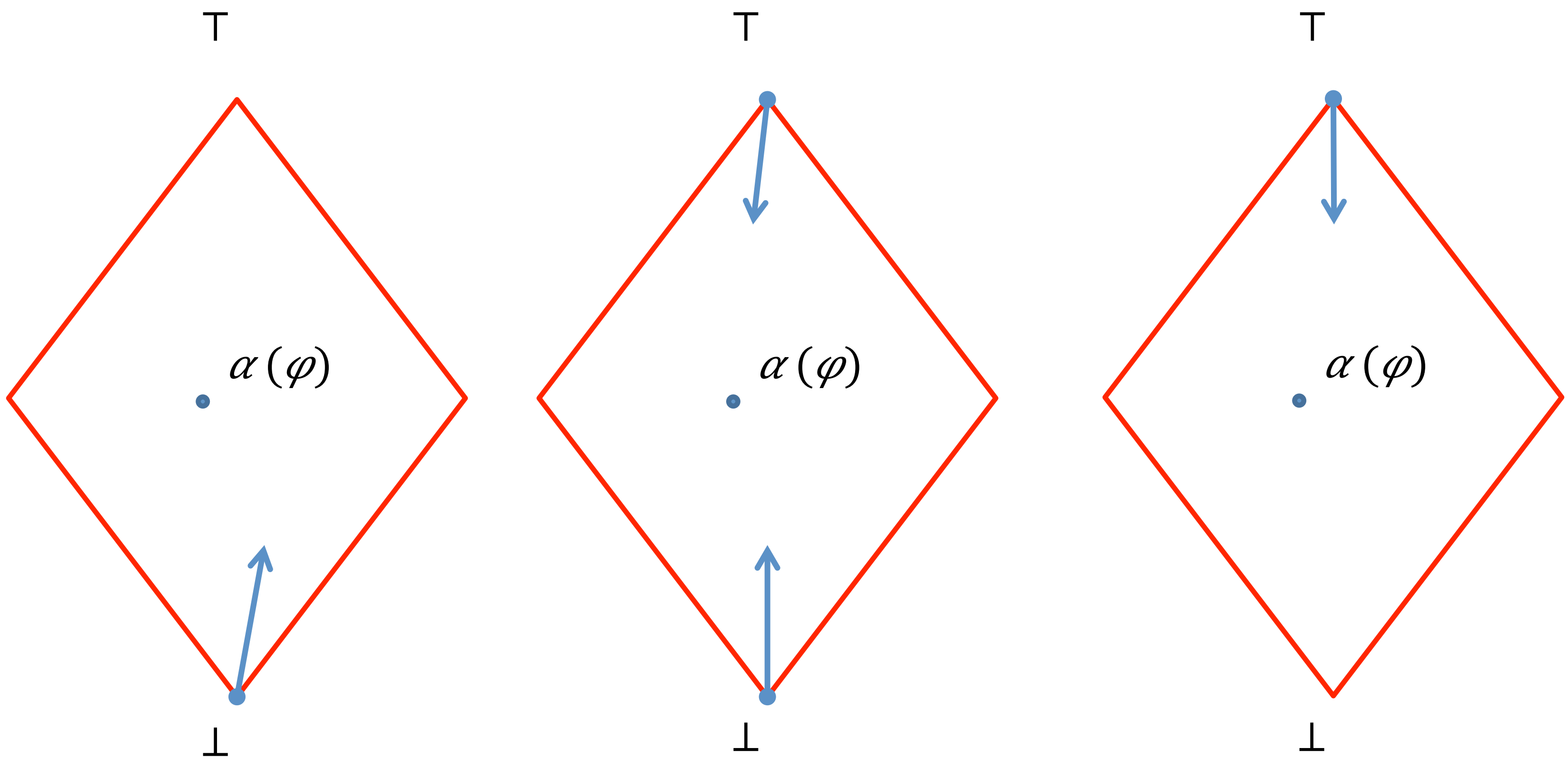
$\mathcal{L}$

Dual-use

$\widehat{\alpha}(a \wedge \varphi \downarrow \tau)$ gives the best abstract transformer

$\mathcal{A}$

Machine-code analysis



α-from-below
[VMCAI 2004]

Bilateral α
[SAS 2012]

α-from-above
[CAV 2012]

10x faster than α-from-below

11x faster than Bilateral α

SMA solver for Separation Logic (SL)

Separation Logic (SL)

- Expressive logic for describing heap properties

ls(src)

```
List * list_copy ( List * src )
{
  …
  …
  return cpy;
}
```

ls(src) * ls(cpy)

"I have a list pointed to by src, and another pointed to by cpy, occupying separate storage, and nothing else."

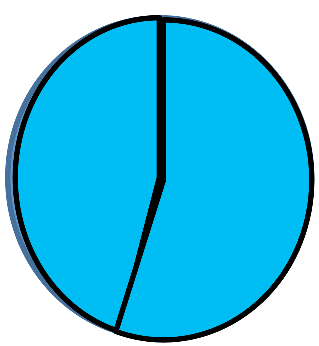
Separating conjunction $*$

heap $h \models \varphi_1 * \varphi_2$ **iff**

there exists a partition h_1 and h_2 of h

such that $h_1 \models \varphi_1$ and $\mathrm{h}_1 \models \varphi_2$

Implicit second-order quantification



$\models \varphi_1 * \varphi_2$

$h_1 \models \varphi_1$ $h_2 \models \varphi_2$

Local Reasoning

$$\{\varphi_1\}\ P\ \{\varphi_2\} / \{\varphi_1 * \psi\}\ P\ \{\varphi_2 * \psi\}$$

Frame rule

$$\{ls(a)\}\ \text{copy\_list}\ \{ls(a) * ls(b)\} / \{ls(a) * ls(c)\}\ \text{copy\_list}\ \{ls(a) * ls(b) * ls(c)\}$$

Frame rule

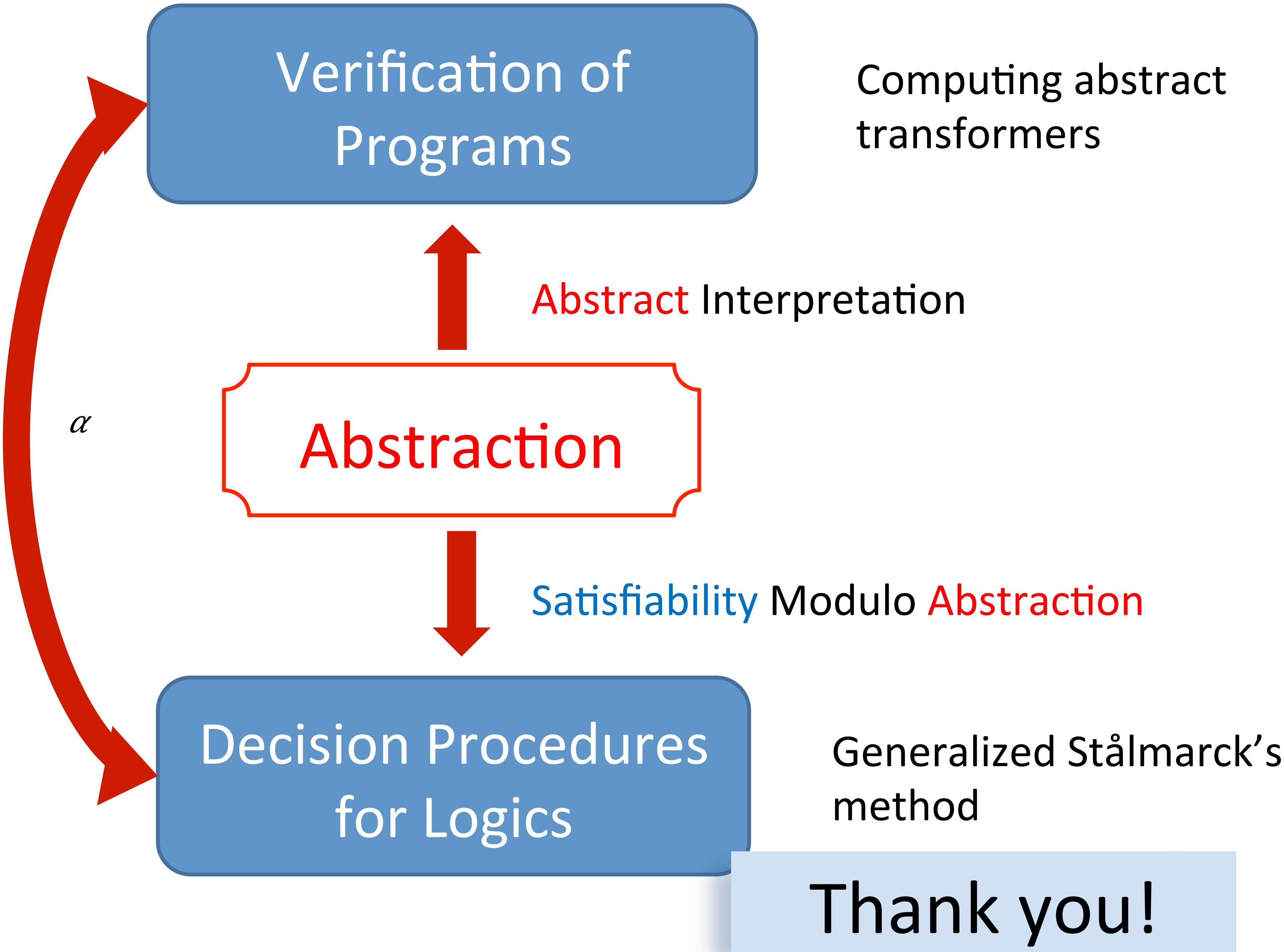
Decision procedures for SL

- SL undecidable, in general
- Current tools work with (limited) decidable fragments of SL
- Cannot handle $(\varphi_1 * \varphi_2) \wedge (\psi_1 * \psi_2)$
  - Useful for describing overlaid data structures
- Cannot handle separating implication (magic wand)
  - Required for weakest-precondition reasoning

SMA solver for SL [TR-1800]

- SMA mantra: To handle richer logic, use a more expressive domain

- To handle SL, use the shape-analysis domain *a la* TVLA (Three-Valued Logic Analyzer) [TOPLAS'02]
  - Parametric analysis framework for reasoning about the heap using first-order logic and canonical abstraction



Verification of Programs
Computing abstract transformers
Abstract Interpretation
α
Abstraction
Satisfiability Modulo Abstraction
Decision Procedures for Logics
Generalized Stålmarck's method
Thank you!