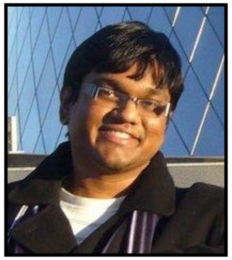
Recovering Components from Executables
[Cooperative Agreement HR0011-12-2-0012]

Thomas Reps

University of Wisconsin

Thomas Reps



Venkatesh Karthik Srinivasan

Tushar Sharma

Divy Vasal

Aditya Thakur

Evan Driscoll

Project Goals

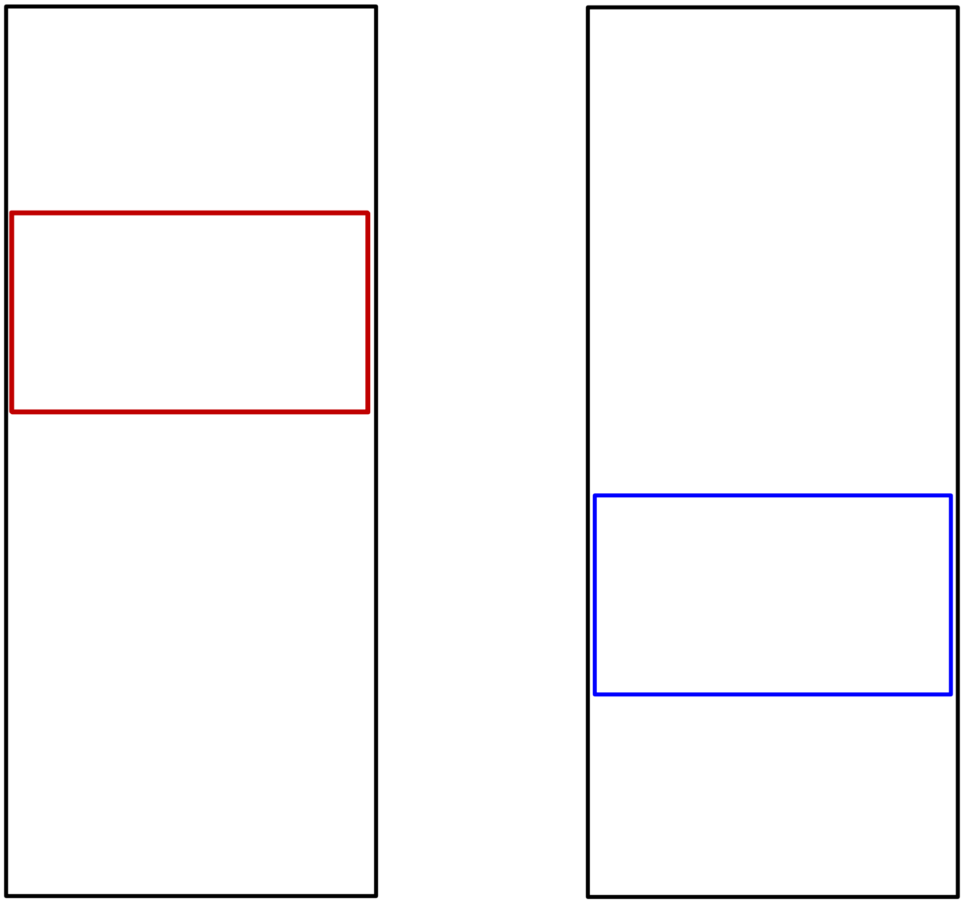
- Develop a "redeveloper's workbench"

  Tools to identify and extract components, and establish their behavioral properties

  – Aid in the harvesting of components from an executable
    - identify components
    - make adjustments to components identified
    - issue queries about a component's properties

  – Queries
    - type information; function prototypes
    - side-effect "footprint"
    - error-triggering properties

Basic scenario



Project Activities

- Component identification
  - Recovering class hierarchies using dynamic analysis
    - group functions into classes
    - identify inheritance and delegation relationships among the inferred classes
- Component extraction
  - Specialization slicing
    - create multiple specialized versions of a procedure, each equipped with a different subset of the original procedure's parameters
    - novel algorithm creates optimal specialization slice
  - Partial evaluation of machine code
    - general method to address extraction, specialization, and optimization of machine code
- Verifying component properties
  - Symbolic abstraction (BET + ONR STTR)
    - methods to obtain most-precise results in abstract interpretation
    - for a given abstract domain, attains the limit of what is achievable by any analysis algorithm
  - Domain-combination technique: combine results from multiple analysis methods
  - Abstract domain of bit-vector inequalities
    - allows a tool to identify inequality invariants for machine arithmetic (arithmetic mod 2^{32} or 2^{64})
    - fills a long-standing need in both source-code and machine-code analysis
  - Format-compatibility checking (ONR)

Outline of Talk

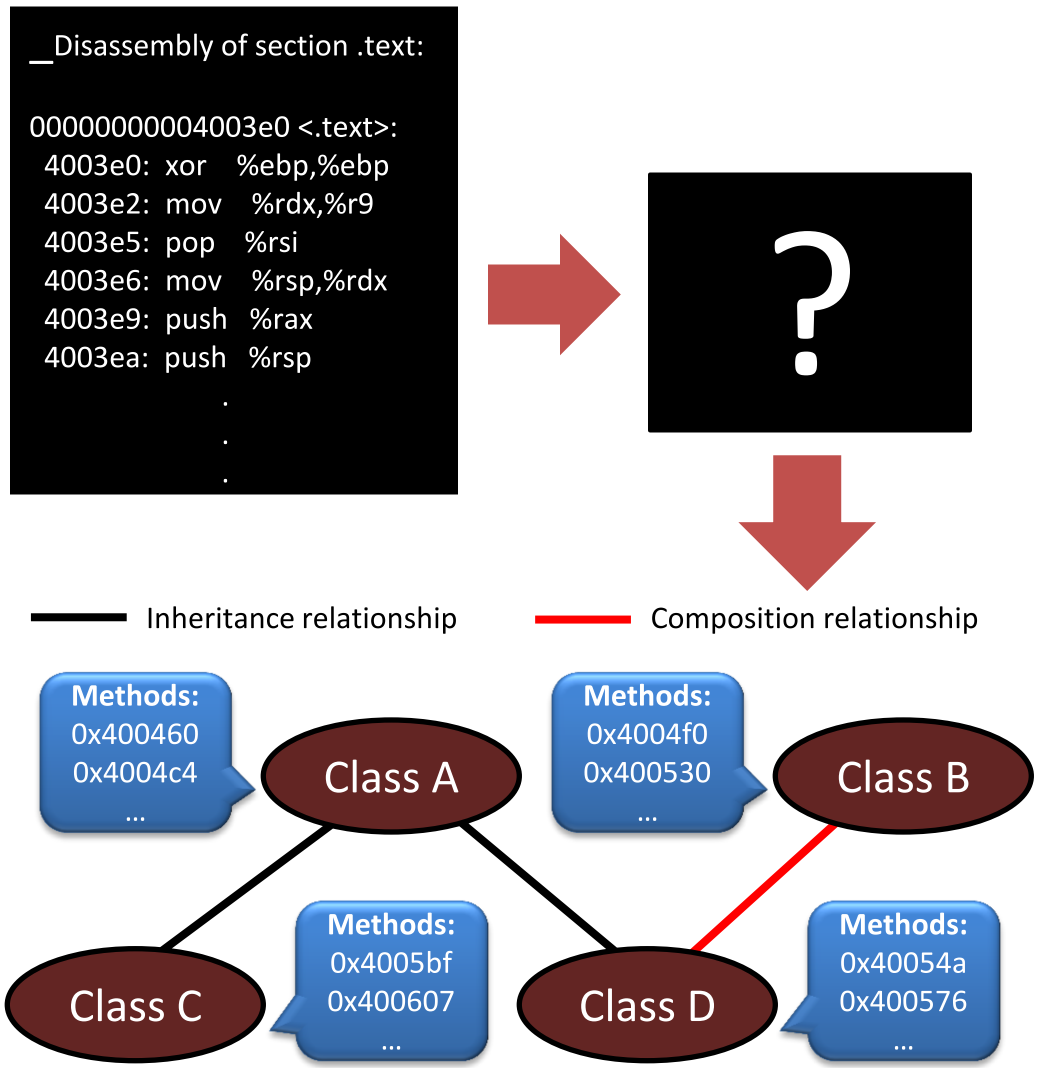
- Review of goals
- Progress (Oct. 2012 - May 2013)
  - Component identification
    - Recovering class hierarchies using dynamic analysis
  - Component extraction
    - Specialization slicing
    - Partial evaluation of machine code
  - Verifying component properties
    - Symbolic abstraction (BET + ONR STTR)
    - Domain-combination technique: combine results from multiple analysis methods
    - Abstract domain of bit-vector inequalities
    - Format-compatibility checking (ONR)
- Recap of publications/submissions
- Recap of plans for 2013

Outline of Talk

- Review of goals
- Progress (Oct. 2012 - May 2013)
  - Component identification
    - Recovering class hierarchies using dynamic analysis
  - Verifying component properties
    - Symbolic abstraction (BET + ONR STTR)
    - Domain-combination technique: combine results from multiple analysis methods
    - Abstract domain of bit-vector inequalities
    - Format-compatibility checking (ONR)
  - Component extraction
    - Specialization slicing
    - Partial evaluation of machine code
- Recap of publications/submissions
- Recap of plans for 2013

Recovering Class Hierarchies

- Given:
  - Stripped binary
- Goals:
  - Group functions in the binary into classes
  - Identify inheritance and composition relationships between inferred classes



Recovering Class Hierarchies

- Why?
  - Reengineering legacy software
  - Understanding architecture of software that lack documentation and source code
- Lego
  - Dynamic analysis tool
  - Recovers software architecture
  - Modulo code coverage

Key Ideas

- "this" pointer idiom
  - Common idiom in object-oriented programming
  - "this" pointer = 1st argument of methods of a class
  - Used to classify sets of functions

```
void SetID(int nID)
```

```
void SetID(Simple* const this, int nID)
```

- Unique finalizer idiom
  - Unique method in each class (Destructor in C++)
  - Cleans up object
  - Parent-class finalizer called at end of child-class finalizer
  - Used to recover inheritance and composition relationships

Lego – 2 Phases

- Phase 1
  - Input: stripped binary and test input
  - Executes given binary under test input
  - Performs dynamic analysis by dynamic binary instrumentation
  - Records methods invoked on allocated objects
  - Output: object-traces (summary of lifetime of every object)
- Phase 2
  - Input: object-traces
  - Uses order of finalizer calls as evidence from object-traces to infer class hierarchies
  - Output: Inferred class hierarchy and composition relationships between inferred classes

Phase 1: Object-Traces

- A sequence of method calls and returns that have the same receiver object

```
class Vehicle {
  public:
    Vehicle();
    ~Vehicle();
};

class Car :
public Vehicle {
  public:
    Car(int n);
    ~Car();
    void print_car();
  private:
    void helper();
};

class Bus :
public Vehicle {
  public:
    Bus();
    ~Bus();
    void print_bus();
};

int main() {
  Car c(10);
  Bus b;
  c.print_car();
  b.print_bus();
  return 0;
}
```

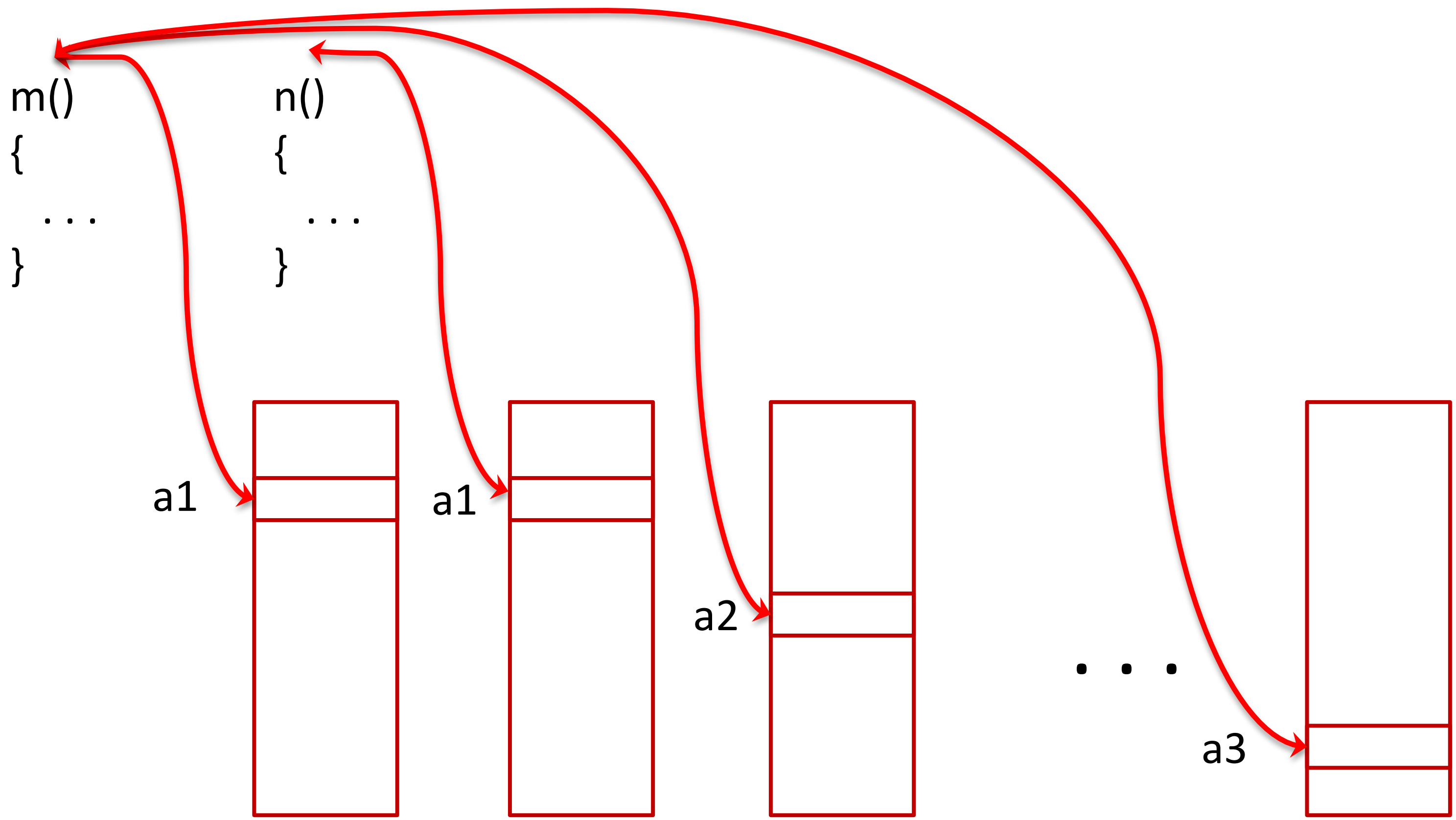
→

```
0xAAAA
(Address of c):
Car(int) C
Vehicle() C
Vehicle() R
Car(int) R
print_car() C
print_car() R
~Car() C
helper() C
helper() R
~Vehicle() C
~Vehicle() R
~Car() R

0xBBBB
(Address of b):
Bus() C
Vehicle() C
Vehicle() R
Bus() R
print_bus() C
print_bus() R
~Bus() C
~Vehicle() C
~Vehicle() R
~Bus() R
```

Object Traces – How to get them?

- Instrument binary using PIN to trace:
  - Values of 1st-arguments of methods
  - Method calls and returns
- Emit a trace of <“this” pointer, method Call/Return> pairs
- Group methods based on “this”-pointer values
- From the trace, compute *object-traces*, pairs <A, S> where
  - A is an object address
  - S is the sequence of method calls/returns that were passed A as the value of the “this” pointer (1st argument)

Object-Traces



Emitted Trace

. . . <a1, m, C> . . <a1, n, C> . . . <a1, n, R> . . <a1, m, R> . . .
<a2, m, C> . . <a2, m , R> . . <a3, m, C> . . <a3, m , R>

Object Traces [a1: <m, C>, <n, C>, <n, R>, <m, R>], [a2: <m, C>, <m, R>], [a3: <m, C>, <m, R>]

Challenges – Blacklisting Methods

- Stand-alone methods and static methods don't receive a "this" pointer

```
void foo();     static void Car::setInventionYear(int a);
```

- Lego maintains estimates of allocated address space
  - Stack pointer values during calls and returns
  - Allocated heap objects – instrument new and delete
- If 1st argument's value of a method is not within allocated address space, method is blacklisted
  - Removed from existing object-traces
  - Never added to future object-traces

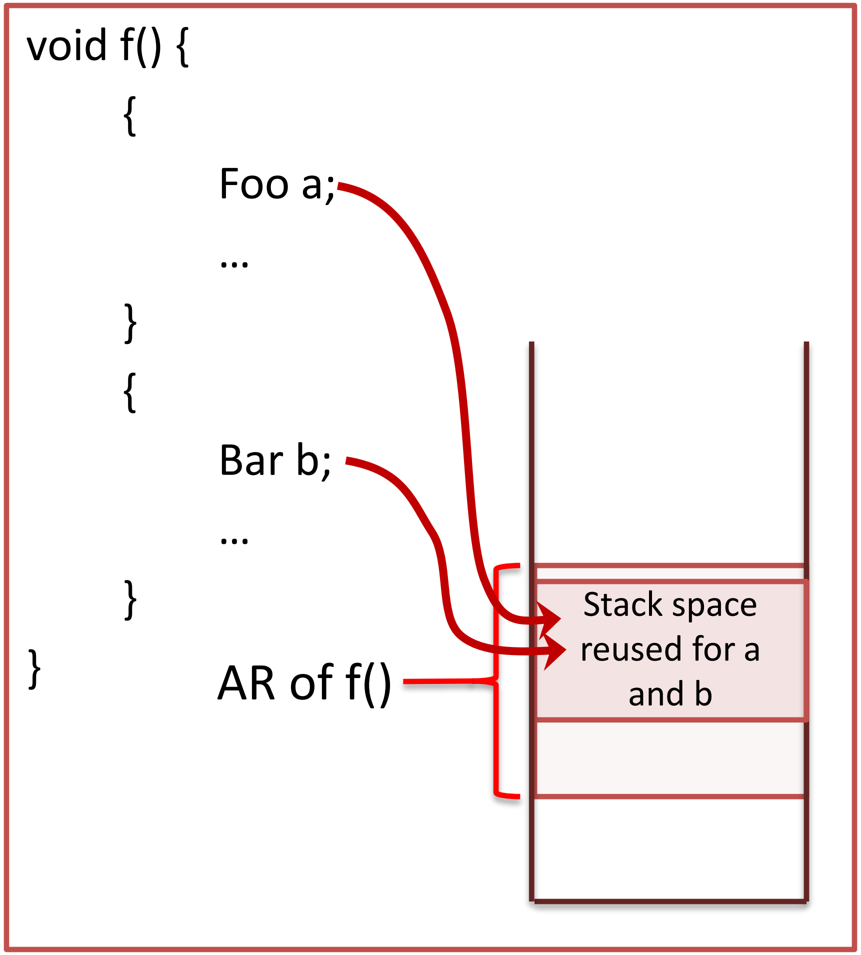
Challenges – Object-address Reuse

```
class A {
  public:
    . . .
    printA();
};

class B {
  public:
    . . .
    printB();
};

void foo() {
  A a;
  a.printA();
}

void bar() {
  B b;
  b.printB();
}

int main {
  foo();
  bar();
  return 0;
}
```

- Methods of two (or more) unrelated classes appear in same object-trace
- Reuse of stack space for objects on different Activation Records (ARs)
- Reuse of same heap space by heap manager
- Lego versions addresses – increment version of address A when A is deallocated

Challenges – Spurious Traces

- Spurious traces
  - Methods of two (or more) unrelated classes appear in the same object-trace
  - Reuse of same stack space by compiler for different objects in different scopes within same AR
  - Locate initializer and finalizer methods to split spurious traces



Phase 2: Object-Trace Fingerprints

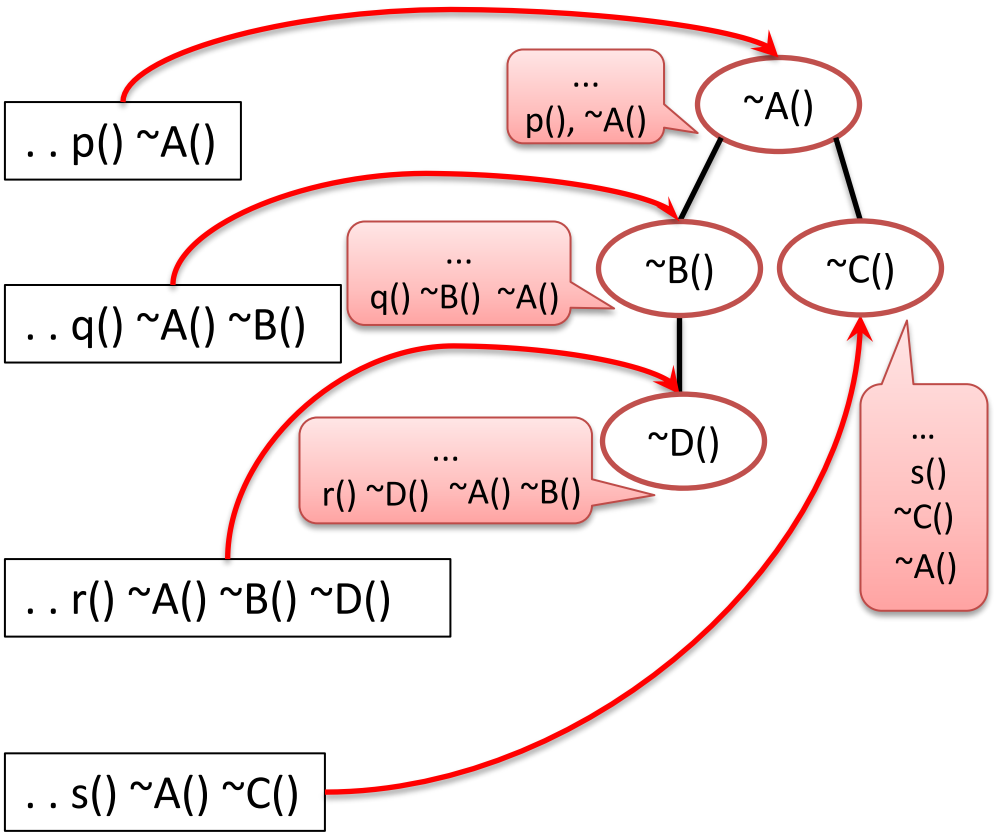
- Common semantics of OO languages – derived class's finalizer calls base finalizer just before returning
- Fingerprint – 'return-only' suffix of object-trace
- 'return-only' – Methods that were called just before caller returned
- Has methods involved in cleanup of object and inherited parts

```
class A {
  ~A();
};

class B :
public A {
  ~B();
};

class C :
public B {
  ~C();
  helper();
};

class D:
public C {
  ~D();
};

~D()   C
~C()   C
helper() C
helper() R
~B()   C
~A()   C
~A()   R
~B()   R
~C()   R
~D()   R
```

- Length indicates possible number of levels in class hierarchy
- Methods in fingerprint – potential finalizers in the class and ancestor classes

Finding Class Hierarchies

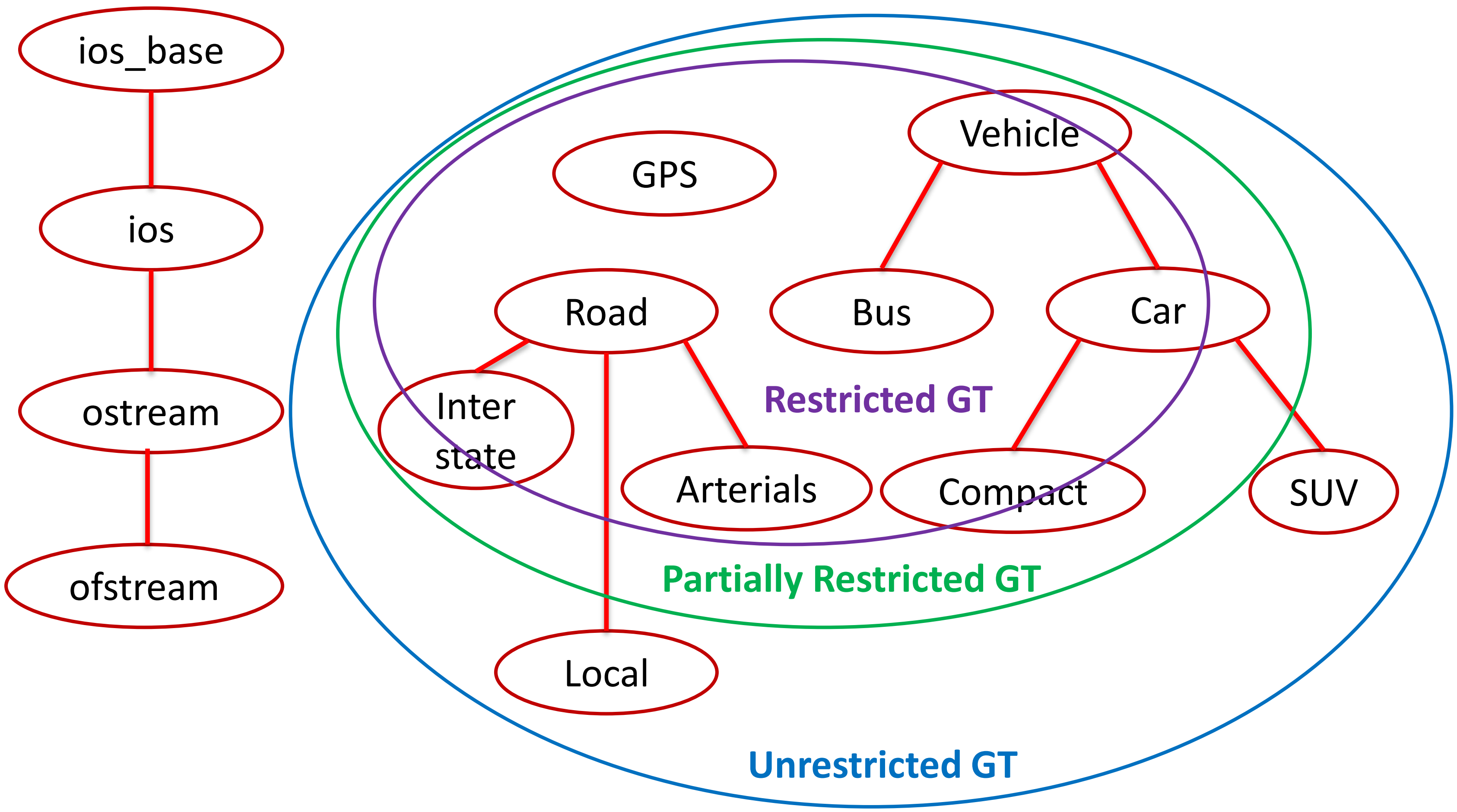
- Create a trie from fingerprints
- Associate each object-trace with trie node that accepts object-trace's fingerprint
- Add methods in each object-trace to associated trie node
- If parent and child nodes have common methods, remove common methods from child



Composition Relationships

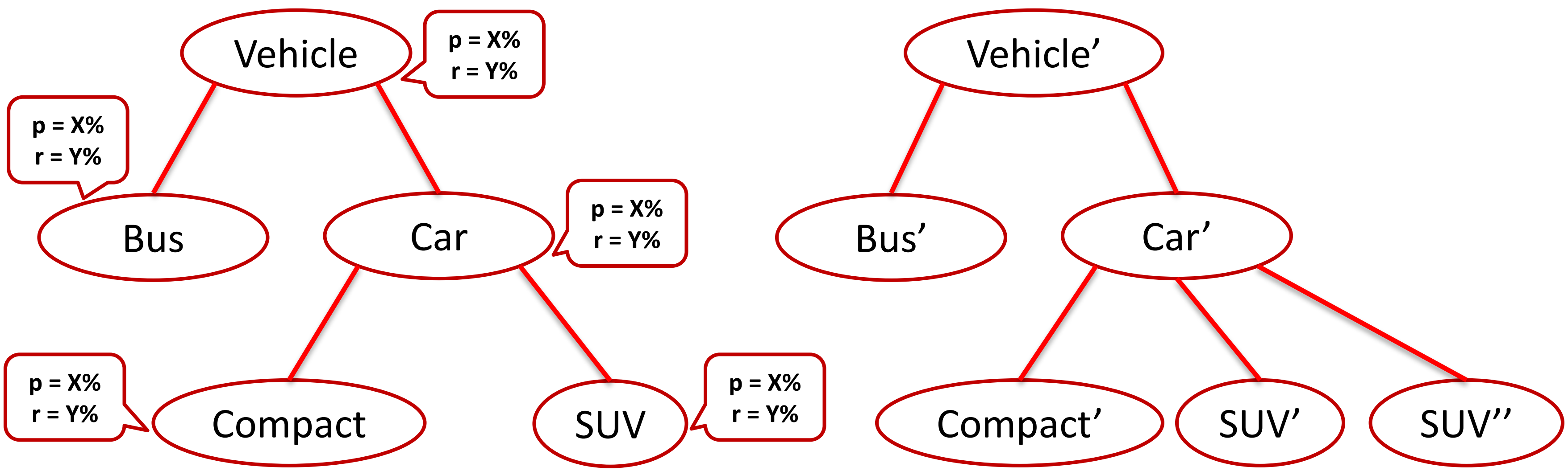
- Class A has a member instance of B
- A is responsible for cleaning up B – A's finalizer calls B's finalizer
- Record the methods directly called by each method in object-trace
- Conditions for a composition relationship to exist between inferred classes A and B
  - A's finalizer calls B's finalizer
  - A is not B's ancestor or descendant in the inferred hierarchy

Scoring – Ground Truth

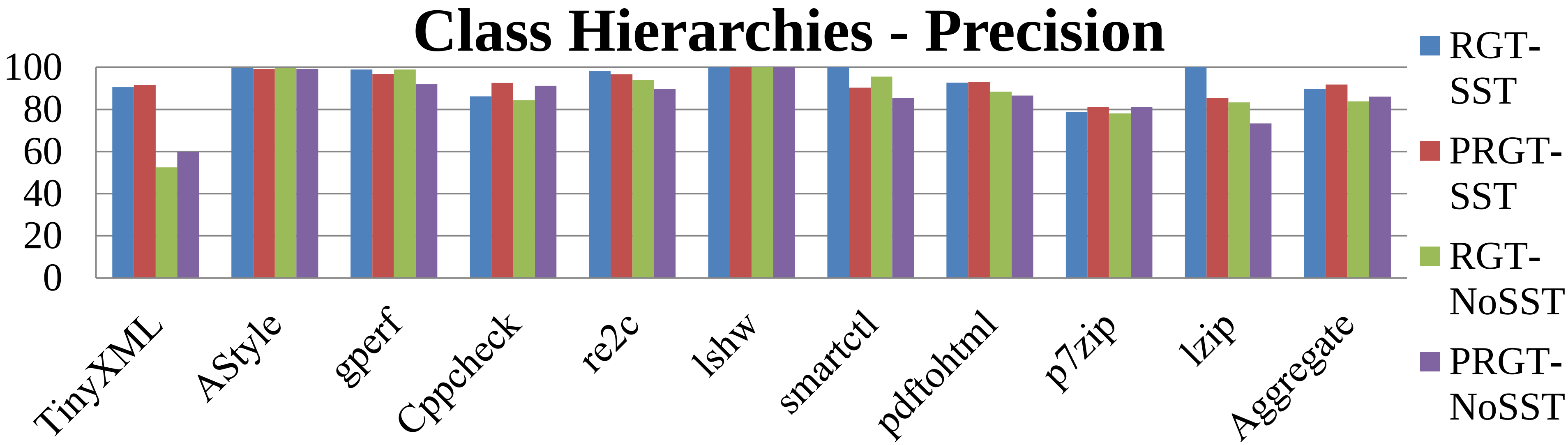


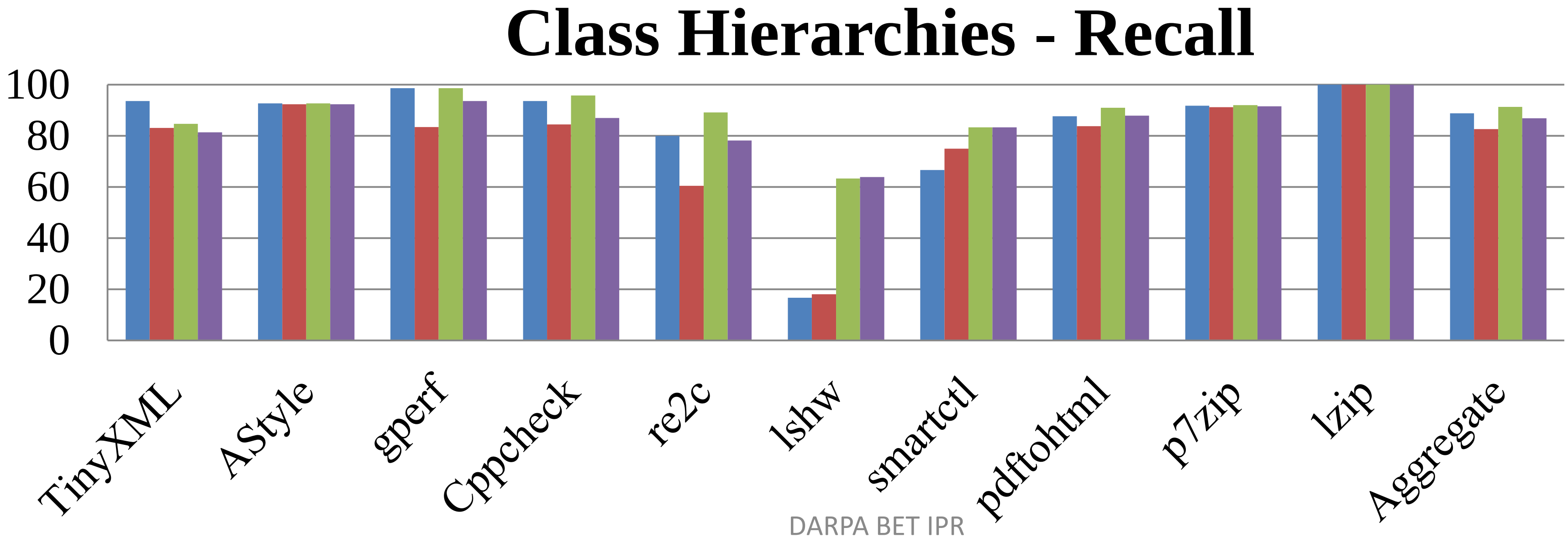
Scoring

- Precision and Recall
- Can't treat classes as flat sets of methods – inheritance relationships between classes
- For every path in the GT inheritance hierarchy, find the path in the inferred hierarchy that gives maximum precision and recall



Results


Class Hierarchies - Precision
100
80
60
40
20
0
TinyXML
AStyle
gperf
Cppcheck
re2c
lshw
smartctl
pdftohtml
p7zip
lzip
Aggregate
RGT-SST
PRGT-SST
RGT-NoSST
PRGT-NoSST


Class Hierarchies - Recall
100
80
60
40
20
0
TinyXML
AStyle
gperf
Cppcheck
re2c
lshw
smartctl
pdftohtml
p7zip
lzip
Aggregate

Outline of Talk

- Review of goals
- Progress (Oct. 2012 - May 2013)
  - Component identification
    - Recovering class hierarchies using dynamic analysis
  - Verifying component properties
    - Symbolic abstraction (BET + ONR STTR)
    - Domain-combination technique: combine results from multiple analysis methods
    - Abstract domain of bit-vector inequalities
    - Format-compatibility checking (ONR)
  - Component extraction
    - Specialization slicing
    - Partial evaluation of machine code
- Recap of publications/submissions
- Recap of plans for 2013

Verifying component properties

- Property holds for all possible inputs

- No null-pointer deferences
- No accesses outside array bounds
- No stack smashing
- No division by zero ✔

```
while(1) {
  x = input();
  If (x > 0)  {
    y = 2*x;
    z = w/y;
  }
}
```

$y \rightarrow 2$
$y \rightarrow 8$
$y \rightarrow 42$
$y \rightarrow 178$
…

Sign Abstraction: only track whether variable is positive, negative, or zero

⇨ $y > 0$

Program statement

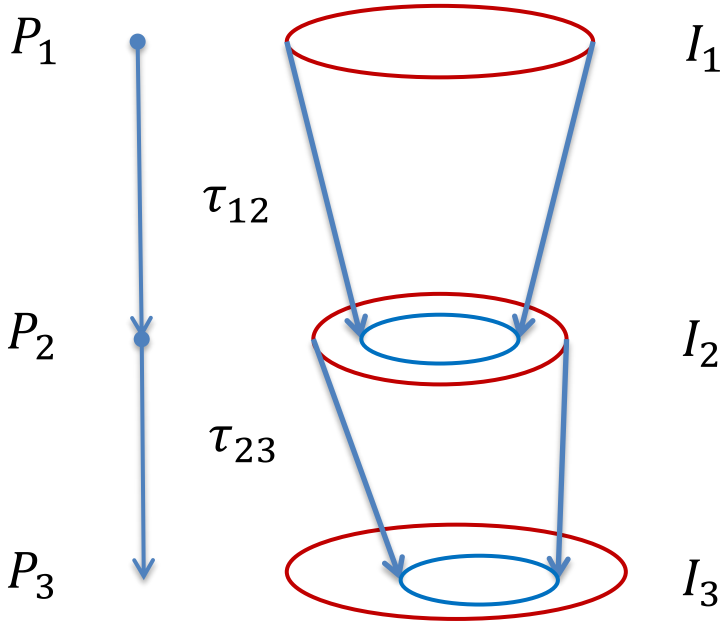
Possible concrete values of y

Invariant

Inductive Invariants

Program points

Inductive Invariants



Abstract Interpretation

Concrete

Concrete state $\mathcal{C}$

$[x \rightarrow 2, y \rightarrow 2, z \rightarrow -3]$
$[x \rightarrow 7, y \rightarrow 8, z \rightarrow -6]$

Concrete transformer
$\tau: \mathcal{C} \rightarrow \mathcal{C}$

Concrete execution

- Start with concrete input, one of the possibly infinite set of concrete inputs
- Apply τ for each statement
- Not guaranteed to terminate

Abstract

Abstract state $\mathcal{A}$ ✔

$[x > 0, y > 0, z < 0]$

Abstract transformer
$\tau^{\#}: \mathcal{A} \rightarrow \mathcal{A}$

Has to be sound, precise over-approximation

Abstract execution ✔

- Start with abstract input *that represents all possible concrete inputs*
- Apply $\tau^{\#}$ for each statement
- Guaranteed to reach *fixpoint*

Transformers via reinterpretation

- Define abstract operator $*^{\#}$ for each concrete operator $*$ in the program

$*^{\#}$	< 0	$= 0$	> 0
< 0	> 0	$= 0$	> 0
$= 0$	$= 0$	$= 0$	$= 0$
> 0	< 0	$= 0$	> 0

Transformers via reinterpretation

- Define abstract operator $*^{\#}$ for each concrete operator $*$ in the program

$*^{\#}$	< 0	$= 0$	> 0
< 0	> 0	$= 0$	< 0
$= 0$	$= 0$	$= 0$	$= 0$
> 0	< 0	$= 0$	> 0

Transformers via reinterpretation

- Compositionally define abstract transformers for statements using abstract operators

$[x > 0, y > 0, z < 0]$

$a =^{\#} (x *^{\#} y) *^{\#} z;$

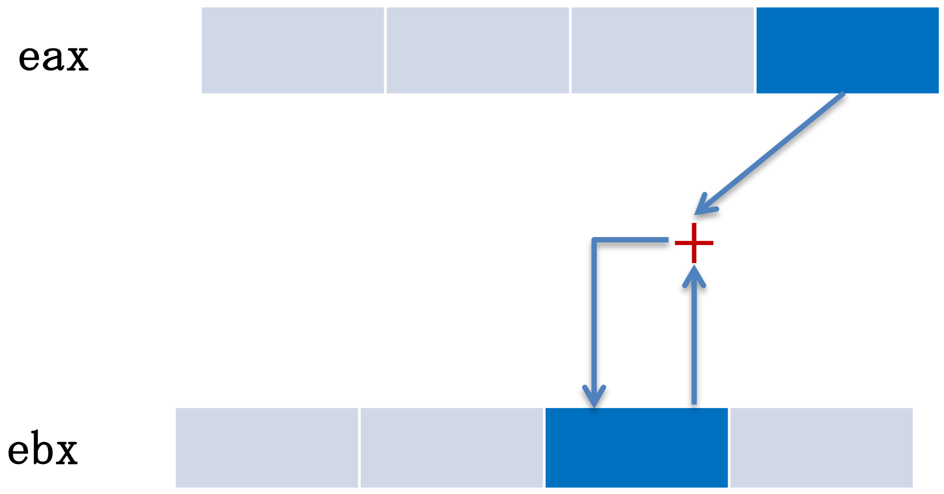
$[a < 0, x > 0, y > 0, z < 0]$

$*^{\#}$	< 0	$= 0$	> 0
< 0	> 0	$= 0$	< 0
$= 0$	$= 0$	$= 0$	$= 0$
> 0	< 0	$= 0$	> 0

Transformers via reinterpretation

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`



Transformers via reinterpretation

τ: add bh, al

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

$\text{ebx} - \text{ecx} = 0 \in \mathcal{A}$

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

Semantics expressed as a formula

$2^{24}\,\text{ebx}' - 2^{24}\,\text{ecx}' = 0 \in \mathcal{A}$

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

Not the most-precise value

Primed variables represent values in post-state.

Automation of best transformer

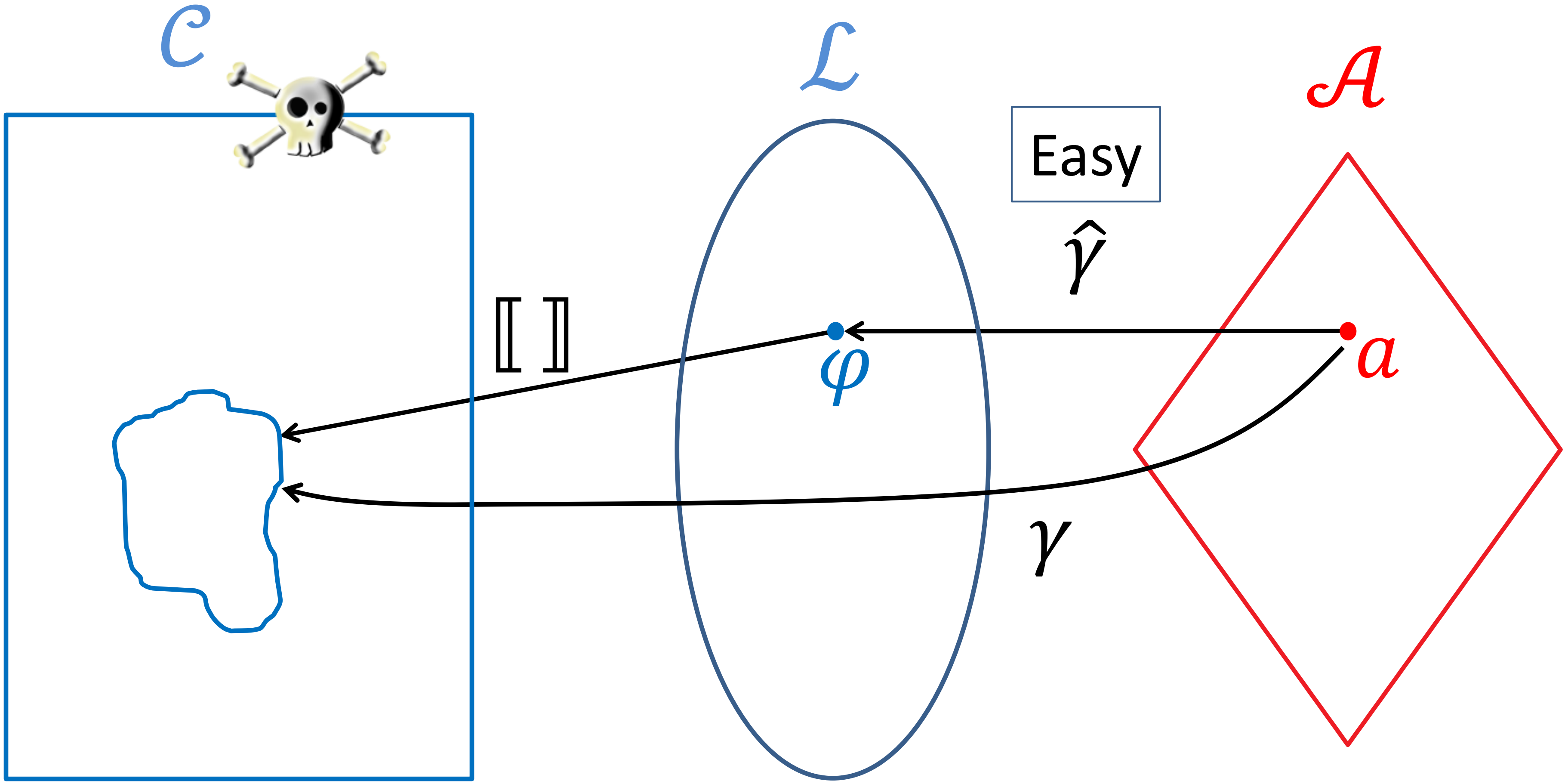
τ $a \in \mathcal{A}$

- Ensures correctness
- Ensures precision
- Reduces time to implement primitives

a'

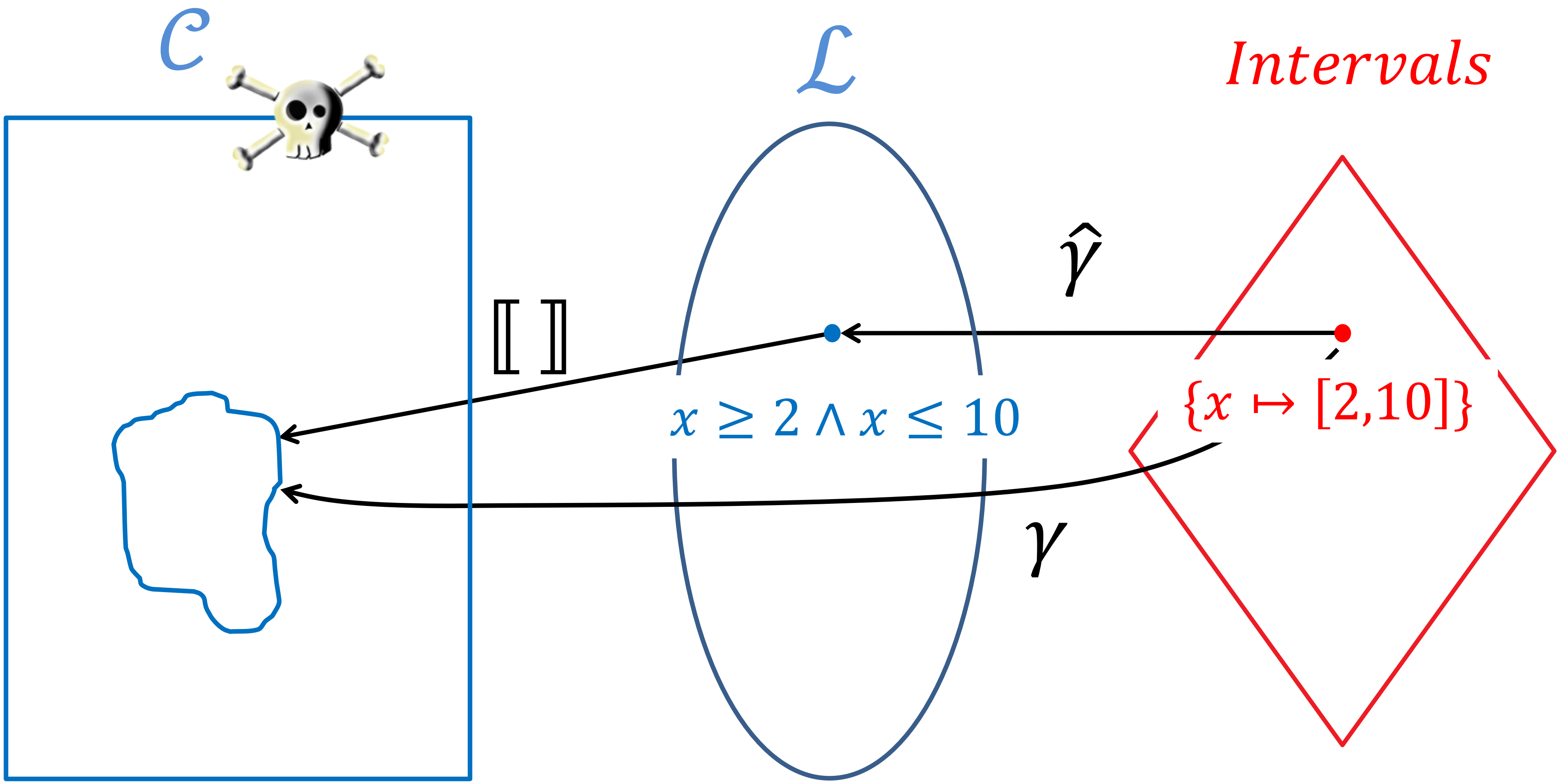
Application of best transformer

Symbolic Abstract Interpretation



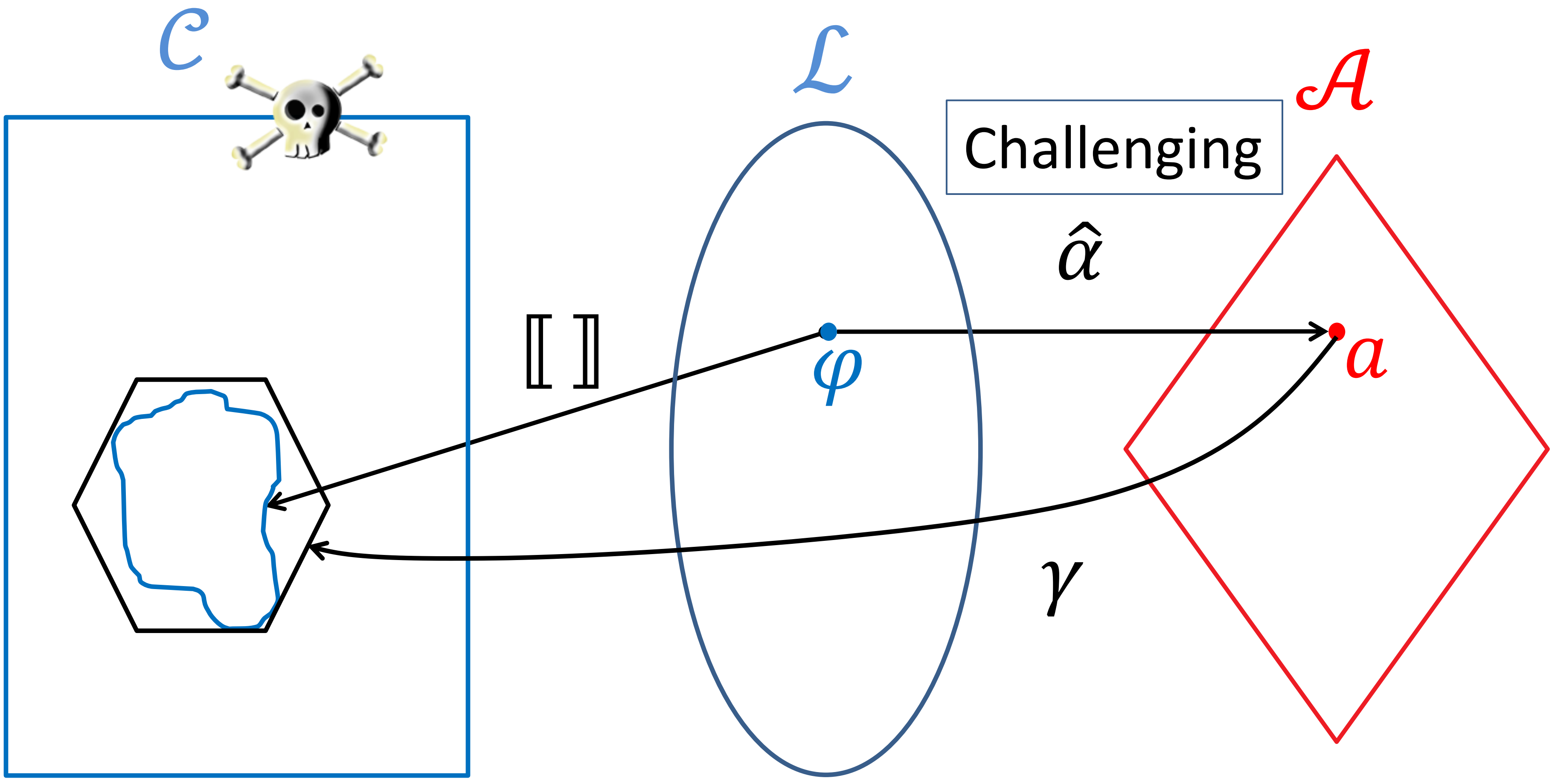
Symbolic Concretization

Symbolic Abstract Interpretation



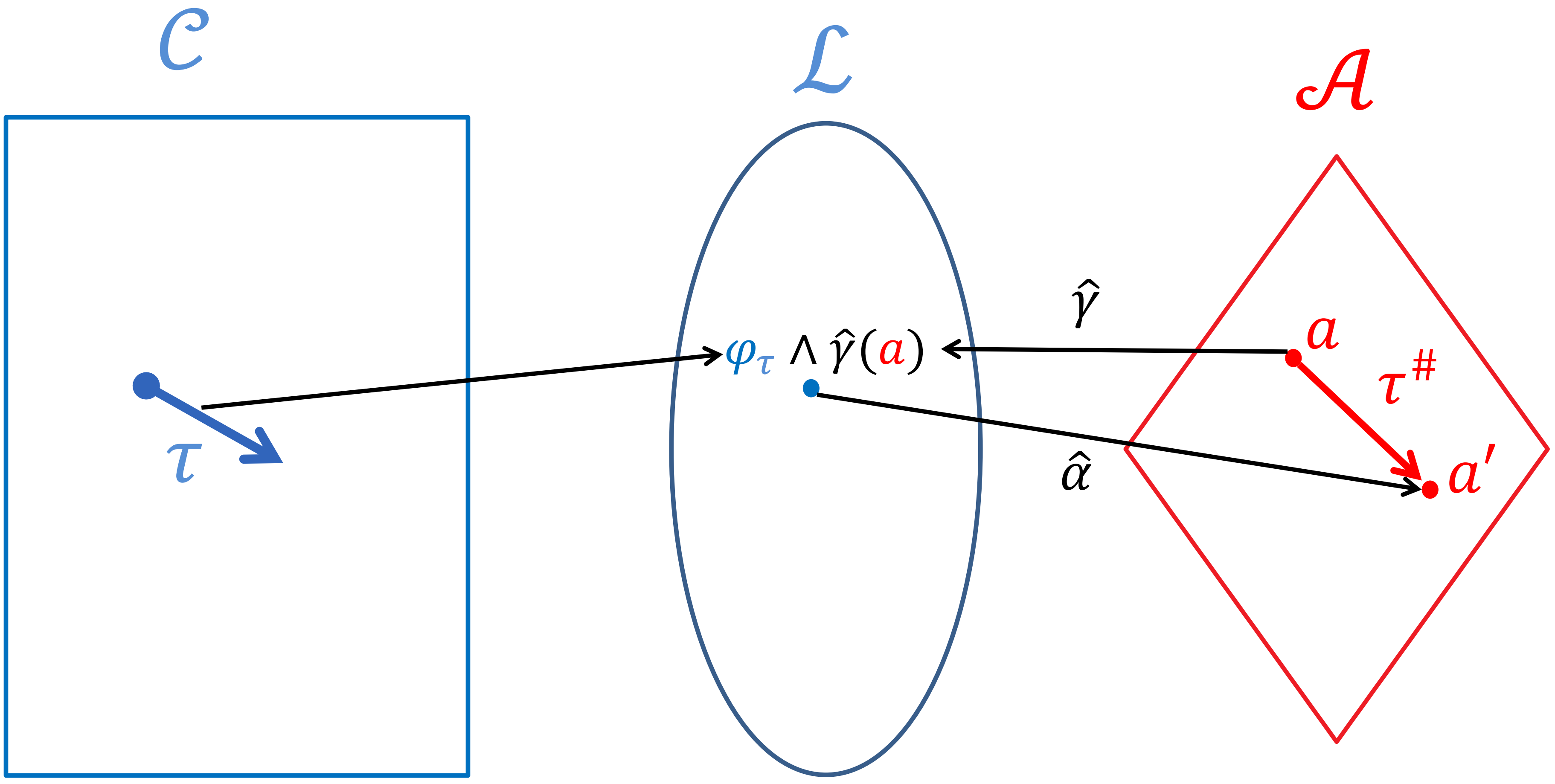
Symbolic Concretization

Symbolic Abstract Interpretation



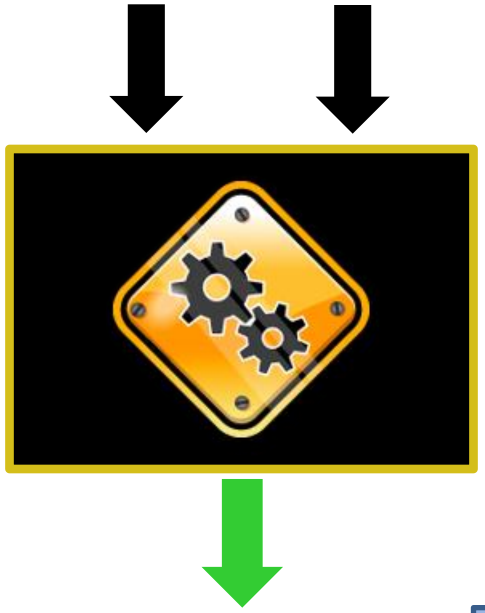
Symbolic Abstraction

Symbolic abstraction ⇒ best transformer



Automation of best transformer

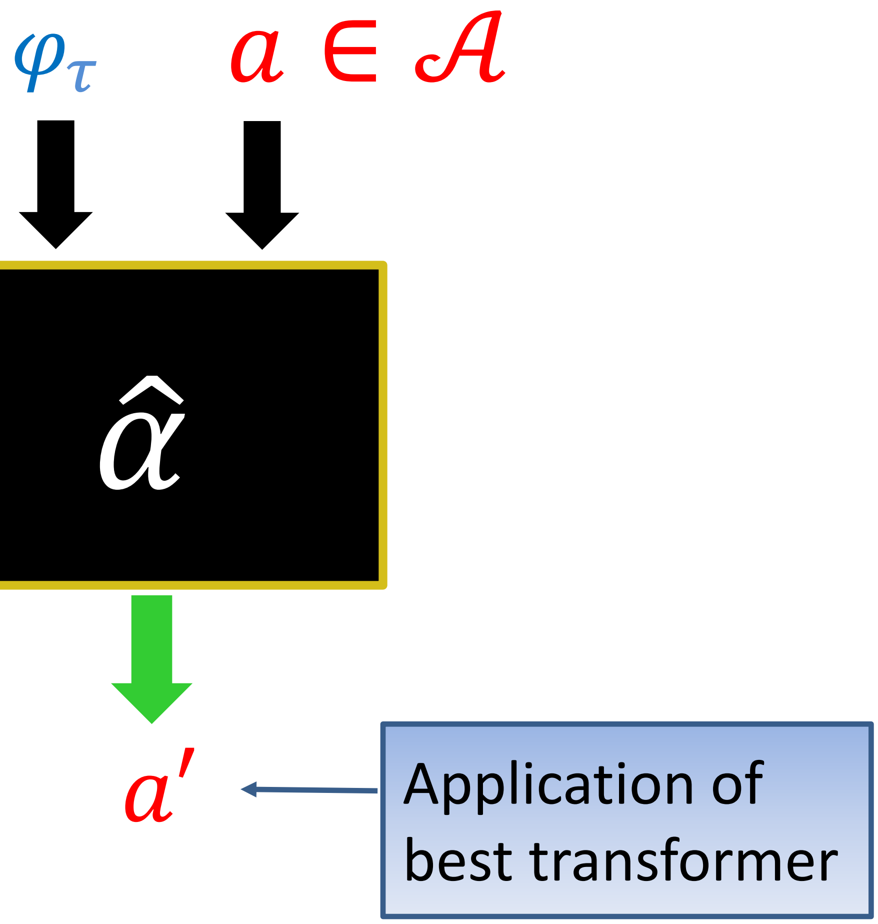
τ $a \in \mathcal{A}$



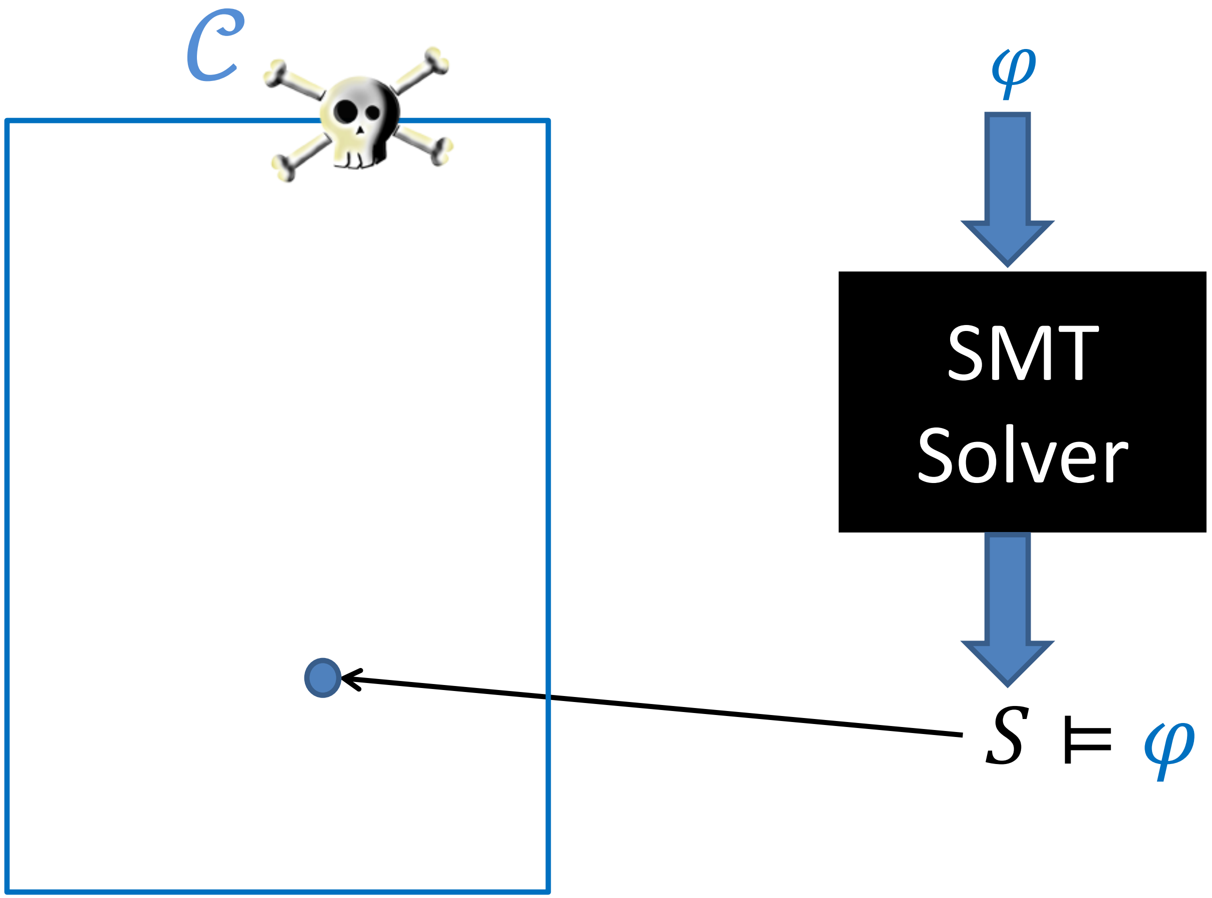
a'

Application of best transformer

Automation of best transformer

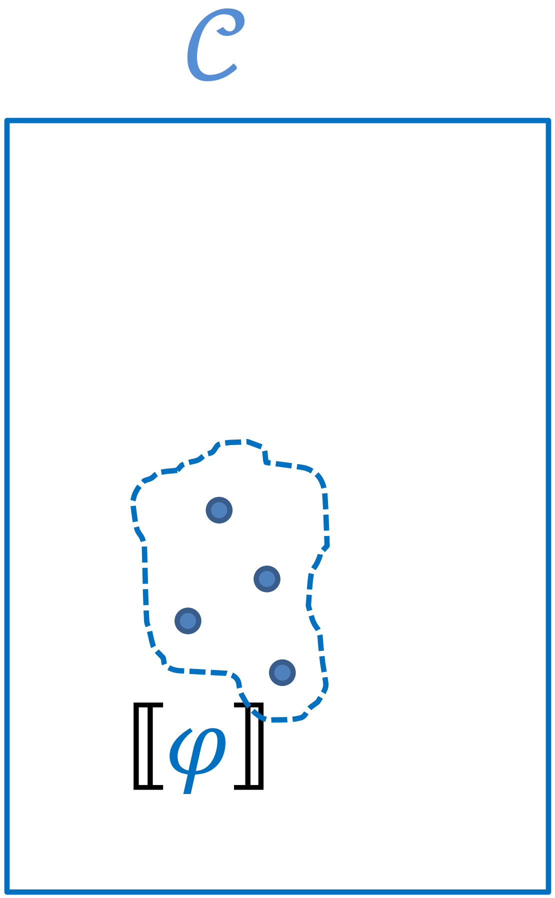


Algorithm for $\hat{\alpha}(\varphi)$



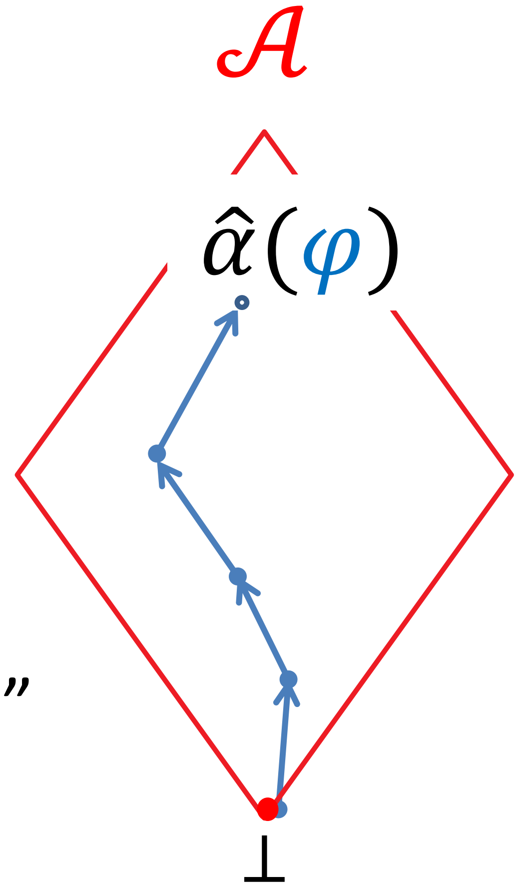
SMT:= Satisfiability Modulo Theory

RSY algorithm for $\hat{\alpha}(\varphi)$ [VMCAI'04]

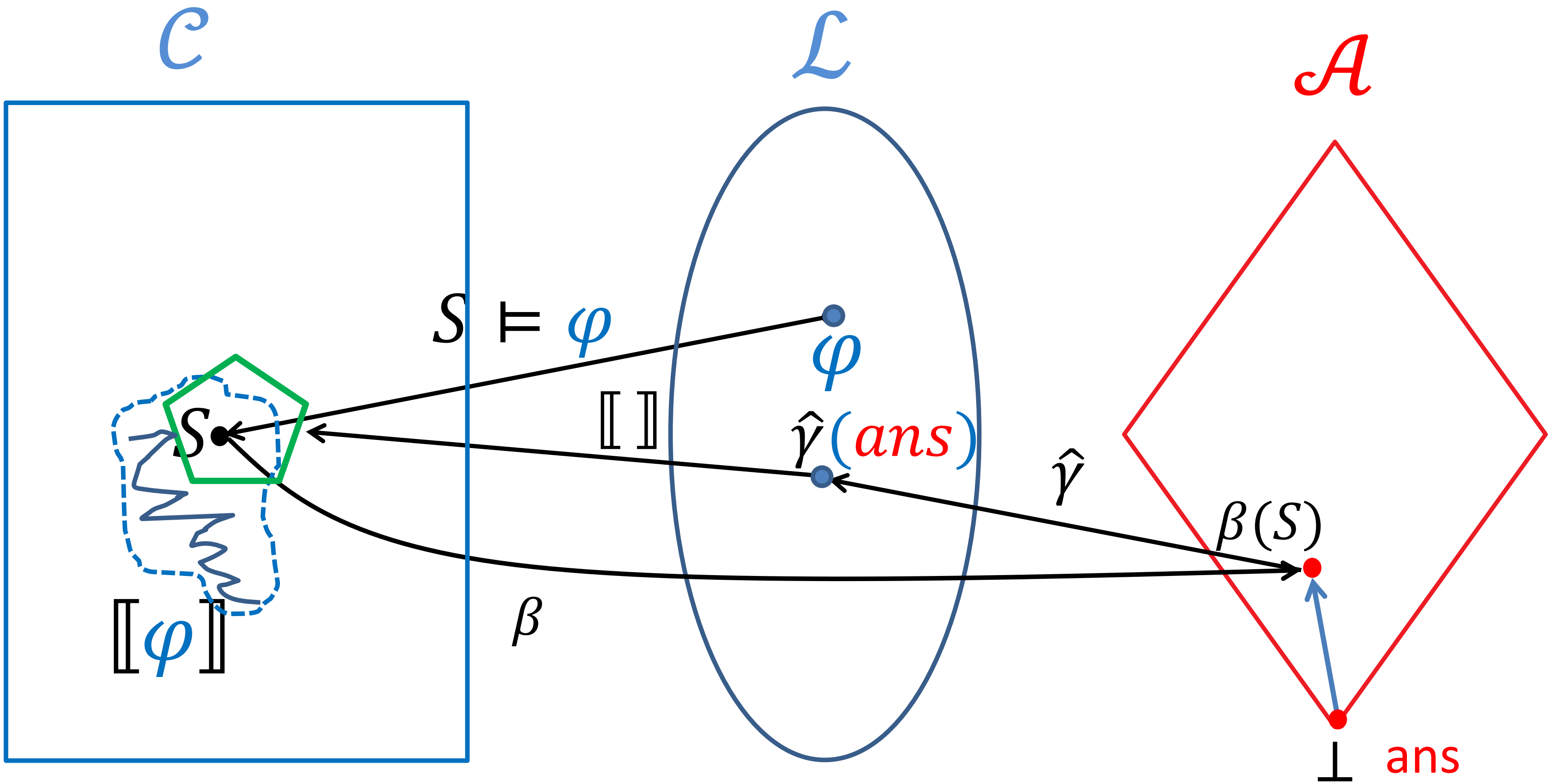


Smart sampling

Converge "from below"

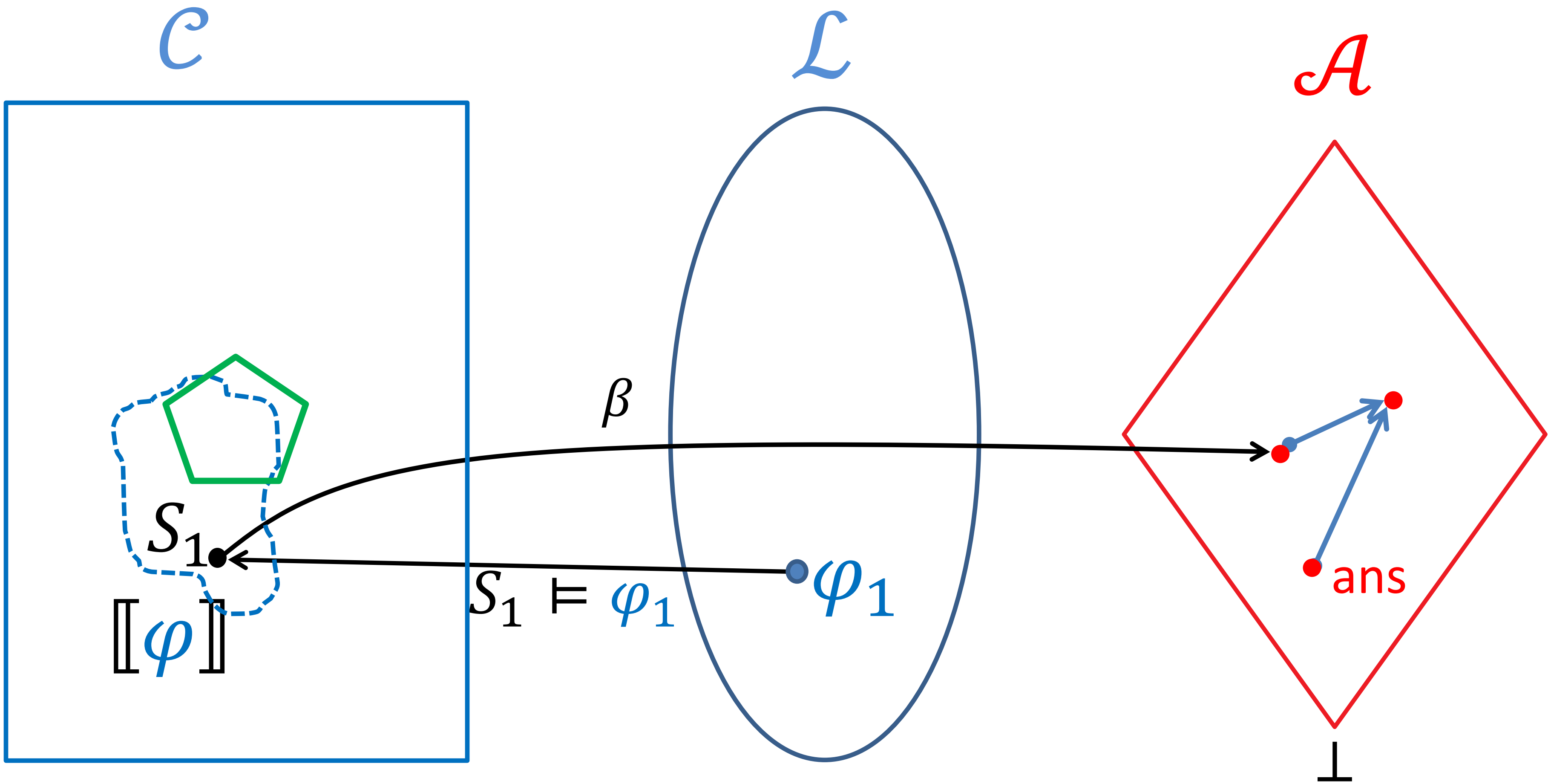


RSY algorithm for $\hat{\alpha}(\varphi)$ [VMCAI'04]



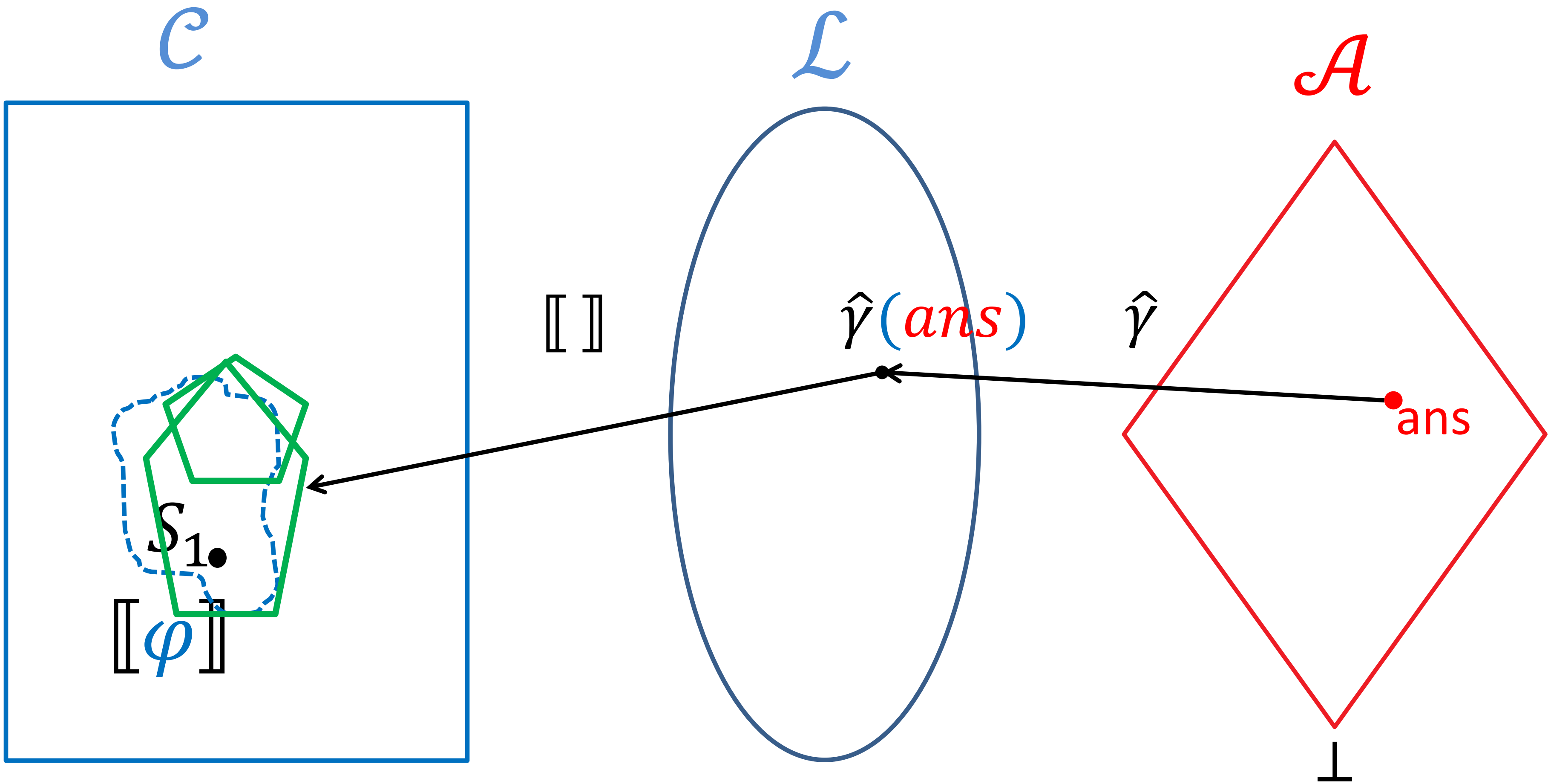
β: α for singleton set

RSY algorithm for $\hat{\alpha}(\varphi)$ [VMCAI'04]



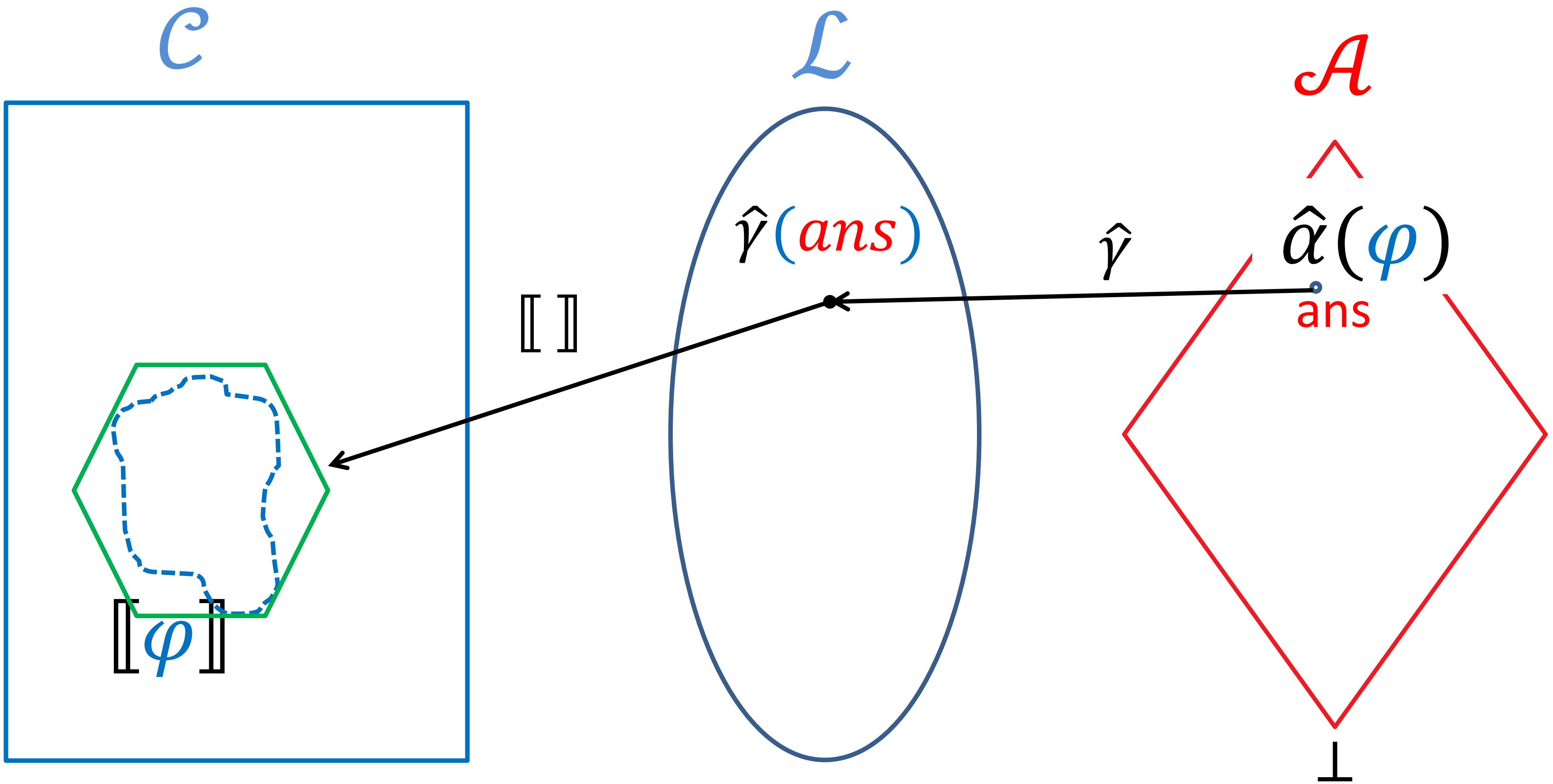
$$\varphi_1 = \varphi \land \neg\hat{\gamma}(\text{ans})$$

RSY algorithm for $\hat{\alpha}(\varphi)$ [VMCAI'04]



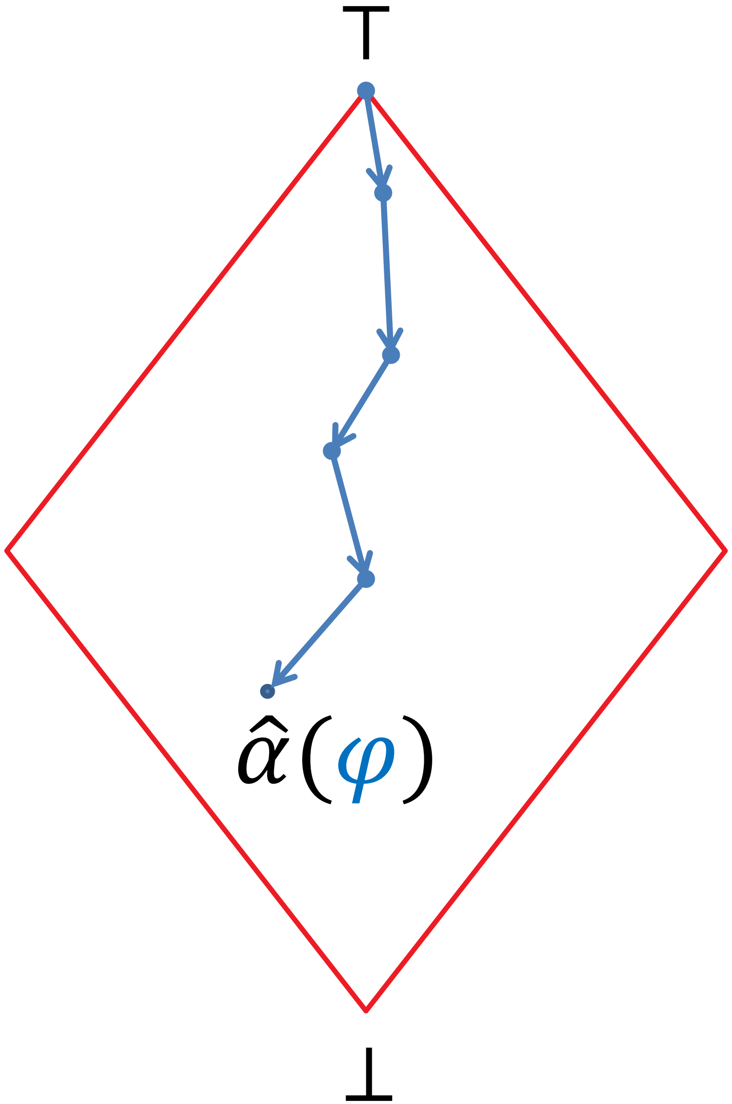
$$\varphi_1 = \varphi \wedge \neg\hat{\gamma}(\text{ans})$$

RSY algorithm for $\hat{\alpha}(\varphi)$ [VMCAI'04]



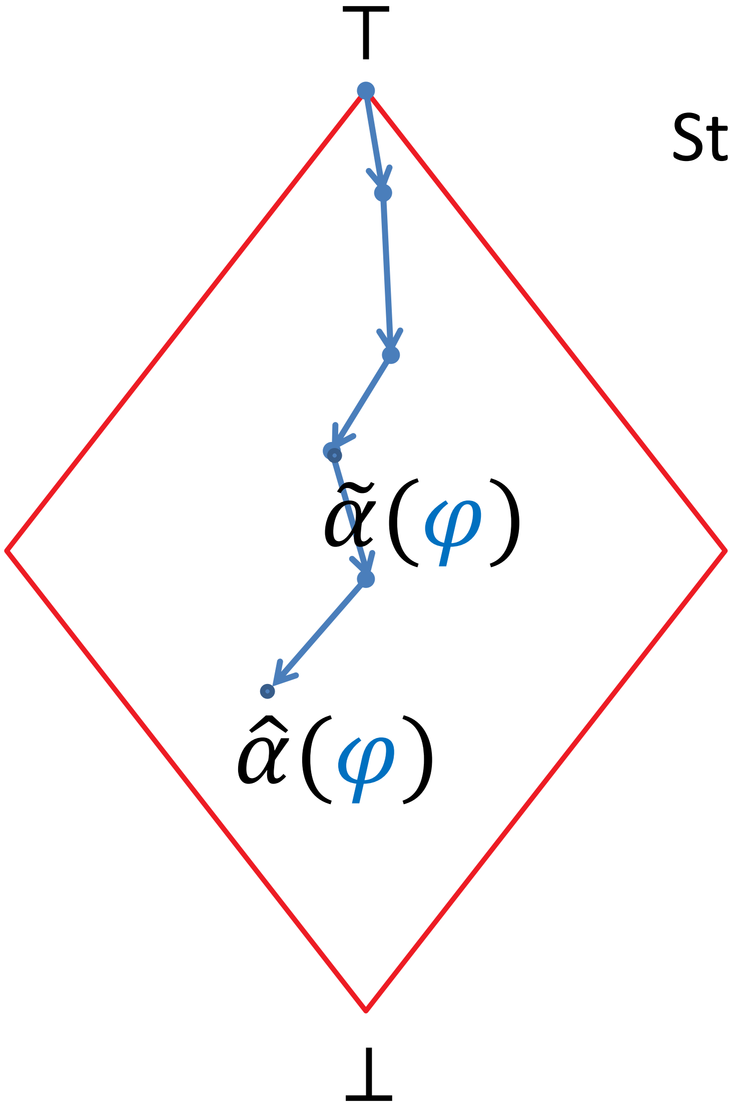
$\varphi_k = \varphi \wedge \neg\hat{\gamma}(\text{ans})$ UNSAT

Bilateral algorithm for $\hat{\alpha}(\varphi)$ [SAS'12]



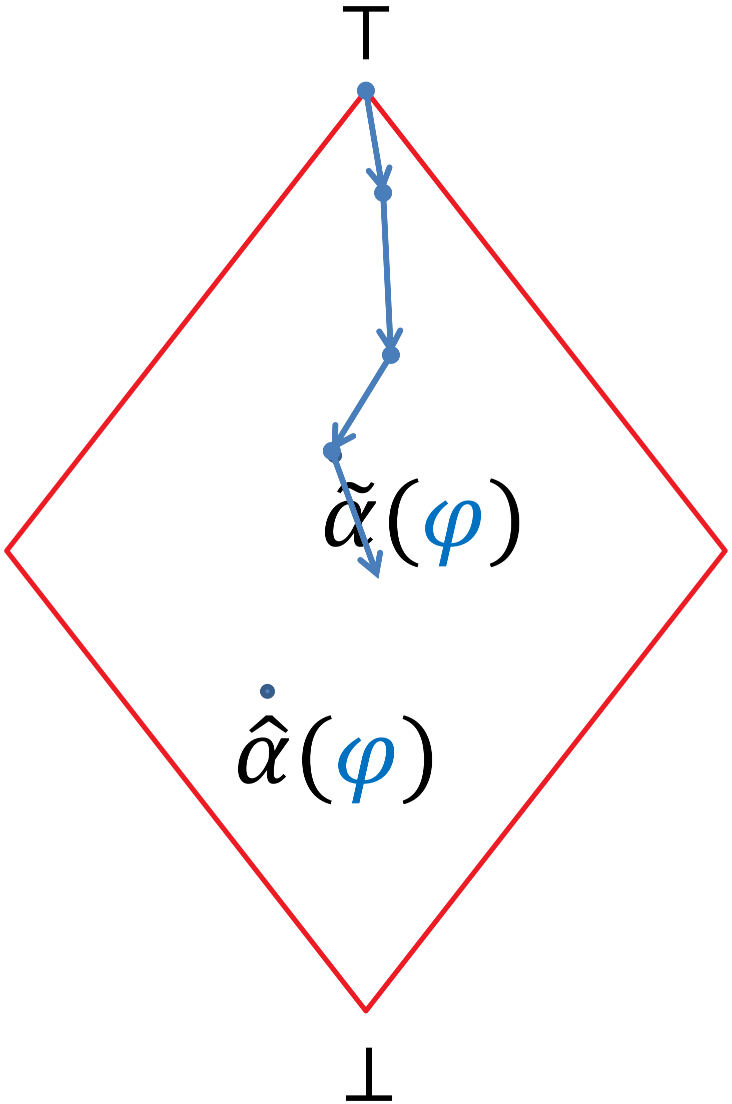
Converge "from below" and *"from above"*

Bilateral algorithm for $\hat{\alpha}(\varphi)$ [SAS'12]



Stop at any time→ sound answer

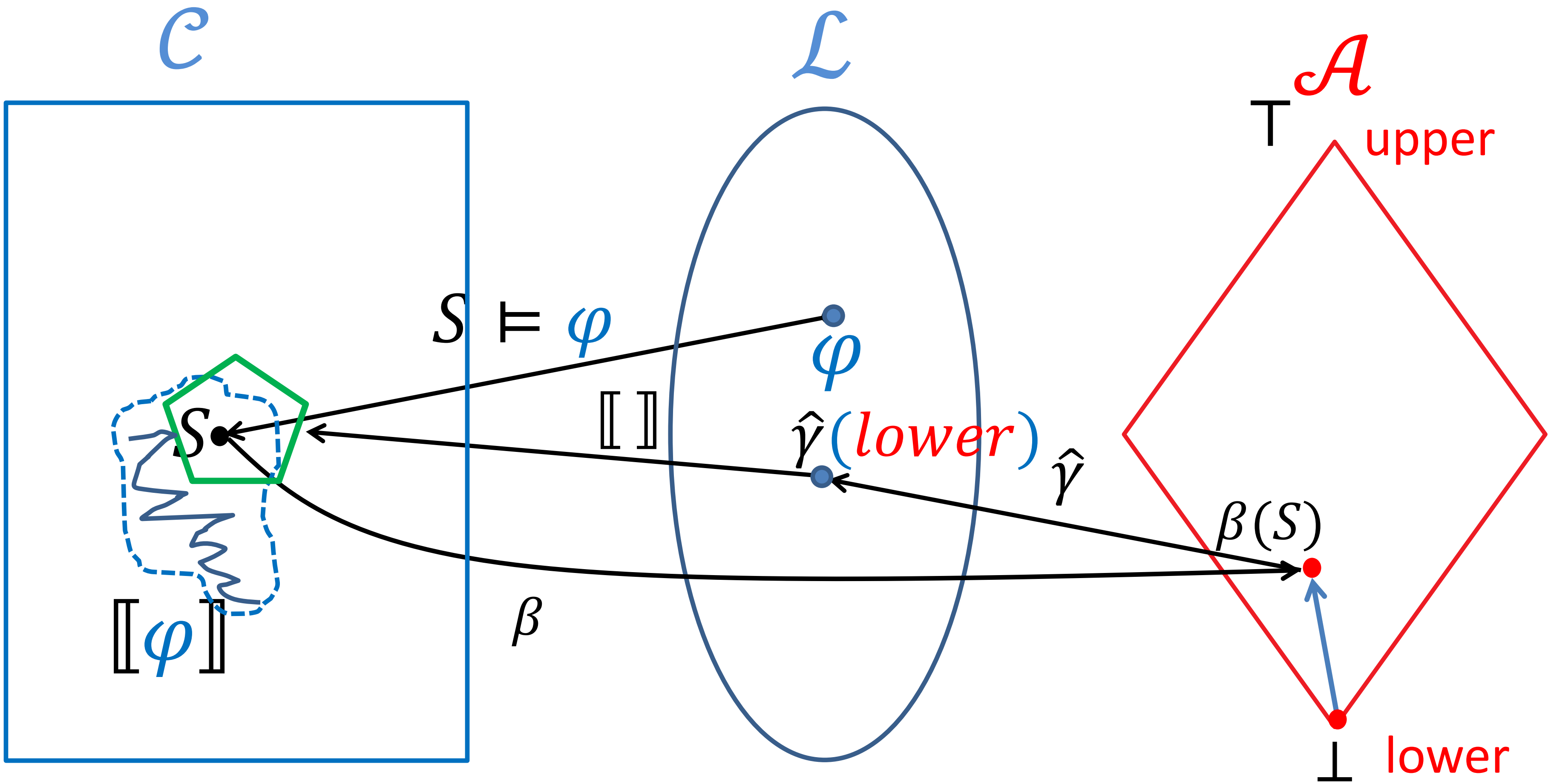
Bilateral algorithm for $\hat{\alpha}(\varphi)$ [SAS'12]



Tunable

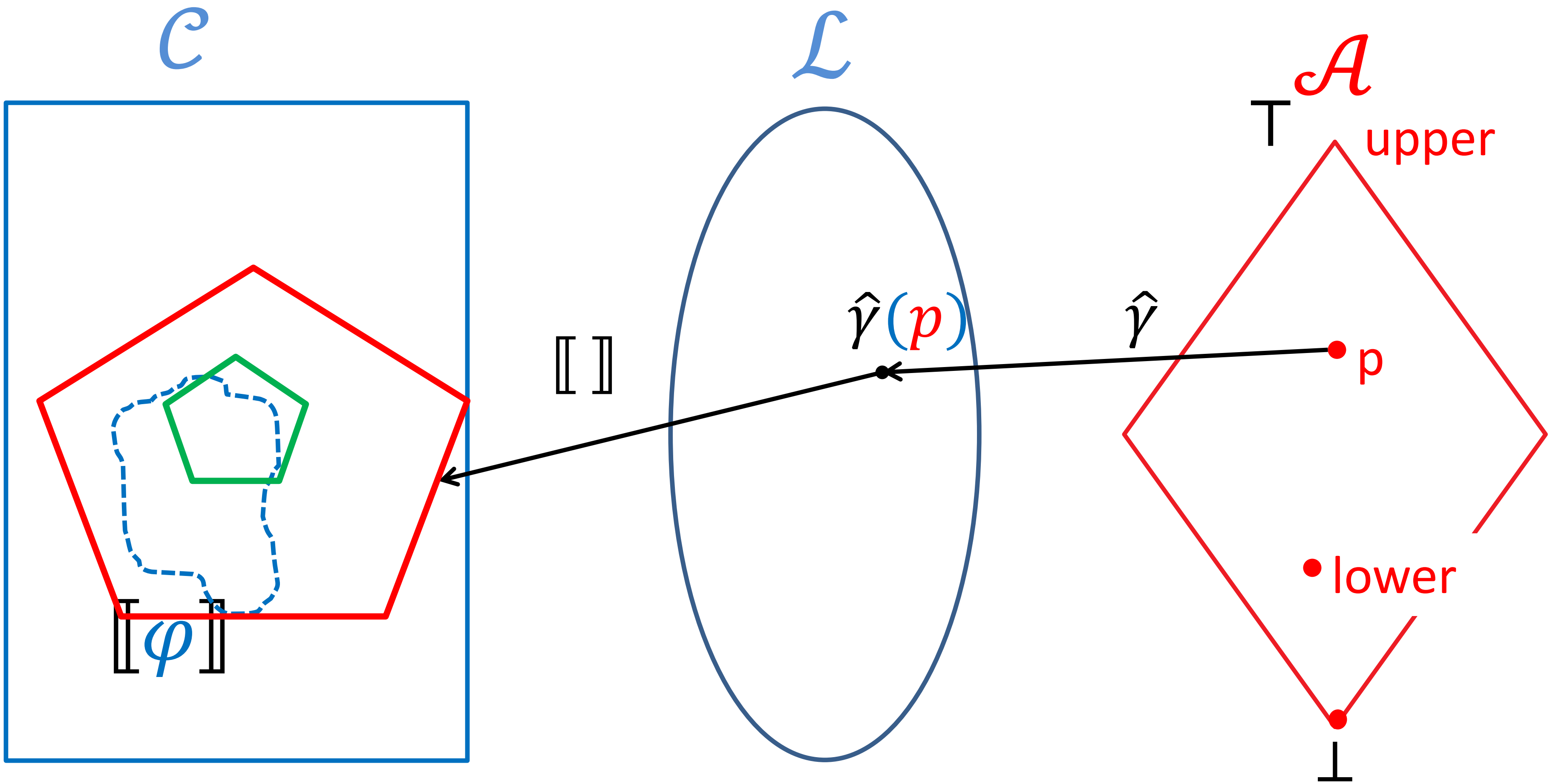
More time → more precision

Bilateral algorithm for $\hat{\alpha}(\varphi)$ [SAS'12]



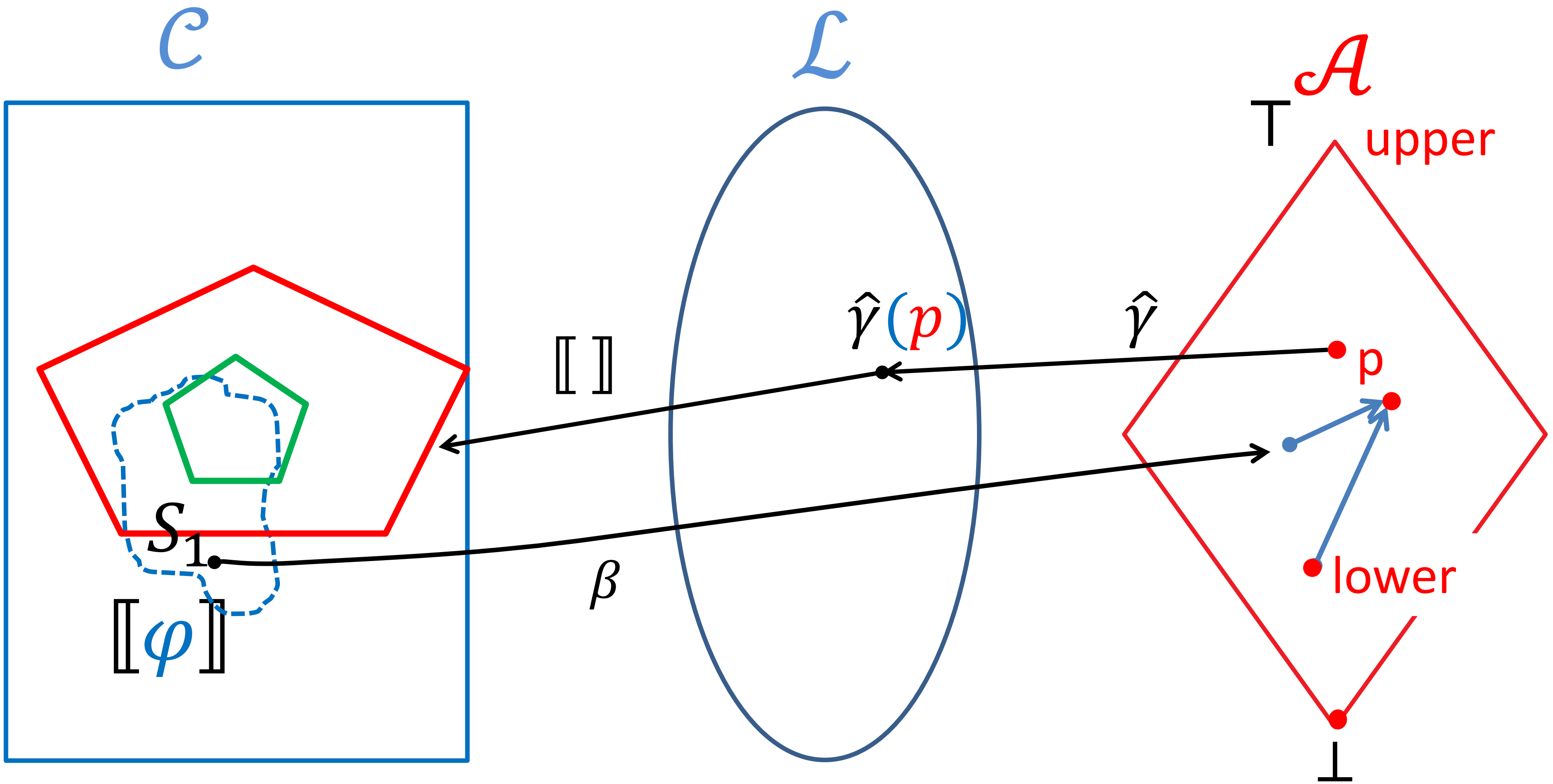
β: α for singleton set

Bilateral algorithm for $\hat{\alpha}(\varphi)$ [SAS'12]



$\varphi_1 = \varphi \wedge \neg\hat{\gamma}(\mathrm{p})$ UNSAT!

Bilateral algorithm for $\hat{\alpha}(\varphi)$ [SAS'12]



$\varphi_1 = \varphi \wedge \neg\hat{\gamma}(\mathrm{p})$ SAT!

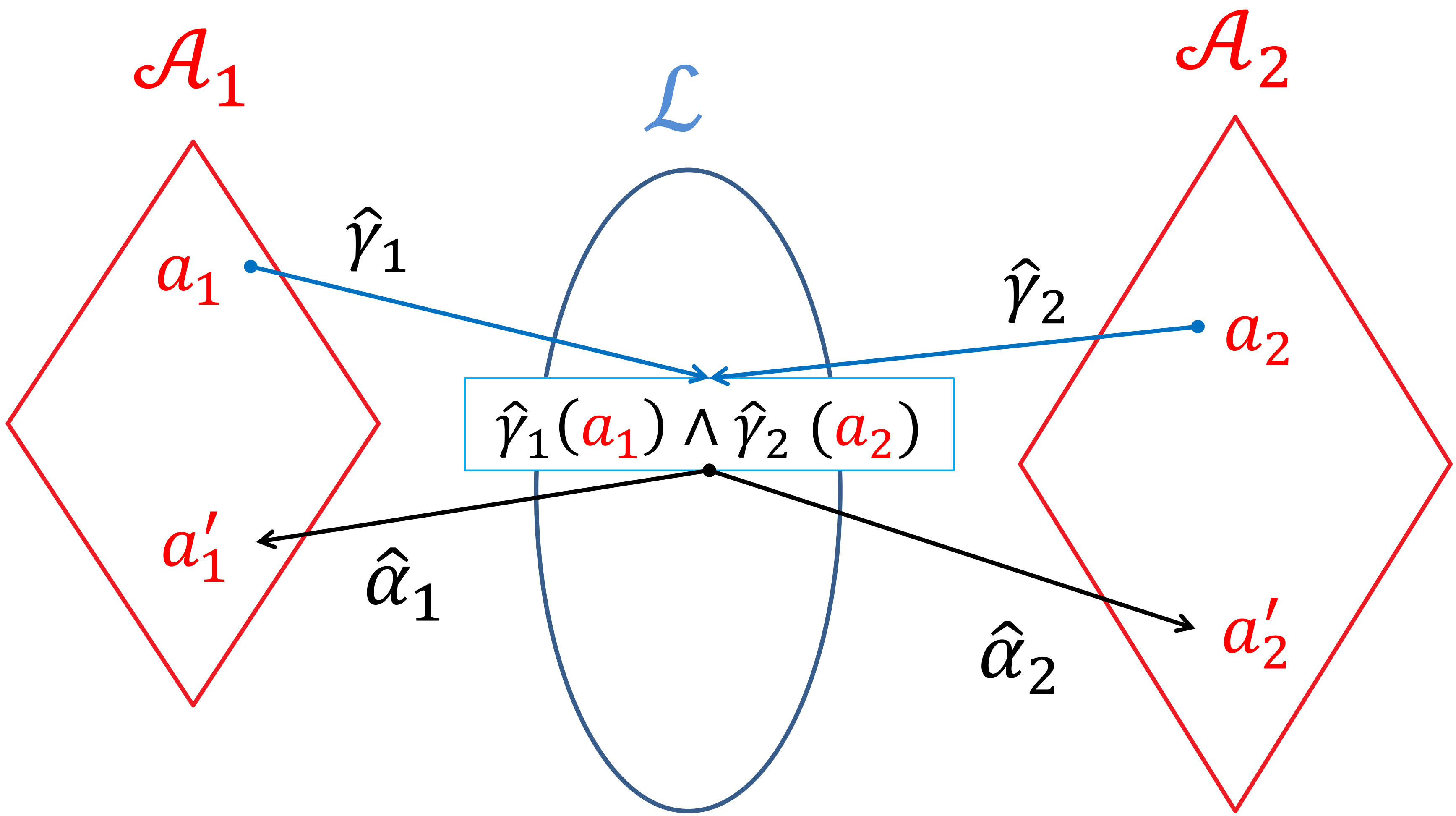
Symbolic abstraction ⇒ Best inductive invariants

- Theoretical limit of attainable precision
- Achieved via repeated application of best transformer
  - That's it! [TAPAS 2013]

Combination of domains

- Exchange of information among different domains during analysis
- More precision
  - "sum is greater than parts"
  - $x \geq 0, x\ odd$ reduces to $\boldsymbol{x > 0}, x\ odd$
- Enables heterogeneous ("fish-eye") analysis

Symbolic abstraction ⇒ information exchange



Summary

Symbolic abstraction increases level of automation, and ensures correctness when

- applying abstract transformers,
- computing best inductive invariants, and
- exchanging information among domains

Algorithms for symbolic abstraction require

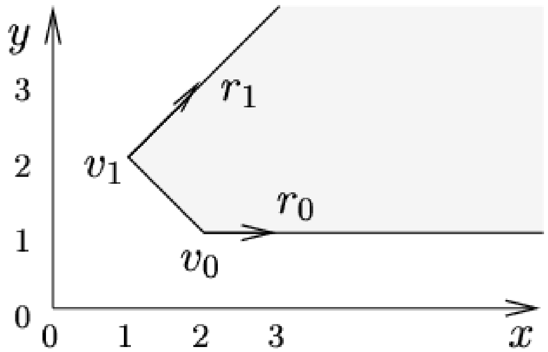
- off-the-shelf SMT solvers, and
- implementation of very few abstract-domain operations

Outline of Talk

- Review of goals
- Progress (Oct. 2012 - May 2013)
  - Component identification
    - Recovering class hierarchies using dynamic analysis
  - Verifying component properties
    - Symbolic abstraction (BET + ONR STTR)
    - Domain-combination technique: combine results from multiple analysis methods
    - Abstract domain of bit-vector inequalities
    - Format-compatibility checking (ONR)
  - Component extraction
    - Specialization slicing
    - Partial evaluation of machine code
- Recap of publications/submissions
- Recap of plans for 2013

Convex Polyhedra

[Figures from Halbwachs et al. FMSD97]



$$P = \left\{ (x, y) \mid \begin{pmatrix} y & \geq & 1 \\ x + y & \geq & 3 \\ -x + y & \leq & 1 \end{pmatrix} \right\}$$

$$V = \left\{ v_0 \begin{pmatrix} 2 \\ 1 \end{pmatrix}, v_1 \begin{pmatrix} 1 \\ 2 \end{pmatrix} \right\} \quad R = \left\{ r_0 \begin{pmatrix} 1 \\ 0 \end{pmatrix}, r_1 \begin{pmatrix} 1 \\ 1 \end{pmatrix} \right\}$$

Figure 1: A convex polyhedron and its 2 representations

Conjunctions of linear inequalities over rationals

$$a_1 x_1 + a_2 x_2 + \ldots + a_k x_k \leq c$$

Limitations of convex polyhedra

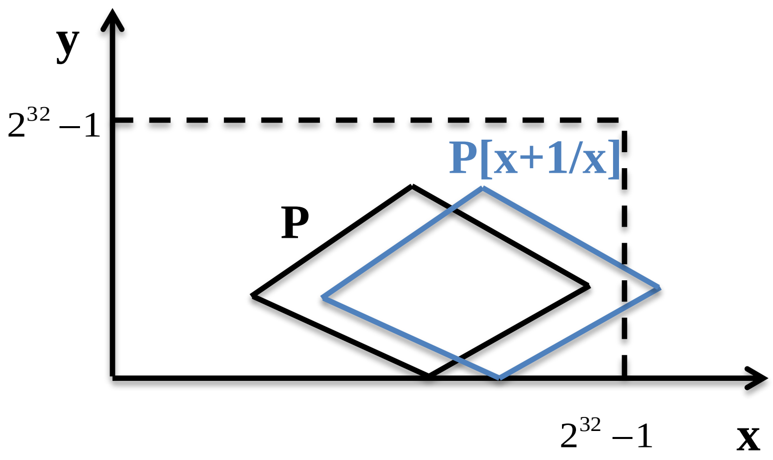
- Consider the following code fragment:

```
assume (0 <= low <= high ) ;
mid = ( low + high ) / 2 ;
assert (0 <= low <= mid <= high ) ;
```

- Polyhedral analysis unsoundly verifies that the assert holds.

$$\begin{array}{c} low = 1 \\ high = INT\_MAX \end{array} \Longrightarrow mid = INT\_MIN / 2$$

Limitations of convex polyhedra

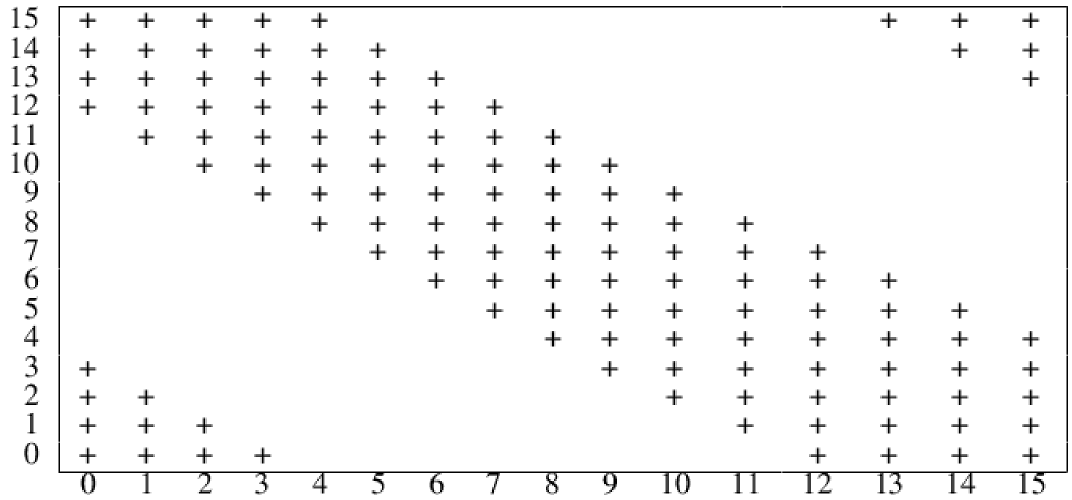


- Effect of the linear transformation might overflow
- Polyhedra expresses constraints over rational not bit-vector integers

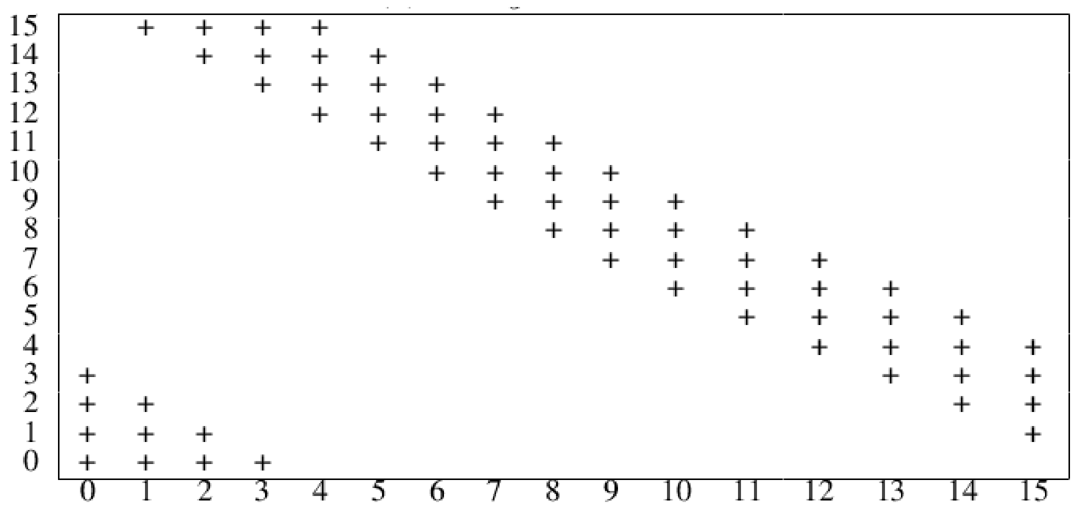
Problems with Polyhedra

- Unsound for machine arithmetic
  - machine integers wrap
  - mathematical integers do not
- Solution: Bit-Vector Inequality Domain

Bitvectors (Not so well-behaved . . .)



(a) $x + y + 4 \leq 7$



(b) $x + y \leq 3$

Key Idea!

- Split inequality into an equality and an interval by using a view variable

  For example, $a+b \le 5$ is changed to
  $a+b=s, s \in [0,5]$

- Examples on previous page:
  $x+y+4 \le 7$ and $x+y \le 3$ are represented as
  $x+y=s, s \in [-4,3]$ and $x+y=s, s \in [0,3]$
  respectively.

Bit-Vector Inequality Domain (BVI)

- Use a *Bit-Vector* equality domain for equalities (Ɛ) (King-Sondergaard 2010; Elder et al. 2011)
  - Ɛ is and equality-element over P ∪ S
- *Bit-Vector* Interval domain (*I*) on view variables
  - *I* is an interval-element over S
- P and S are the set of program and view variables, respectively

Bit-Vector Inequality Domain (BVI)

- S, the set of slack variables, is shared between Ɛ and *I*
- S acts as information exchange between the two domains
  - Example: $\lambda =< a-b=5 \wedge a+b=s, s \in [0,5] >$
    - Ɛ specifies the constraints $a-b=5$ and $a+b=s$
    - *I* specifies the constraints $s \in [0,5]$

View Variables

- View variables are defined by integrity constraints
- For example, in $\lambda, a + b = s$ is an integrity constraint

Symbolic Abstraction

- BVI is a combination of Ɛ and *I*
- Symbolic abstraction for Ɛ and *I* is available
- Information exchange is provided through common vocabulary S
- Symbolic abstraction for BVI is automatically available through $\hat{\alpha}(\varphi)$

Preliminary Results

- Setup: View constraints are of the form s = r, where r represents the 32-bit register in Machine Architecture (eg. ia32)
- BVI domain was 3.5 times slower than Bit-Vector equality domain
- BVI more precise than equality domain at 63% of the control points
- BVI's procedure summaries more precise than that of equality domain at 29% of the procedures

Heuristics

- Heuristics to choose view variables
- View constraints are of the form s = r are not sufficient

```
a=0; b=0;
for (i = 0; i < 100; i++)  {
    a++;
    if (i%2 == 0)
       b++;
}
```

Cannot get the constraint that $0 \leq 2b - a \leq 1$

Heuristics

- Linear expressions in branch predicates and assert statements
- “Invariants” produced by unsound analysis, eg polyhedra

Handling Memory

- Previous analysis only focused on registers
- Memory is treated as flat array in machine code
- Memory constraints represent memory views:

  v = mm[e], where

  v is the memory view,

  mm is the memory map,

  e is the address.

- **Memory domain:** Set of memory constraints

BVMI domain

- BVMI domain is capable of expressing Bit-Vector inequalities over memory variables
- BVMI components
  - Ɛ is an equality-domain element over P ∪ U ∪ S
  - *I* is an interval-domain element over S
  - *M* is an memory-domain element over U
- Information exchange happen between Ɛ and *I* through common variables S and between Ɛ and *M* through common variables U.

Current Status

- Implementation of BVI is completed
- Undergoing restructuring of code to utilize symbolic abstraction

Future Work

- Implementing heuristics for BVI and BVMI
- Integrating memory domain in the new framework

Recap

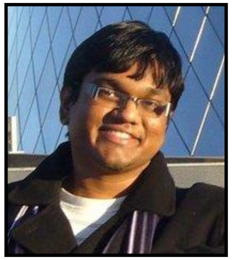
- Convex polyhedra doesn’t work for machine integers
- Bit-Vector Inequality Domain (BVI) handles Bit-Vector Inequalities by splitting them into Bit-Vector Equalities and Bit-Vector Intervals
- Memory Variables can be incorporated in a similar fashion by splitting them into Bit-Vector Equalities and Memory Constraints
- Information Exchange between the two domains happen through View Variables

Outline of Talk

- Review of goals
- Progress (Oct. 2012 - May 2013)
  - Component identification
    - Recovering class hierarchies using dynamic analysis
  - Verifying component properties
    - Symbolic abstraction (BET + ONR STTR)
    - Domain-combination technique: combine results from multiple analysis methods
    - Abstract domain of bit-vector inequalities
    - Format-compatibility checking (ONR)
  - Component extraction
    - Specialization slicing
    - Partial evaluation of machine code
- Recap of publications/submissions
- Recap of plans for 2013

Partial Evaluation for Machine-Code

- Slicing has limitations
  - limited semantic information – i.e., just dependence edges
  - no evaluation/simplification
- Partial evaluation: a framework for specializing programs
  - software specialization, optimization, etc.
- Binding-time analysis
  - what patterns are foo and bar called with?
    - e.g, { foo(S,S,D,D), foo(S,D,S,D), bar(S,D), bar(D,S) }
  - polyvariant binding-time analysis? specialized slicing!
- Design and implement an algorithm for partial evaluation of machine code



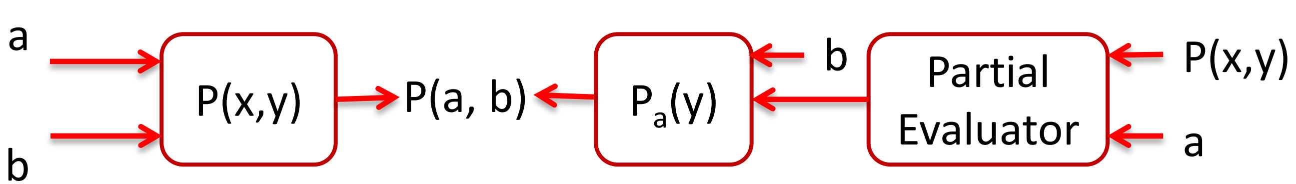
Venkatesh Srinivasan

Partial Evaluation of Machine code

- Given:
  - Machine-code procedure P(x, y)
  - Value “a” for x
- Goals:
  - Create a specialized procedure $P_a(y)$
  - If the value “b” is supplied for y, $P_a(y)$ computes P(a,b)

```
. . .
mov     dword [ebp - C],eax
. . .
mov     dword [ebp - 8],eax
mov     eax,dword [ebp - 8]
mov     edx,dword [ebp - C]
add     eax, edx
mov     dword [ebp - 4],eax
mov     eax,0
leave
ret
```

```
. . .
mov     dword [ebp - C],eax
mov     eax,dword [ebp - C]
add     eax, 2
mov     dword [ebp - 4],eax
mov     eax,0
leave
ret
```



Partial Evaluation – Why?

- Extraction of functional components
  - gzip executable has code that compresses and decompresses bundled together
  - Partial evaluation with ‘-c’ as the value of compress/decompress flag produces an executable that only compresses
- Binary specialization
  - Produces faster and smaller binaries optimized for a specific task
- Offline optimizer for unoptimized binaries
  - Partial evaluator performs optimizations such as constant propagation and constant folding, loop unrolling, elimination of unreachable/infeasible basic blocks, etc.

Methods

- Binding-time analysis
  - Classify instructions as:
    - Static – Instructions that only depend on inputs whose values are known at specialization time (can be evaluated at specialization time)
    - Dynamic – Instructions that are not static
- Specialization
  - Evaluate static instructions
  - Simplify dynamic instructions using partial static state
  - Emit residual code (simplified dynamic instructions)
  - Evaluate static jumps to eliminate entire basic blocks

Binding-Time Analysis

- Construct Program Dependence Graph (PDG) for binary
  - Using CodeSurfer/x86
- Add the instructions that initialize dynamic inputs' memory locations to the slicing criterion
- Compute an interprocedural forward slice
- Instructions included in the slice are dynamic instructions
- Remaining instructions are static (solely depend on static inputs)

Specialization

- Initialize static locations in program state to given values
- Worklist algorithm – <first basic block, initial state> is put in worklist
- Remove an item from worklist
- Static instructions
  - Evaluate and update state
- Dynamic instructions
  - Emit instructions that set up values for static hidden operands (for example, registers and flags)
  - Simplify dynamic instruction to use static values as immediate operands
  - Emit simplified instruction
  - Dynamic jumps – For each target basic block put <basic block, state> in worklist
  - If a <basic block, state> pair was already processed, do not put in worklist
- Keep processing until worklist is empty

Challenges

Outline of Talk

- Review of goals
- Progress (Oct. 2012 - May 2013)
  - Component identification
    - Recovering class hierarchies using dynamic analysis
  - Verifying component properties
    - Symbolic abstraction (BET + ONR STTR)
    - Domain-combination technique: combine results from multiple analysis methods
    - Abstract domain of bit-vector inequalities
    - Format-compatibility checking (ONR)
  - Component extraction
    - Specialization slicing
    - Partial evaluation of machine code
- Recap of publications/submissions
- Recap of plans for 2013

Recap of publications/submissions

1. Lim, J. and Reps. T., TSL: A system for generating abstract interpreters and its application to machine-code analysis. To appear in ACM Trans. on Program. Lang. and Syst. (TOPLAS), April 2013. http://www.cs.wisc.edu/wpis/papers/toplas13-tsl-final.pdf

2. Srinivasan, V.K. and Reps, T., Software-architecture recovery from machine code. TR-1781, Computer Sciences Department, University of Wisconsin, Madison, WI, March 2013. Submitted for conference publication. http://www.cs.wisc.edu/wpis/papers/tr1781.pdf

3. Aung, M., Horwitz, S., Joiner, R., and Reps, T., Specialization slicing. TR-1776, Computer Sciences Department, University of Wisconsin, Madison, WI, October 2012. Submitted for journal publication. http://www.cs.wisc.edu/wpis/papers/SpecSlicing-submission.pdf

4. Thakur, A., Lal, A., Lim, J., and Reps, T., PostHat and all that: Attaining most-precise inductive invariants. To appear in 4th Workshop on Tools for Automatic Program Analysis, June 2013. TR-1790, Computer Sciences Department, University of Wisconsin, Madison, WI, April 2013. http://www.cs.wisc.edu/wpis/papers/tr1790.pdf

5. Sharma, T., Thakur, A., and Reps, T., An abstract domain for bit-vector inequalities. TR-1789, Computer Sciences Department, University of Wisconsin, Madison, WI, April 2013. http://www.cs.wisc.edu/wpis/papers/tr1789.pdf

Recap of plans for 2013

- Component identification
  - object traces $\rightarrow$ class hierarchies
- Component extraction
  - partial evaluator for machine code
- Verifying component properties
  - $\tilde{\alpha}^{\downarrow}$
    - separation logic
    - WALi-based and Boogie-based invariant finding
  - bitvector-inequality domain
  - Stretched-TreeIC3

Outline of Talk

- Review of goals
- Progress (Oct. 2012 - May 2013)
  - Component identification
    - Recovering class hierarchies using dynamic analysis
  - Verifying component properties
    - Symbolic abstraction (BET + ONR STTR)
    - Domain-combination technique: combine results from multiple analysis methods
    - Abstract domain of bit-vector inequalities
    - Format-compatibility checking (ONR)
  - Component extraction
    - Specialization slicing
    - Partial evaluation of machine code
- Recap of publications/submissions
- Recap of plans for 2013

Specialization Slicing

- Problem statement
  - Ordinary "closure slices" can have mismatches between call sites and called procedures
    - different call sites have different subsets of the parameters
  - Idea: specialize the called procedures
  - Challenge: find a minimal solution (minimal duplication)

Min Aung

Specialization Slicing

```
int g1, g2, g3;

void p(int a, int b) {
  g1 = a;
  g2 = b;
  g3 = g2;
}





int main() {
  g2 = 100;
  p(g2, 2);
  p(g2, 3);
  p(4, g1+g2);
  printf("%d", g2);
}
```

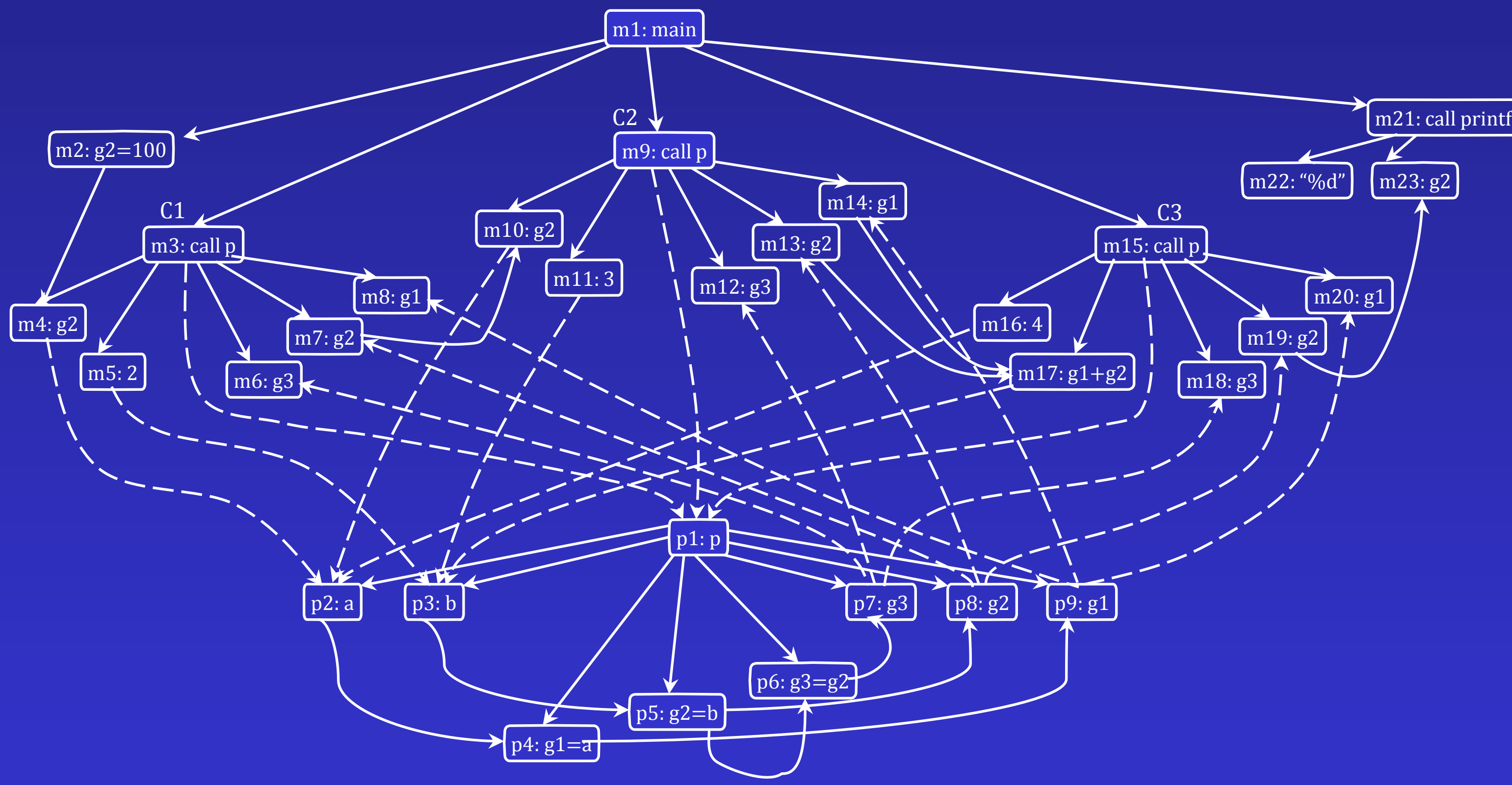
```
int g1, g2;

void p(int a, int b) {
  g1 = a;
  g2 = b;

}






int main() {

  p(     2);
  p(g2, 3);
  p(   g1+g2);
  printf("%d", g2);
}
```

Closure slice

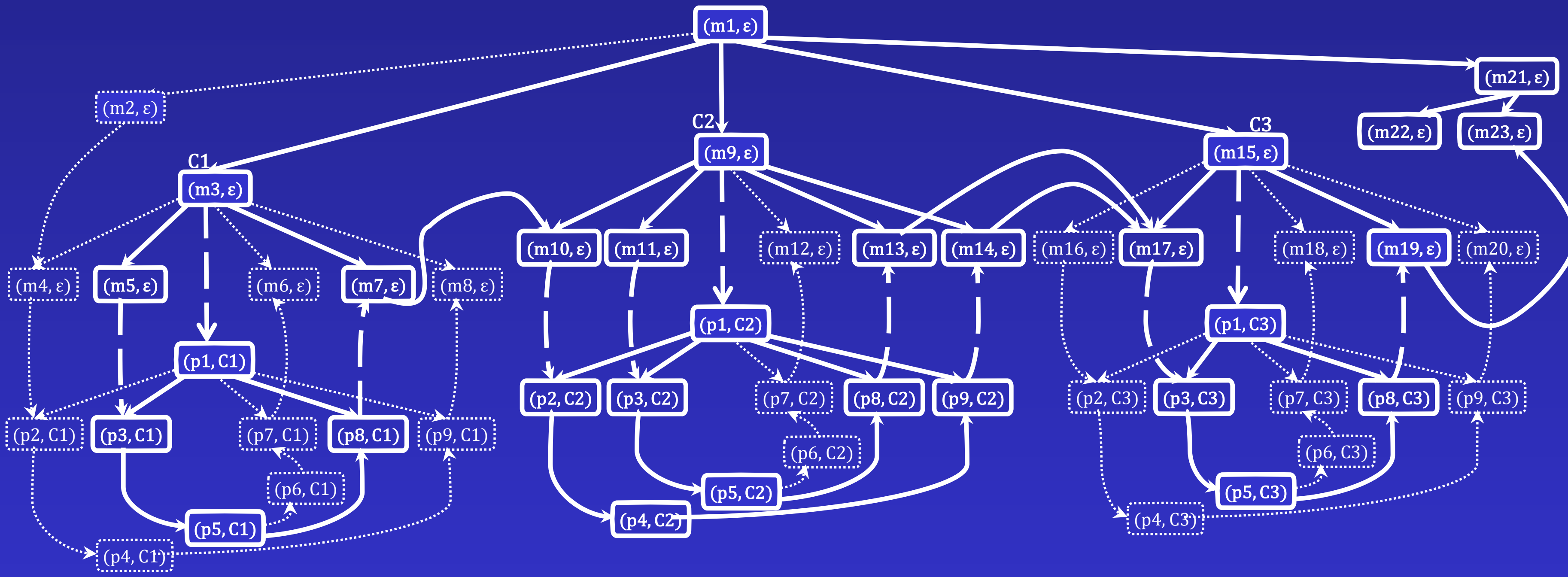
```
int g1, g2;

void p1(int b) {
  g2 = b;
 }

void p2(int a, int b) {
  g1 = a;
  g2 = b;
 }

int main() {

  p1(2);
  p2(g2, 3);
  p1(g1+g2);
  printf("%d", g2);
}
```

Specialized slice

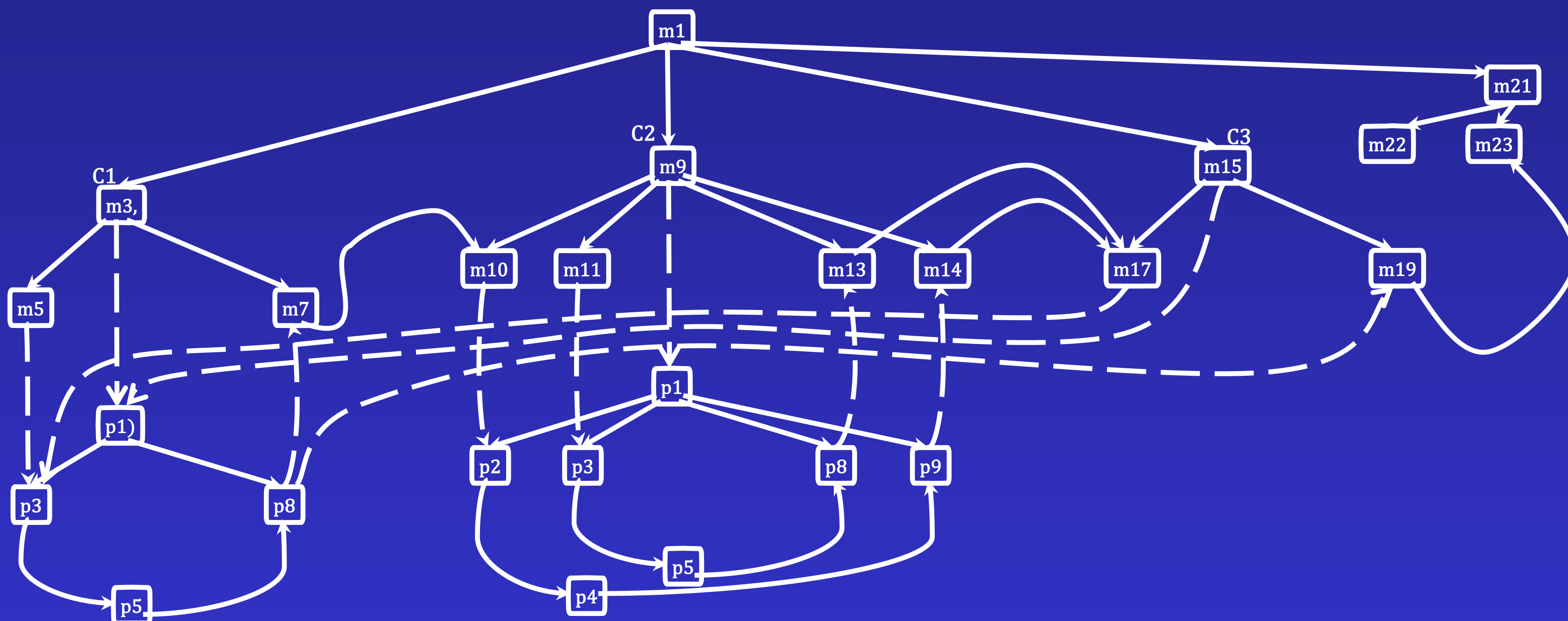
System Dependence Graph (SDG)



Unrolled SDG



Specialized SDG



Specialization slice of a recursive program

```
(1)  int g1, g2;
(2)
(3)  void s(int a,
(4)         int b){
(5)
(6)    g1 = b;
(7)    g2 = a;
(8)  }
(9)
(10)
(11)
(12) int r(int k) {
(13)
(14)   if (k > 0) {
(15)     s(g1, g2);
(16)     r(k-1);
(17)     s(g1, g2);
(18)   }
(19)
(20) }
(21)
(22)
(23)
(24) int main() {
(25)   g1 = 1;
(26)   g2 = 2;
(27)   r(3);
(28)   printf("%d\n", g1);
(29) }
```

```
int g1, g2;

void s_1(int b) {
  g1 = b;
}
void s_2(int a) {
  g2 = a;
}

void r_1(int k) {
  if (k > 0) {
    s_2(g1);
    r_2(k-1);
    s_1(g2);
  }
}
void r_2(int k) {
  if (k > 0) {
    s_1(g2);
    r_1(k-1);
    s_2(g1);
  }
}

int main() {
  g1 = 1;
  r_1(3);
  printf("%d\n", g1);
}
```

Calling pattern:
(27) ((16)(16))*

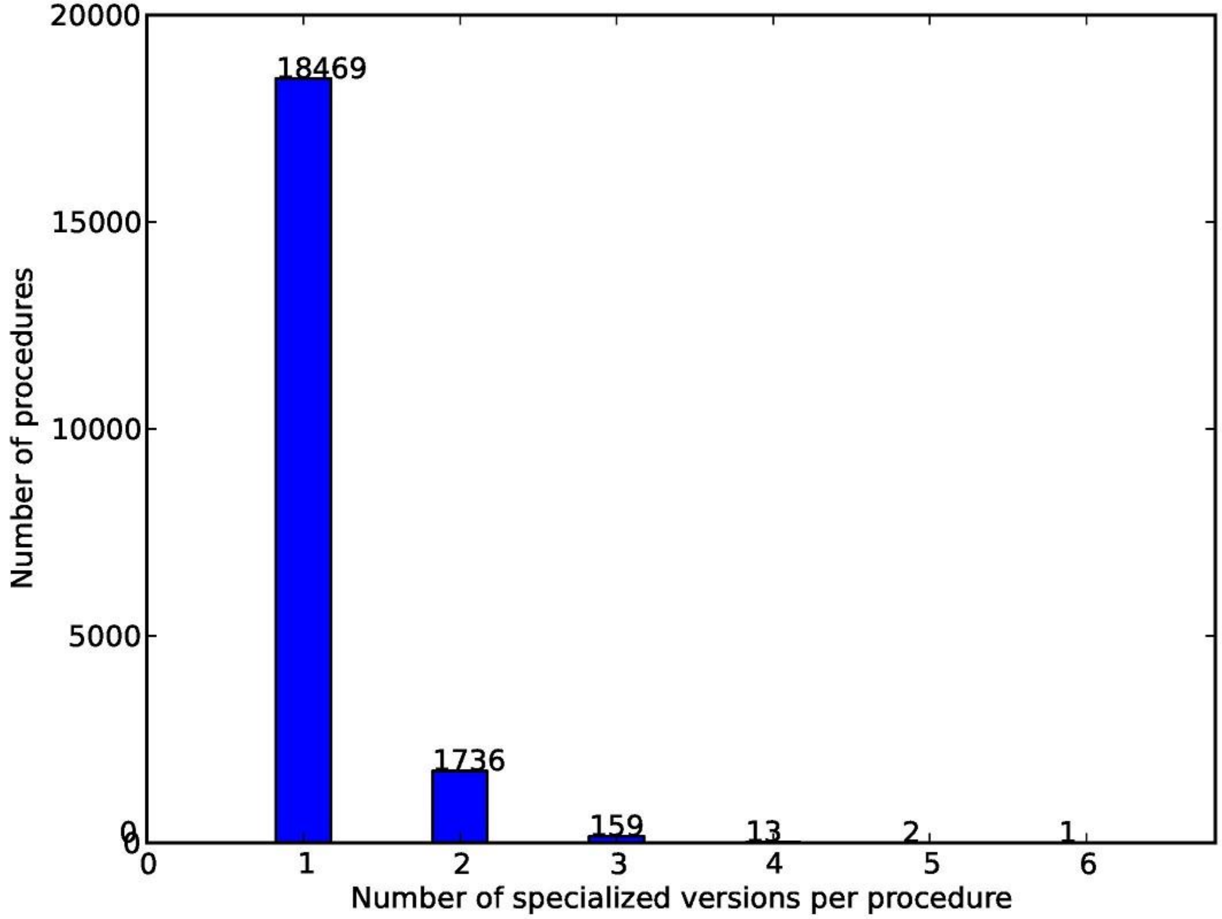
Calling pattern:
(27)(16)((16)(16))*

Specialization Slicing

- Problem statement
  - Ordinary “closure slices” can have mismatches between call sites and called procedures
    - different call sites have different subsets of the parameters
  - Idea: specialize the called procedures
  - Challenge: find a minimal solution (minimal duplication)

1. In the worst case, specialization causes an exponential increase in size
2. In practice, observed a 9.4% increase

Relatively Few Specialized Procedures



Specialization Slicing

- Problem statement
  - Ordinary “closure slices” can have mismatches between call sites and called procedures
    - different call sites have different subsets of the parameters
  - Idea: specialize the called procedures
  - Challenge: find a minimal solution (minimal duplication)
- Key insight
  - minimal solution involves solving a partitioning problem on a certain infinite graph
  - problem solvable using PDSs: all node-sets in infinite graph can be represented via FSMs
  - algorithm: a few automata-theoretic operations

Algorit…

Input: SDG S and slicing criterion C
Output: An SDG R for the specialized slice … with respect to C

// Create A_6, a minimal reverse-deterministic automaton for the
// stack-configuration slice of S with respect to C

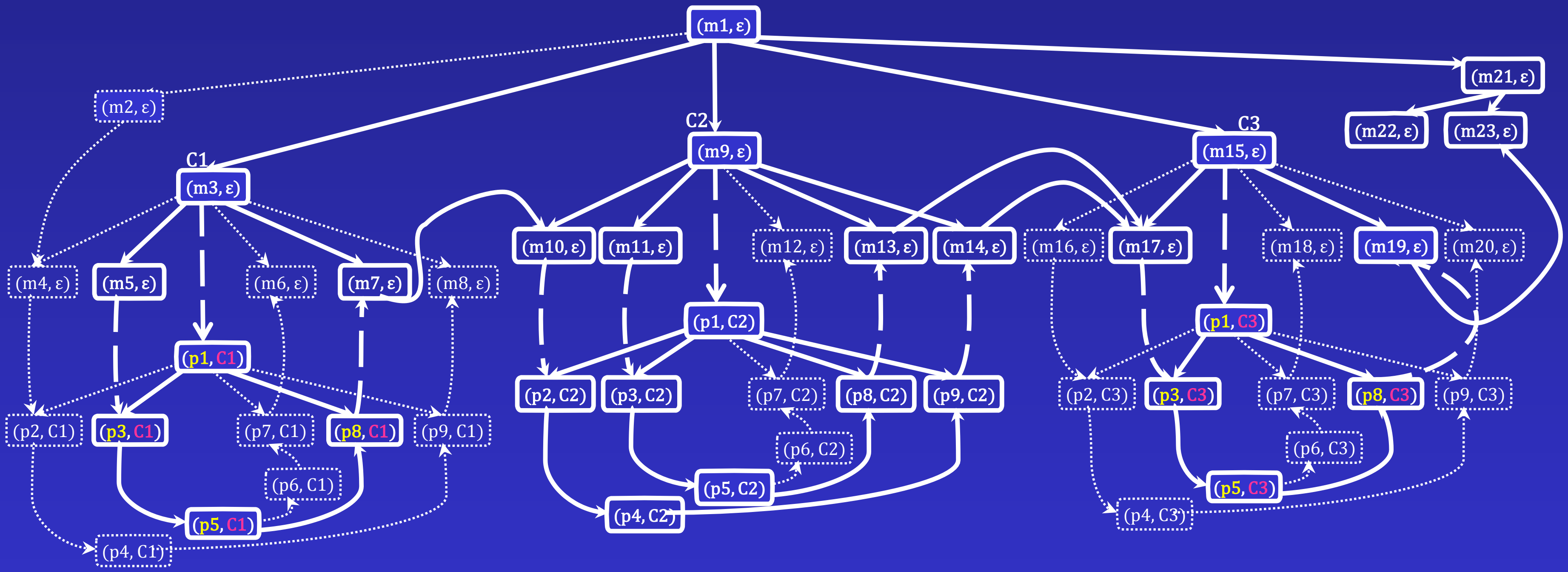
1 P_S = the PDS for S
2 A_0 = a P_S-automaton that accepts C
3 A_1 = *Prestar*[P_S](A_0)
4 A_2 = reverse(A_1)
5 A_3 = determinize(A_2)
6 A_4 = minimize(A_3)
7 A_5 = reverse(A_4)
8 A_6 = removeEpsilonTransitions(A_5)

// Read out SDG R from A_6
...

Partition obtained by determinizing and minimizing:
Each state = set of calling contexts for one specialized procedure

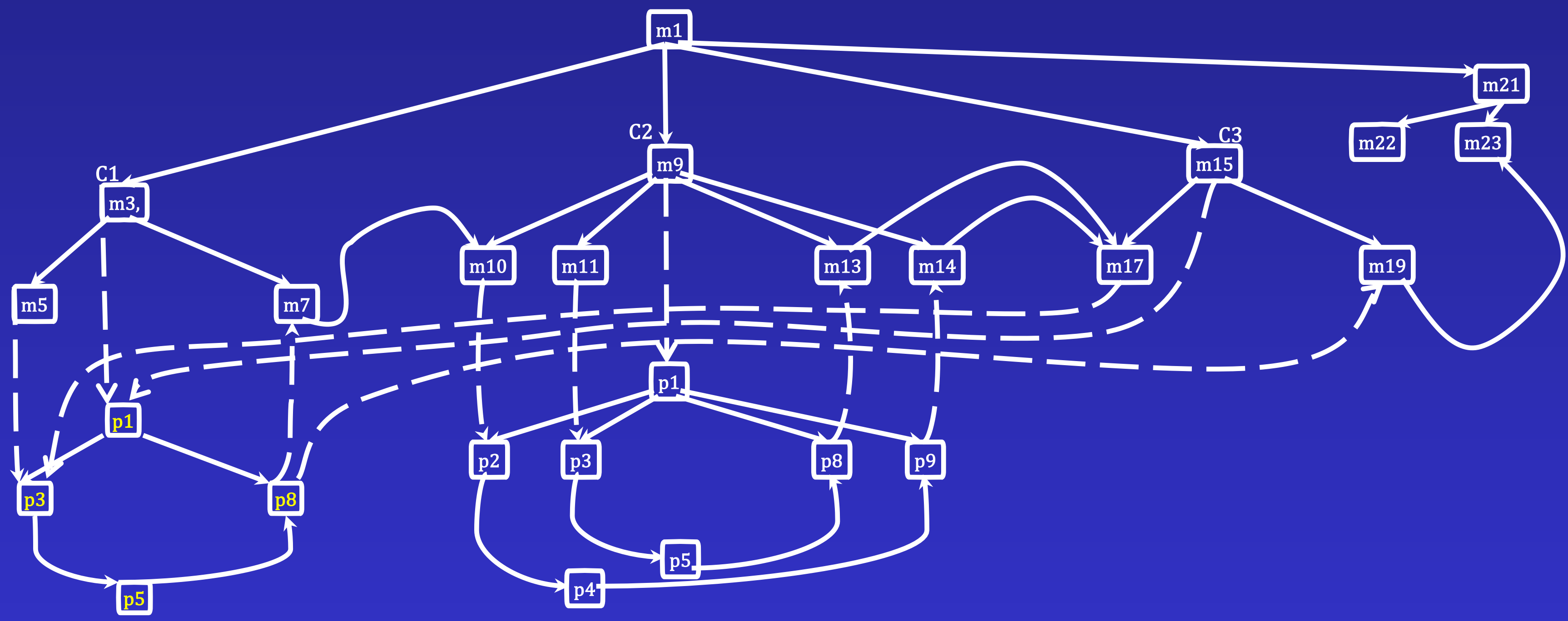
Automata used to hold onto sets of points in possibly infinite graph

Unrolled SDG



Each yellow name has the same set of stack configurations {C1,C3}
Such sets are infinite for recursive programs => FSMs

Specialized SDG



Each yellow name has the same set of stack configurations {C1,C3}
Such sets are infinite for recursive programs => FSMs

Feature Removal

```
int add(int a,int b) {
    q: return a+b;
}

int mult(int a,int b) {
    int i = 0;
    int ans = 0;
    while(i < a) {
        c5: ans = add(ans,b);
        c6: i = add(i,1);
    }
    return ans;
}

void tally
(int& sum,int& prod,int N) {
    int i = 1;
    while(i <= N) {
        c2: sum = add(sum,i);
        c3: prod = mult(prod,i);
        c4: i = add(i,1);
    }
}

int main() {
    int sum = 0;
    int prod = 1;
    c1: tally(sum,prod,10);
    printf("%d ",sum);
    printf("%d ",prod);
}
```

```
int add(int a,int b) {
    q: return a+b;
}

int mult(      int b) {
    int i = 0;
    int ans = 0;




    return;
}

void tally
(int& sum,       int N) {
    int i = 1;
    while(i <= N) {
        c2: sum = add(sum,i);
        c3:       mult(    i);
        c4: i = add(i,1);
    }
}

int main() {
    int sum = 0;

    c1: tally(sum,     10);
    printf("%d ",sum);

}
```

Feature Removal

```
int g1, g2, g3;

void p(int a, int b) {
  g1 = a;
  g2 = b;
  g3 = g2;
}





int main() {
  g2 = 100;
  p(g2, 2);
  p(g2, 3);
  p(4, g1+g2);
  printf("%d", g2);
}
```

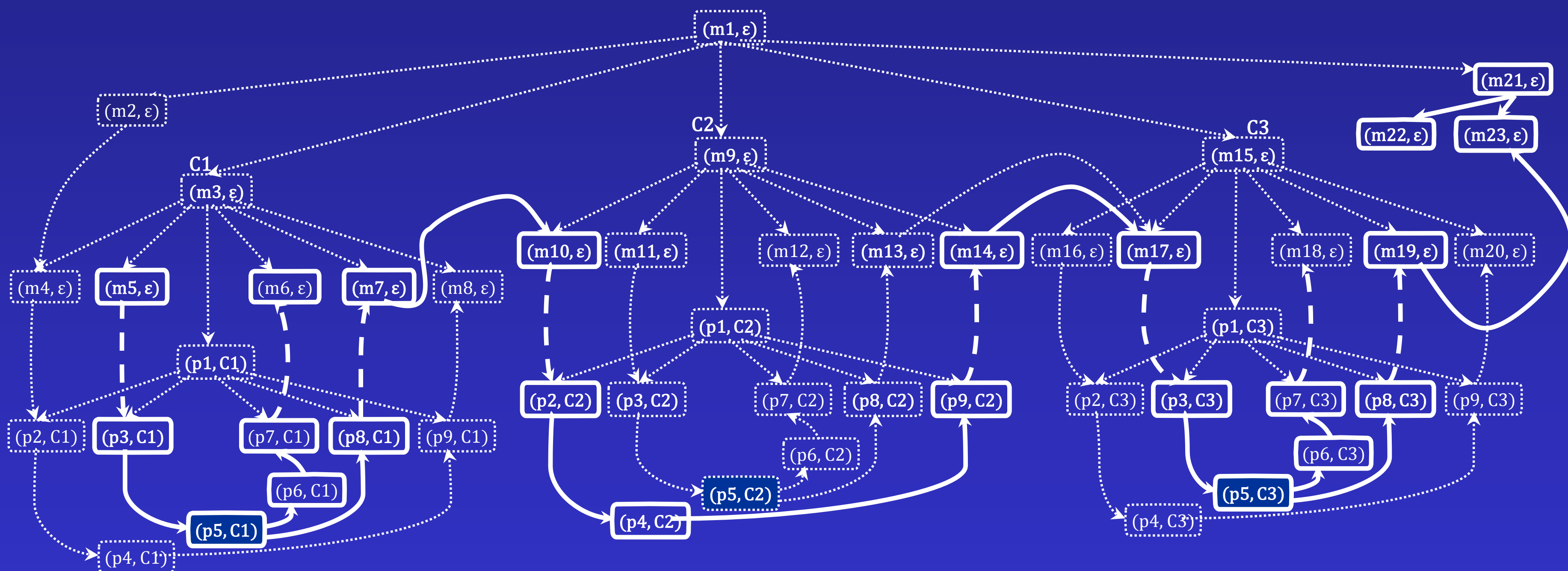
```
int g1, g2, g3;

void p(int a, int b) {
  g1 = a;
  g2 = b;
  g3 = g2;
}





int main() {
  g2 = 100;
  p(g2, 2);
  p(g2, 3);
  p(4, g1+g2);
  printf("%d", g2);
}
```

Forward closure slice

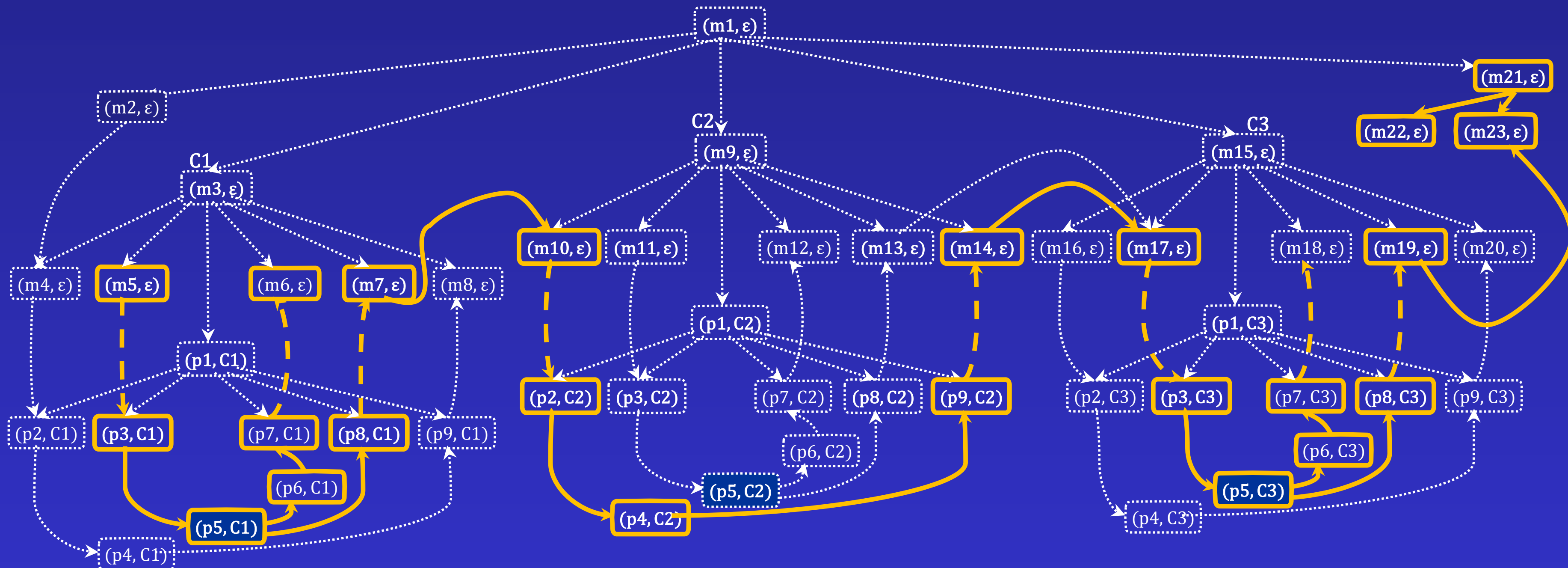
```
int g1, g2;

void p1(int a) {
  g1 = a;
 }

void p2(int b) {
  g2 = b;
  g3 = g2;
 }

int main() {
  g2 = 100;
  p1(g2);
  p2(3);
  p1(4);

}
```

Specialized slice

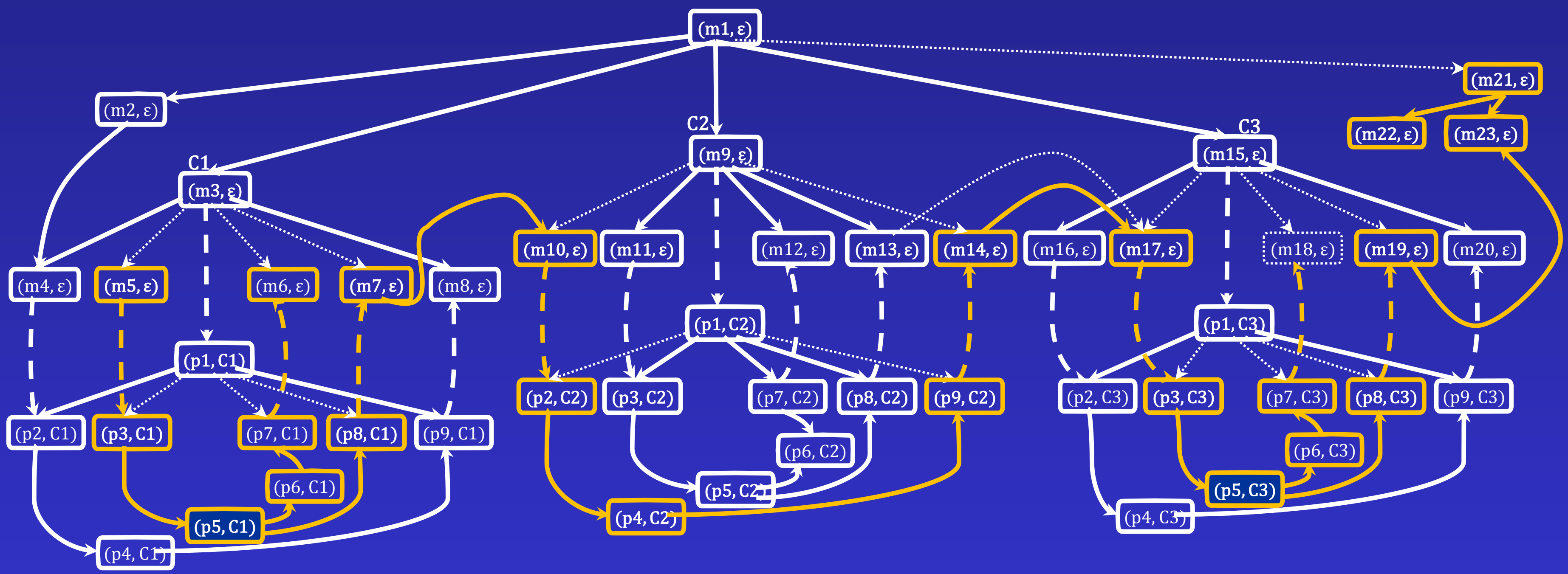
Unrolled SDG



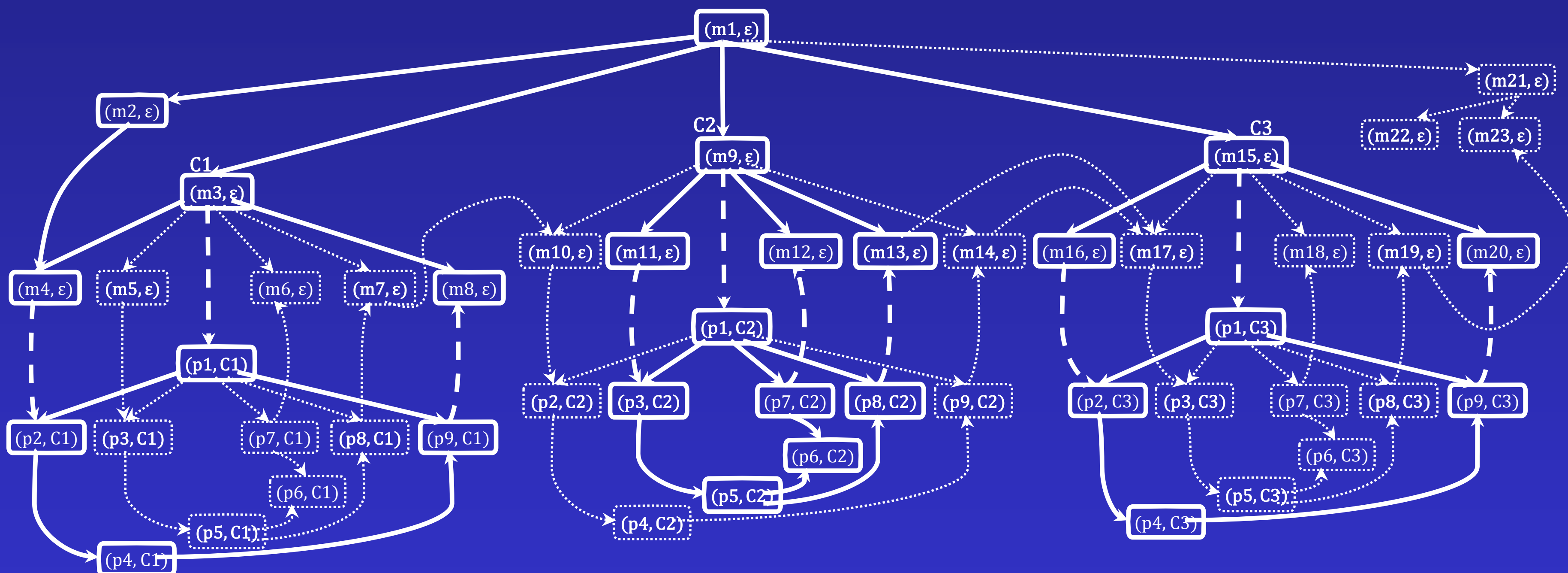
Complemented Unrolled SDG



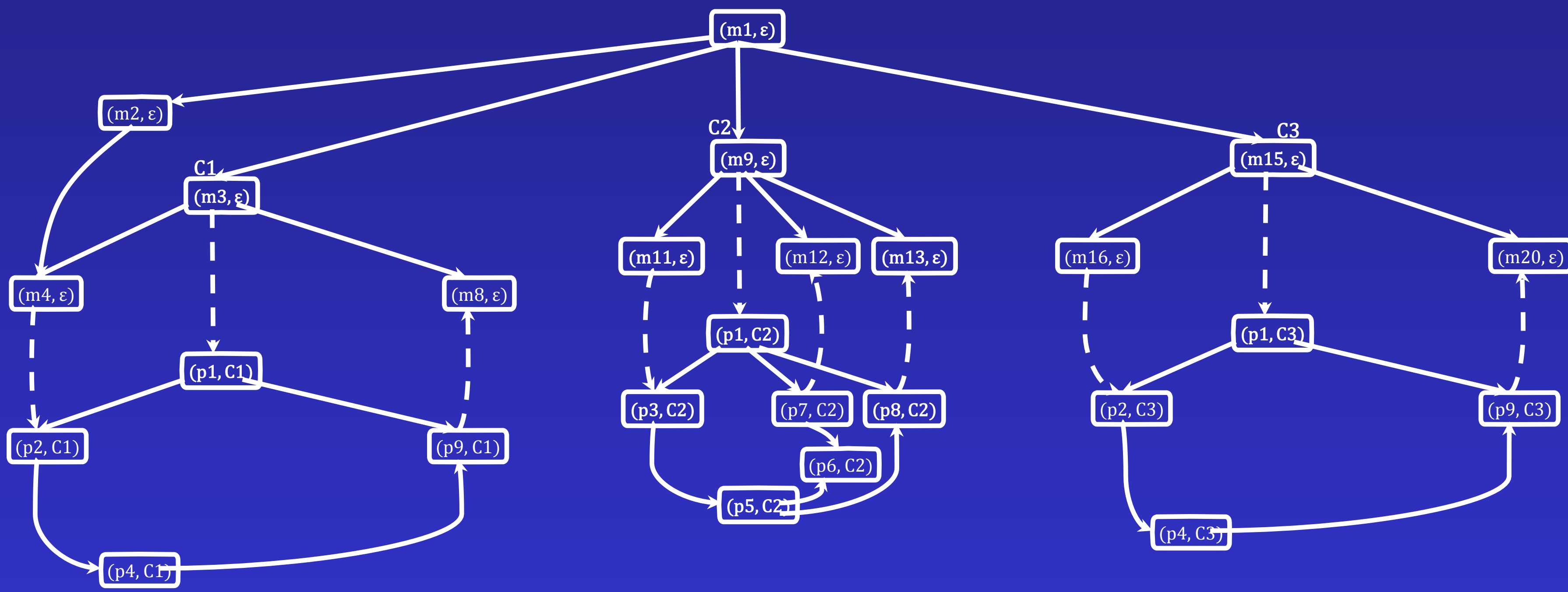
Complemented Unrolled SDG



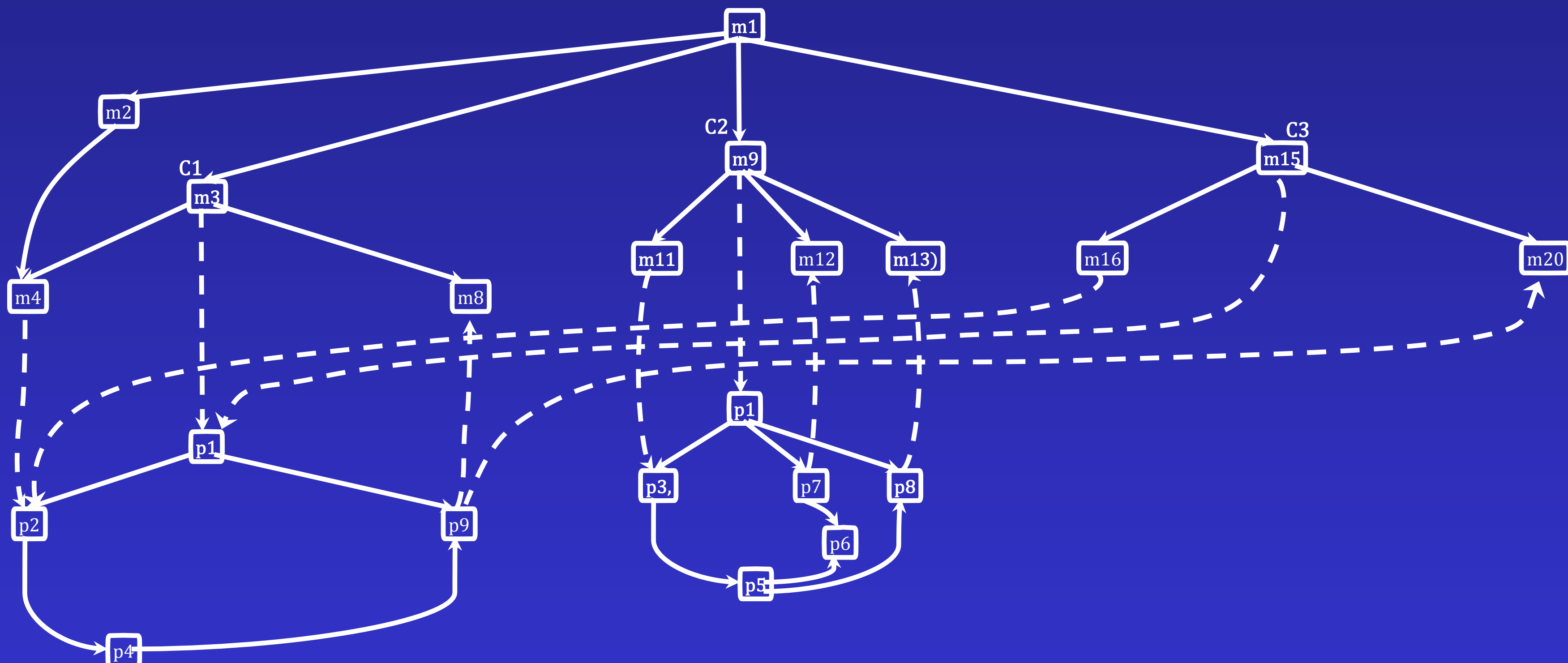
Complemented Unrolled SDG



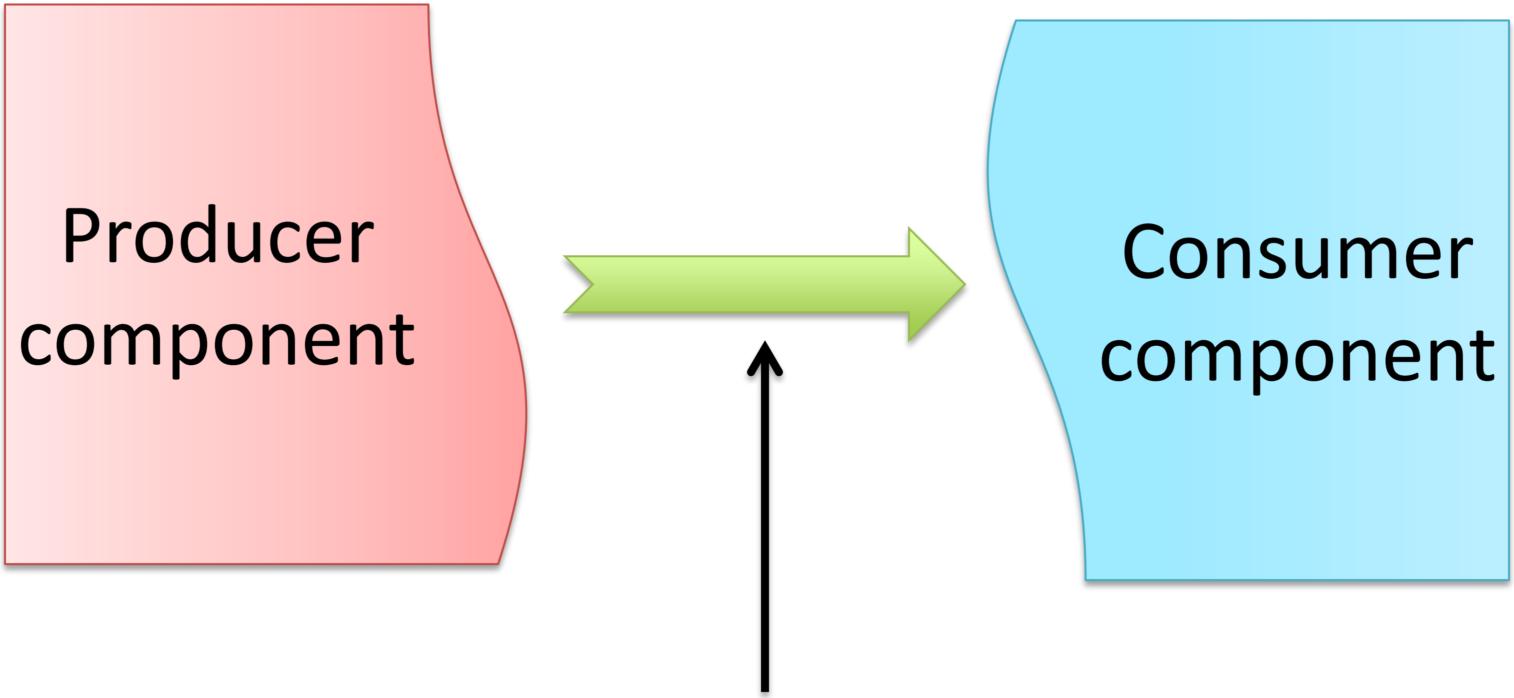
Complemented Unrolled SDG



Complemented Unrolled SDG



Goal: check format compatibility

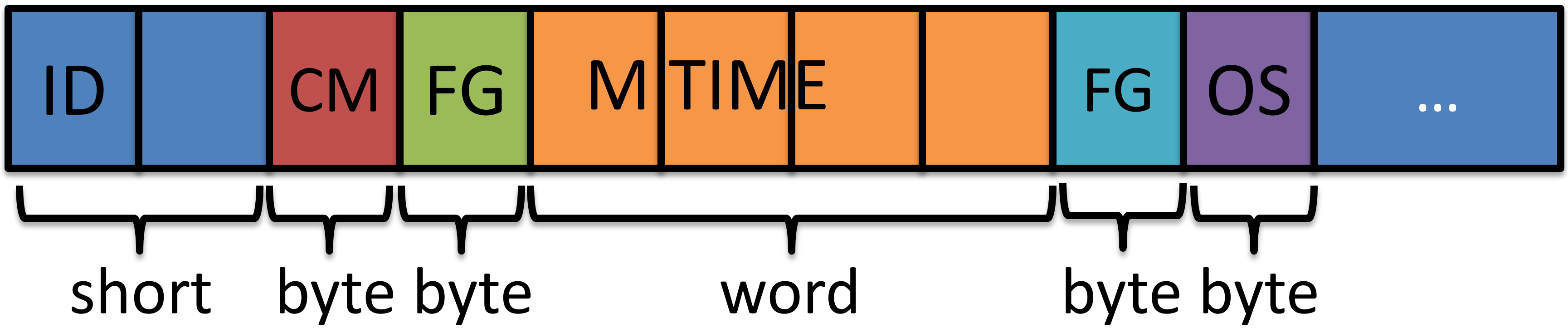


1. Infer output format
2. Infer accepted format
3. Check compatibility

Evan Driscoll

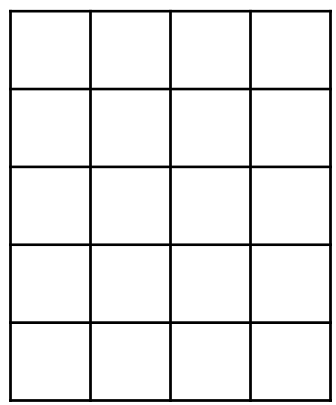
Formats are strings over "types"

Header of gzip format:



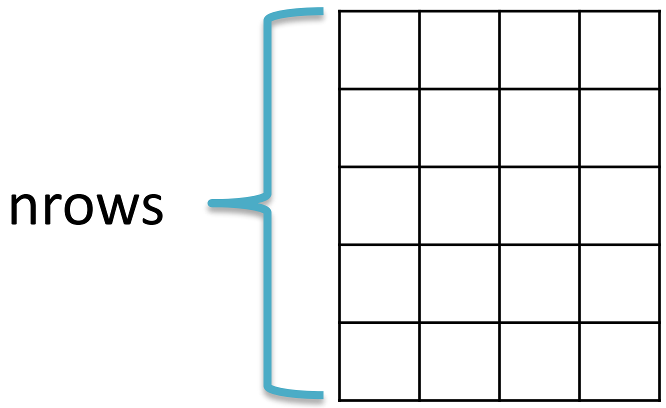
Current work: enhance format spec

nrows ncols pix11 pix12 pix13 pix14 pix21 pix22 pix23 …



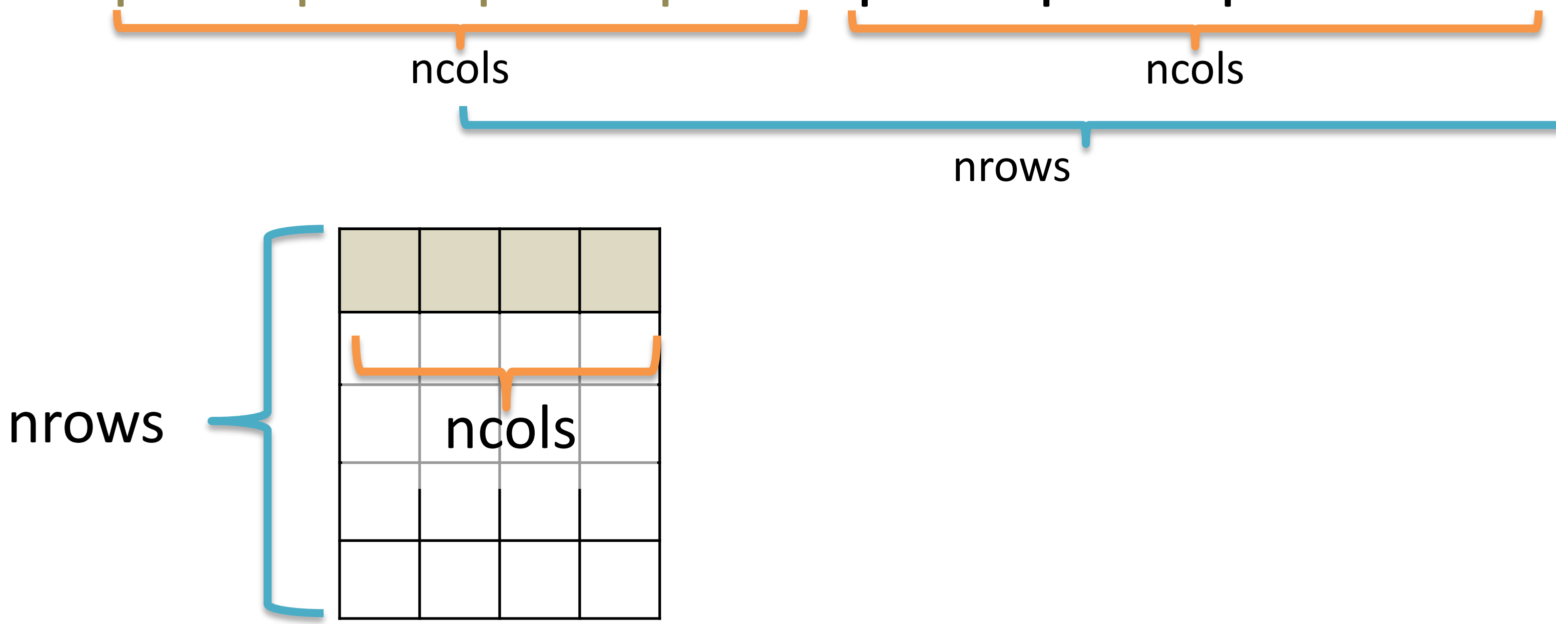
Current work: enhance format spec

nrows ncols pix11 pix12 pix13 pix14 pix21 pix22 pix23 …



Current work: enhance format spec

nrows ncols pix11 pix12 pix13 pix14 pix21 pix22 pix23 …



Current work: enhance format spec

nrows ncols pix11 pix12 pix13 pix14 pix21 pix22 pix23 …

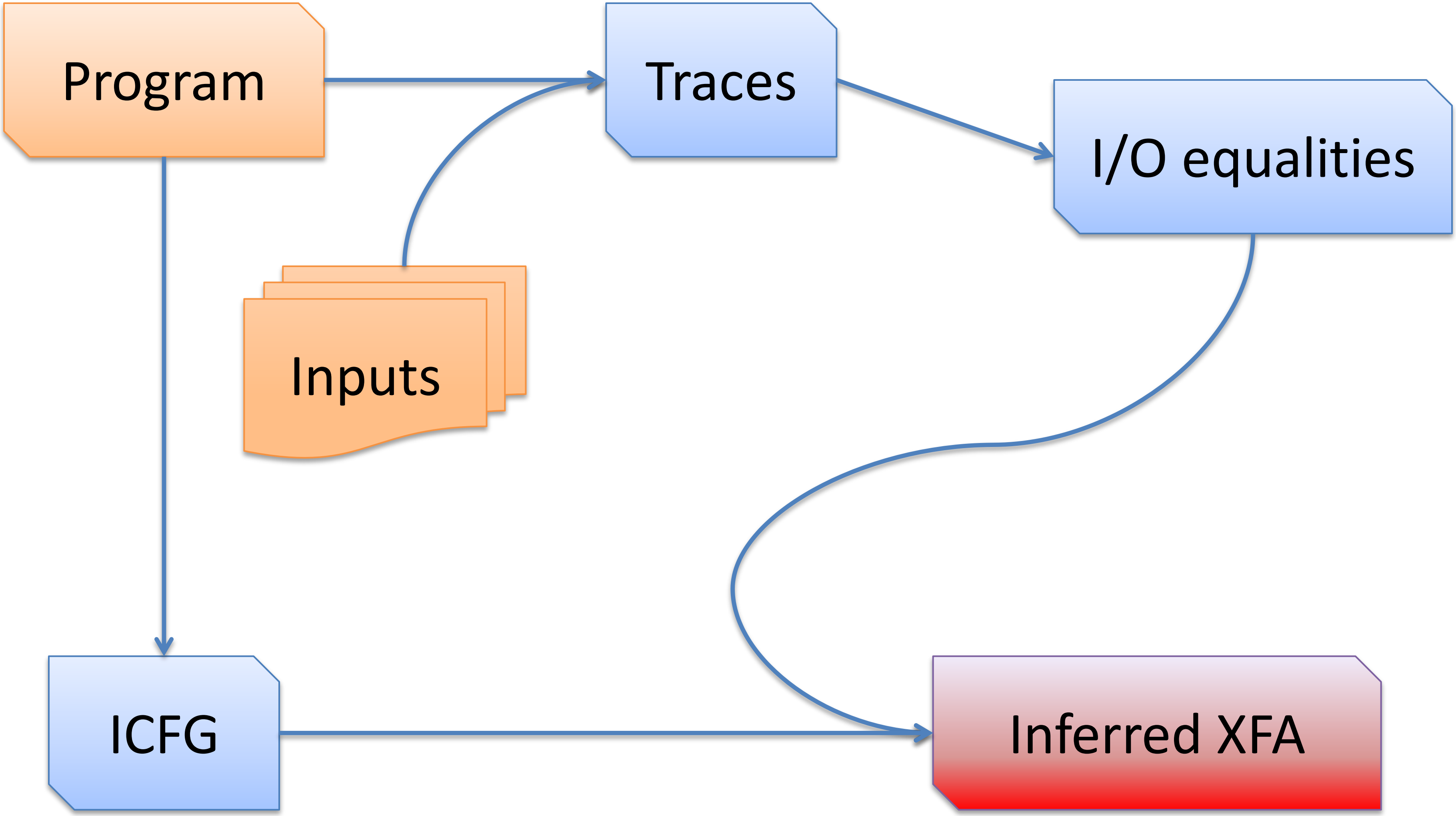
ncols ncols

nrows

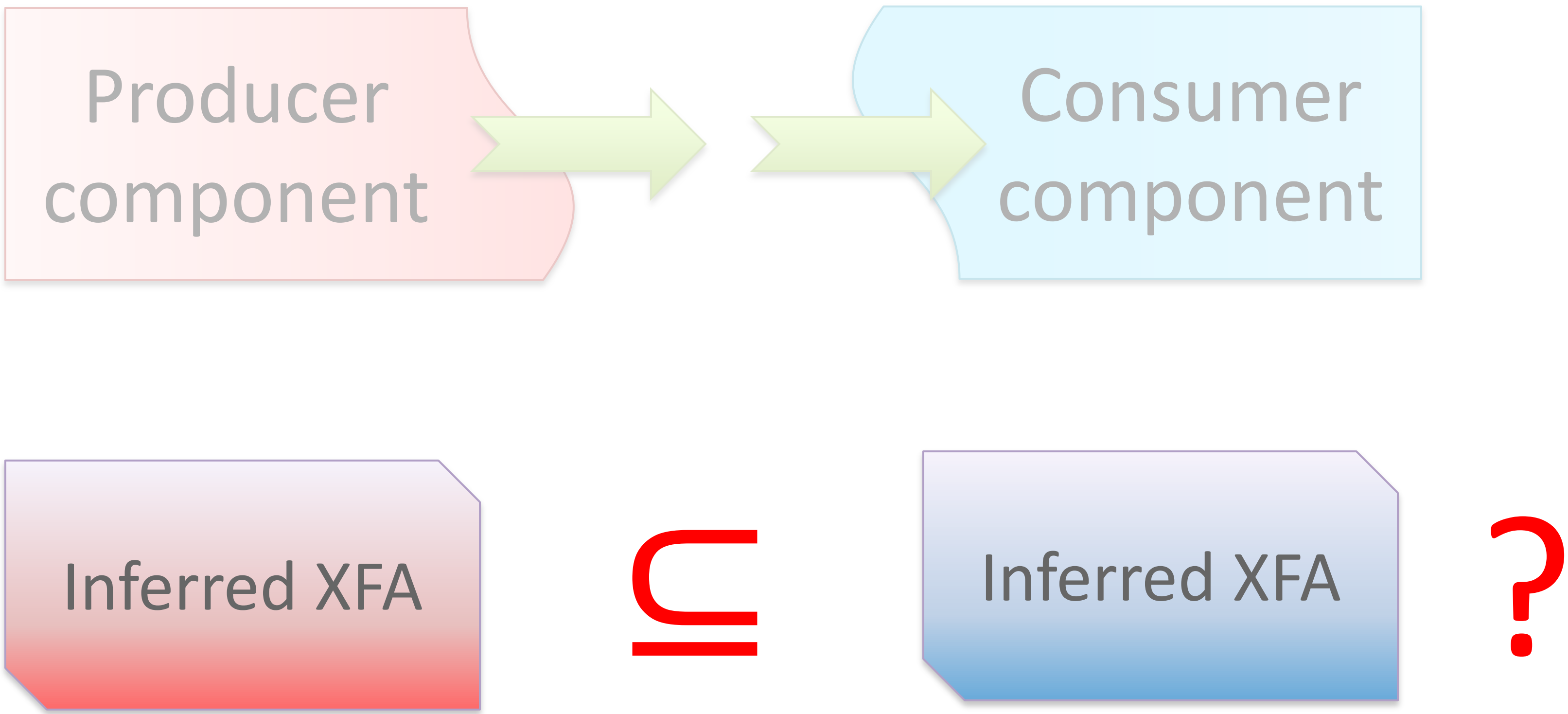
Infer an automaton equivalent to:

$\text{nrows:int ncols:int } ((\text{byte byte byte byte})^{\text{ncols}})^{\text{nrows}}$

Roadmap: Inference



Roadmap: Compatibility



Status

Prototype essentially done, but not well-tested. Working on performance and on finding tests.

How we do it

nrows:int ncols:int $((\text{byte byte byte})^*)^*$

Exponents start as standard Kleene *, and correspond to program loops

How we do it

nrows:int ncols:int ((byte byte byte)*)*

We instrument loops with *trip counts*
We instrument I/O calls to remember values

How we do it

nrows:int ncols:int ((byte byte byte)*)*

remembered I/O value

trip count

We instrument loops with *trip counts*

We instrument I/O calls to remember values

When two of these are found to always equal,
replace the * with an exponent

How we do it

nrows:int ncols:int ((byte byte byte)*)^{nrows}

↑ remembered I/O value ↑ trip count

We instrument loops with *trip counts*

We instrument I/O calls to remember values

When two of these are found to always equal,
replace the * with an exponent

How we do it

nrows:int ncols:int ((byte byte byte)*)^{nrows}

We instrument loops with *trip counts*

We instrument I/O calls to remember values

When two of these are found to always equal,
replace the * with an exponent

How we do it

nrows:int ncols:int ((byte byte byte)^{ncols})^{nrows}

We instrument loops with *trip counts*

We instrument I/O calls to remember values

When two of these are found to always equal, replace the * with an exponent

We use Daikon

Daikon identifies *dynamic* invariants

- Hold over all test runs; might not actually be invariants
- Could use statically inferred instead

We wrote our own Daikon front end for machine code

- Assumes debugging information
  - can we remove this restriction?
- Front ends supplied with Daikon not sufficient
  - checks only entry-to-exit invariants, whereas we need
    - loop trip-count instrumentation
    - I/O-to-loop-exit invariants
- Instruments program using Dyninst

Instrumentation remembers I/O vals

If value is returned:

```
x = read_int();                x = __io1 = read_int();
```

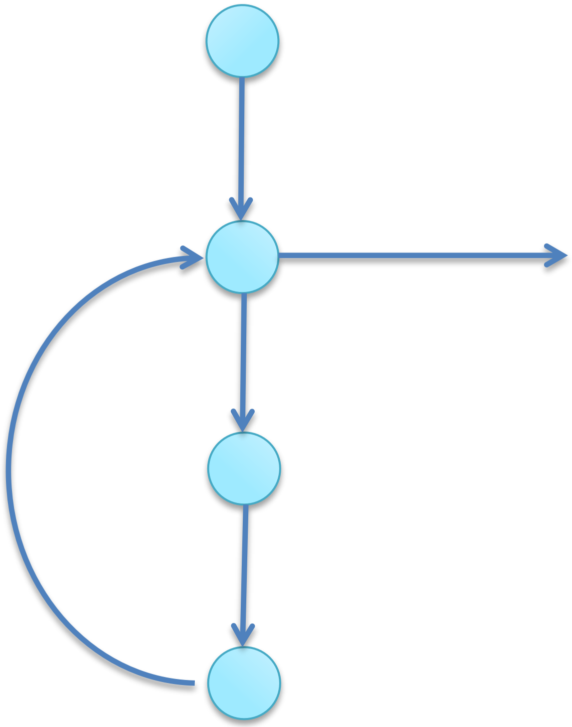
If value is "returned" via out parameter:

```
err = read_int(&x);            err = read_int(&x);
                               __io2 = *(&x);
```

If value is passed by parameter:

```
write_int(x);                  __io3 = x;
                               write_int(x);
```

Instrumentation finds trip counts

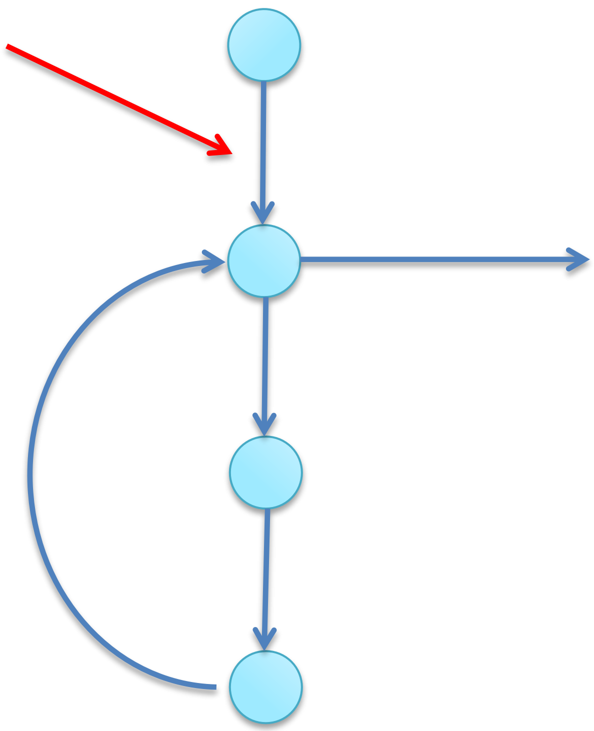


Instrumentation finds trip counts

On loop entry:

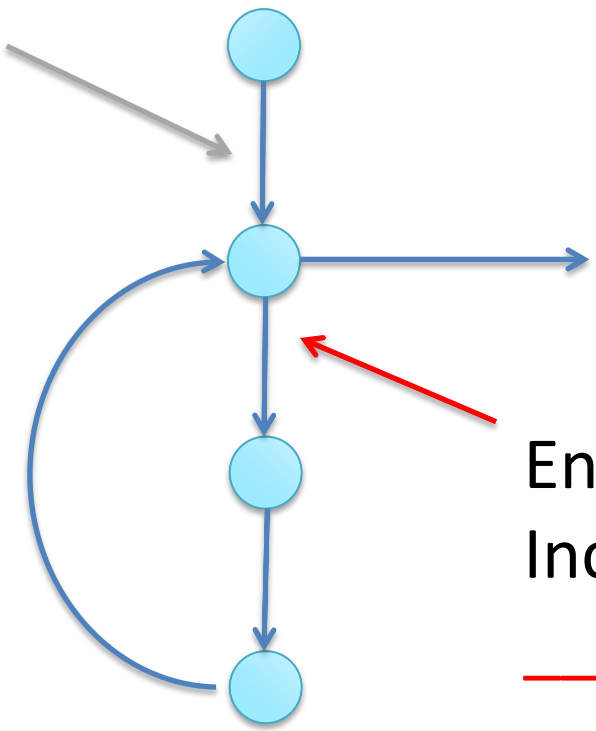
Set trip count to 0

__trip1 = 0;



Instrumentation finds trip counts

On loop entry:
Set trip count to 0
__trip1 = 0;

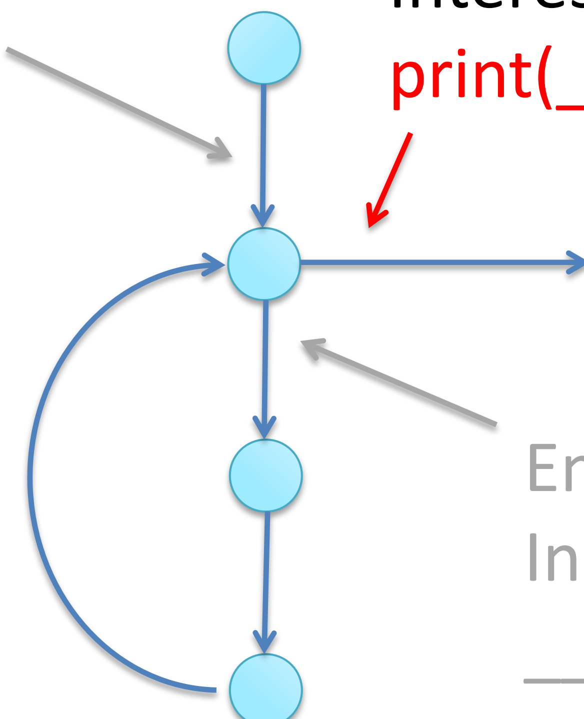


Entering loop body:
Increment trip count
__trip1++;

Instrumentation finds trip counts

On loop entry:
Set trip count to 0
__trip1 = 0;

On loop exit:
Output current value of variables
Interested in invariants here
print(__io1, __io2, …, __trip1);



Entering loop body:
Increment trip count
__trip1++;

We use Daikon to find I/O equalities

Instrumented program → **Value trace** → Dakion dynamic invariant detector → **I/O equalities**

Value trace:

LOOP_EXIT_A
__io2 = 2
__io4 = 5
__trip_count_A = 5

LOOP_EXIT_B
__io2 = 6
__io4 = 5
__trip_count_B = 6

I/O equalities:

__trip_count_A = __io4 = 5
__trip_count_B = __io2 = 6

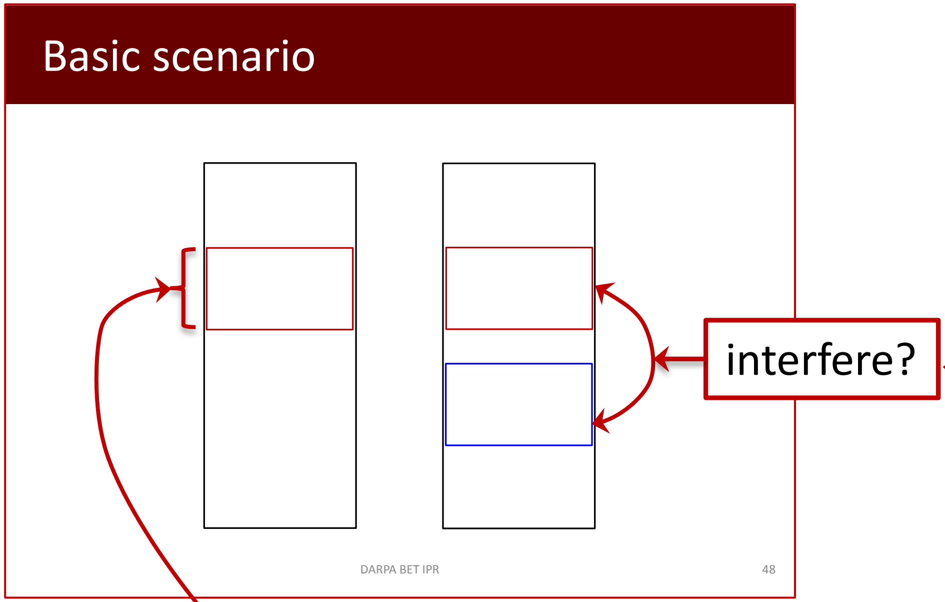
We model programs as XFAs

XFAs: *extended* finite automata

Add separate bounded “data state” to standard FAs

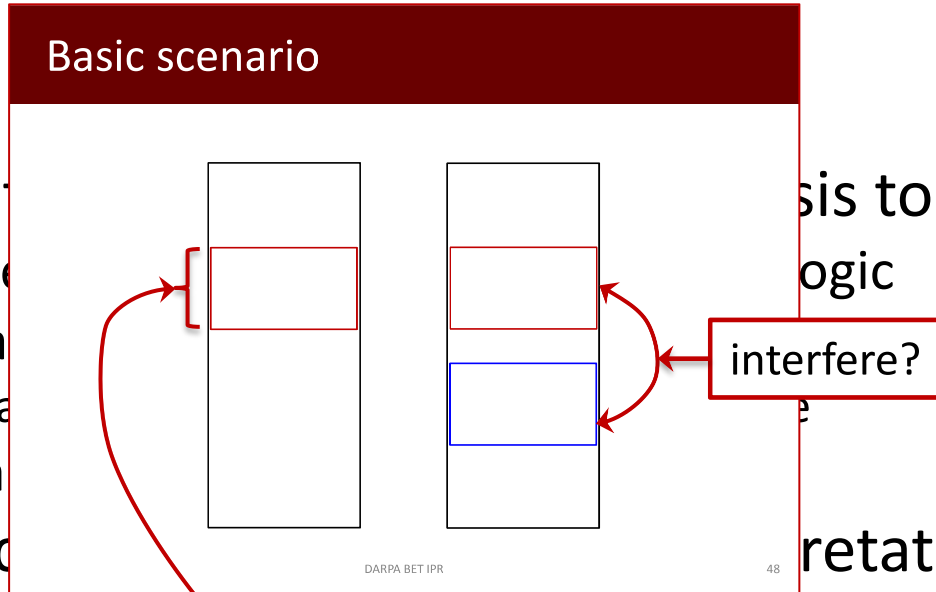
Transformers on transitions describe data-state changes

Symbolic abstraction: Who cares?



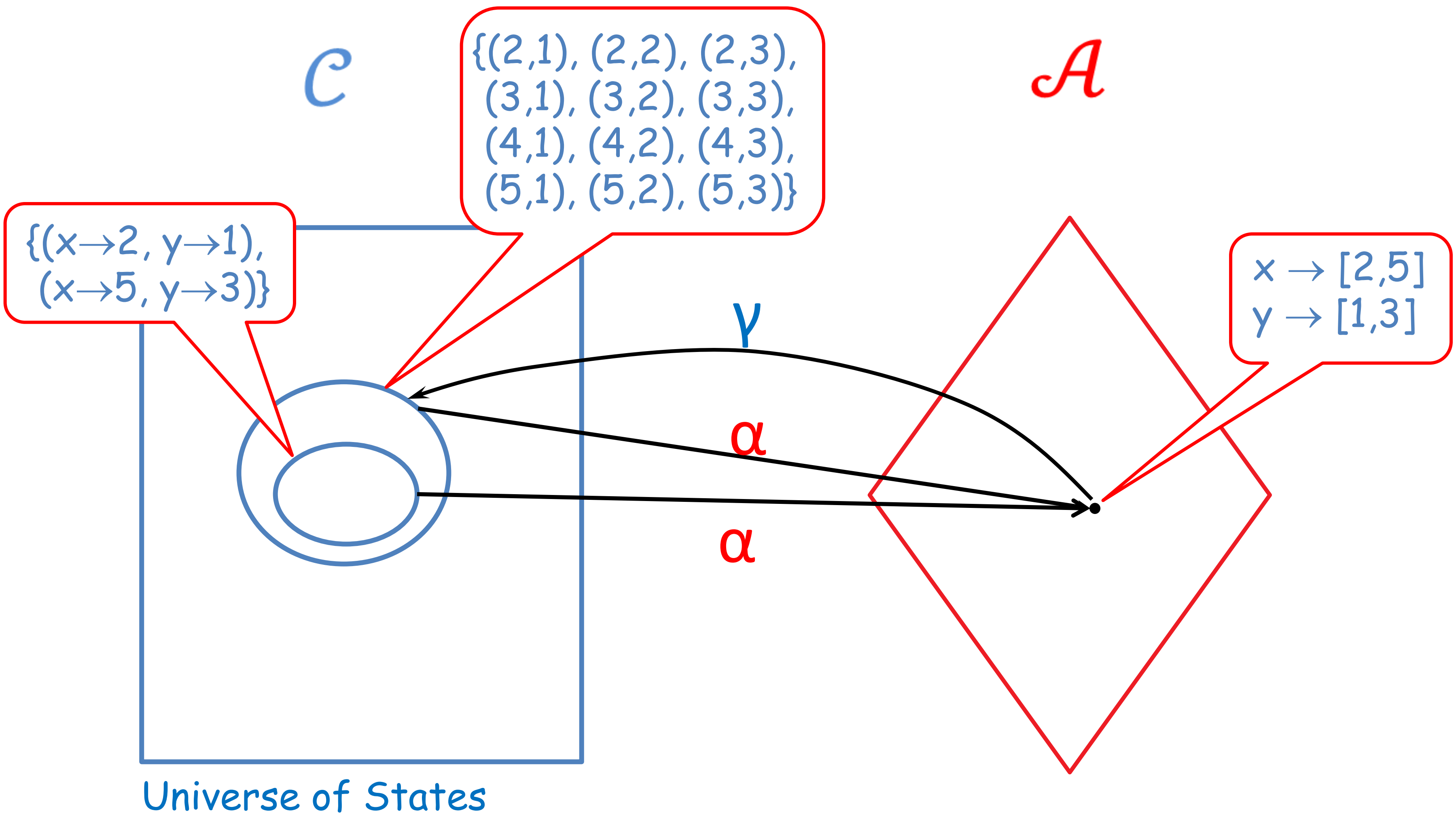
- More precise results in abstract interpretation
  - can identify loop and procedure summaries that are more precise than ones obtained via conventional techniques
- Applies to interesting, non-standard logics (we think!)
  - separation logic: memory safety properties

Symbolic abstraction: Who cares?

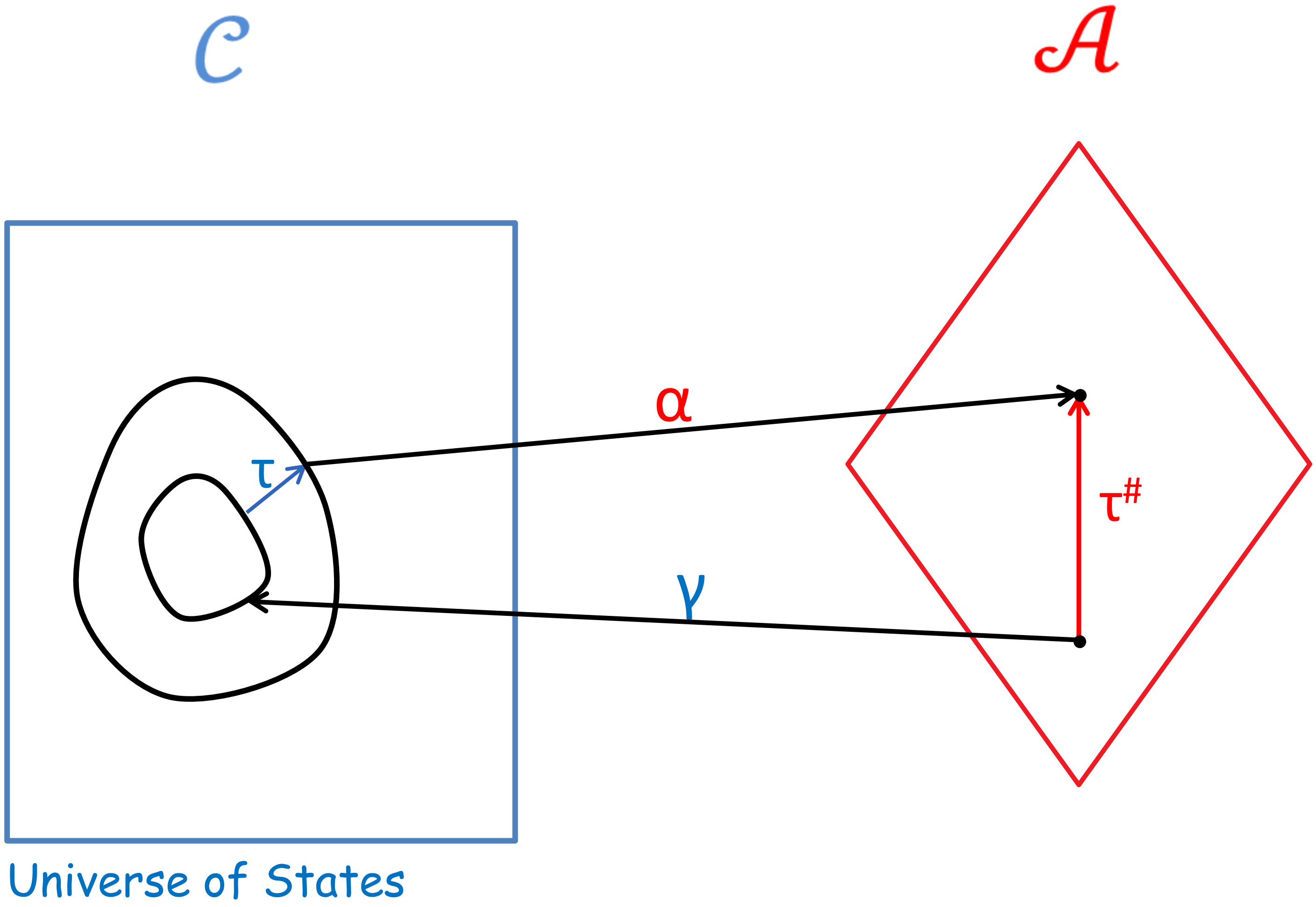


- Win, win,
- Easier/fas sis tools
  - just stat ogic
  - supply a
    - e.g., a e
  - obtain a
- More prec retation
  - can identify loop and procedure summaries that are more precise than ones obtained via conventional techniques
- Applies to interesting, non-standard logics (we think!)
  - separation logic: memory safety properties
- Improve level of automation for creating analyzers
  - implement analysis tools in a much smaller time-span and with drastically reduced programmer effort

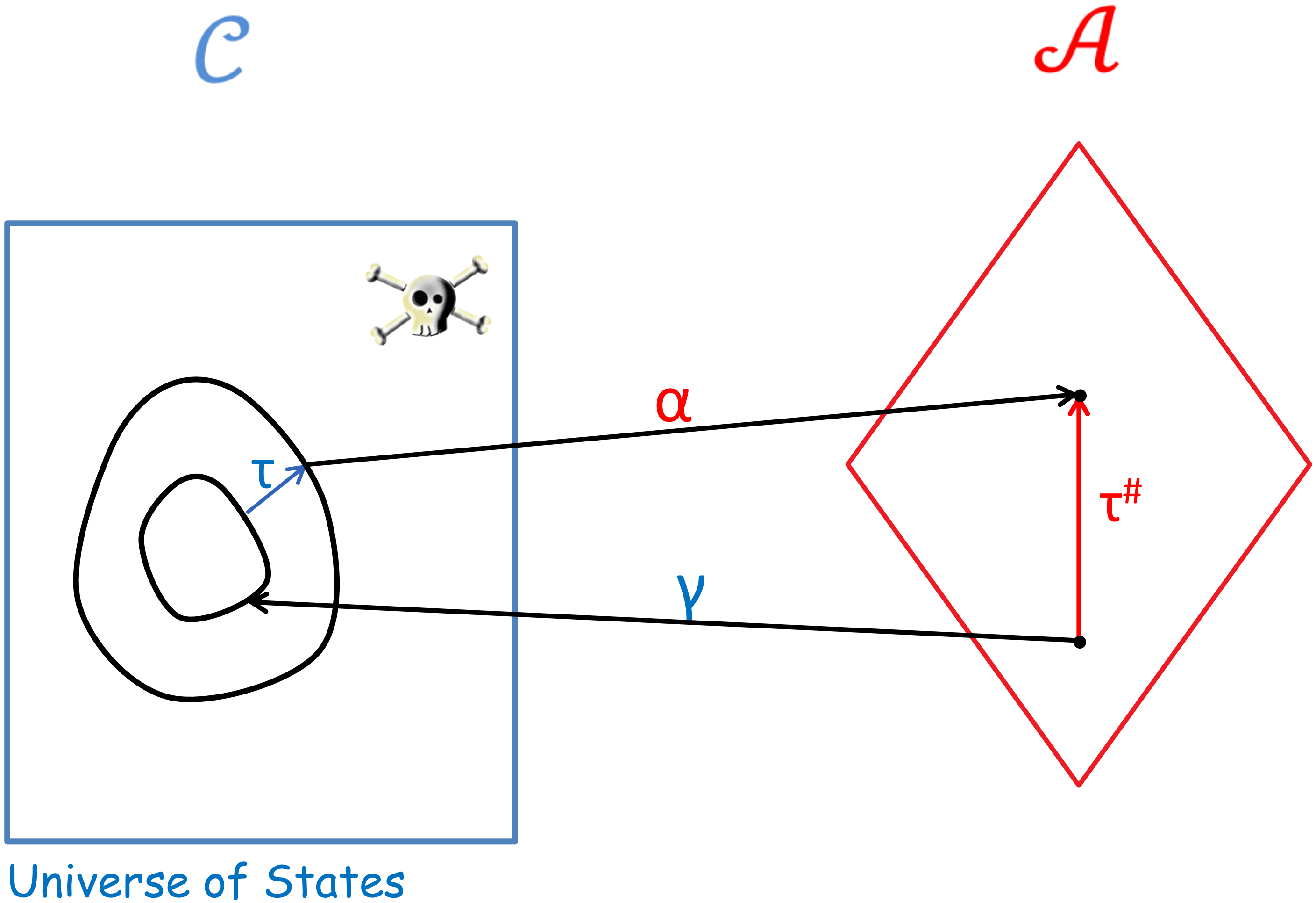
In 1977, Cousot & Cousot gave us a beautiful theory of overapproximation



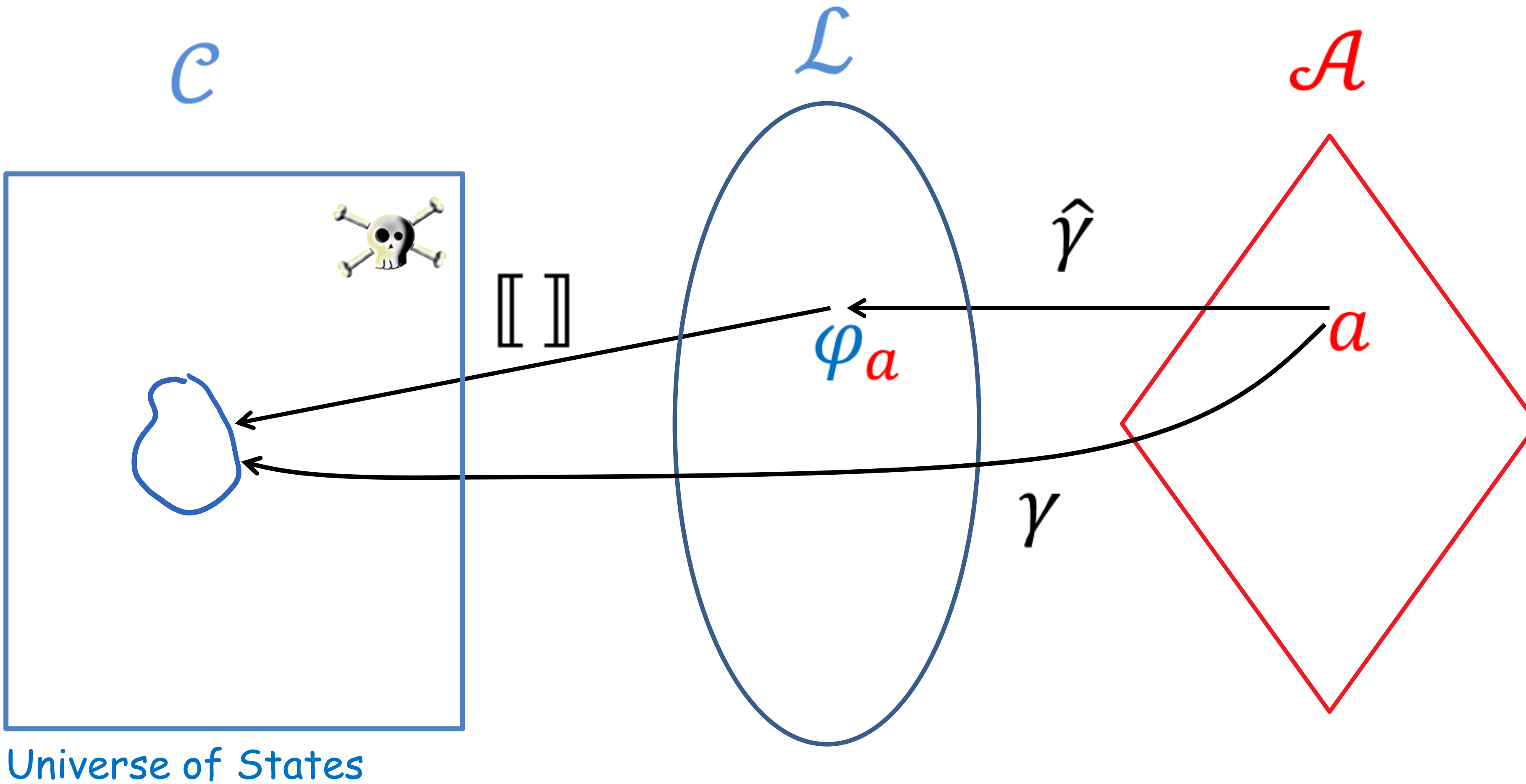
In 1979, Cousot & Cousot gave us:



In 1979, Cousot & Cousot gave us:



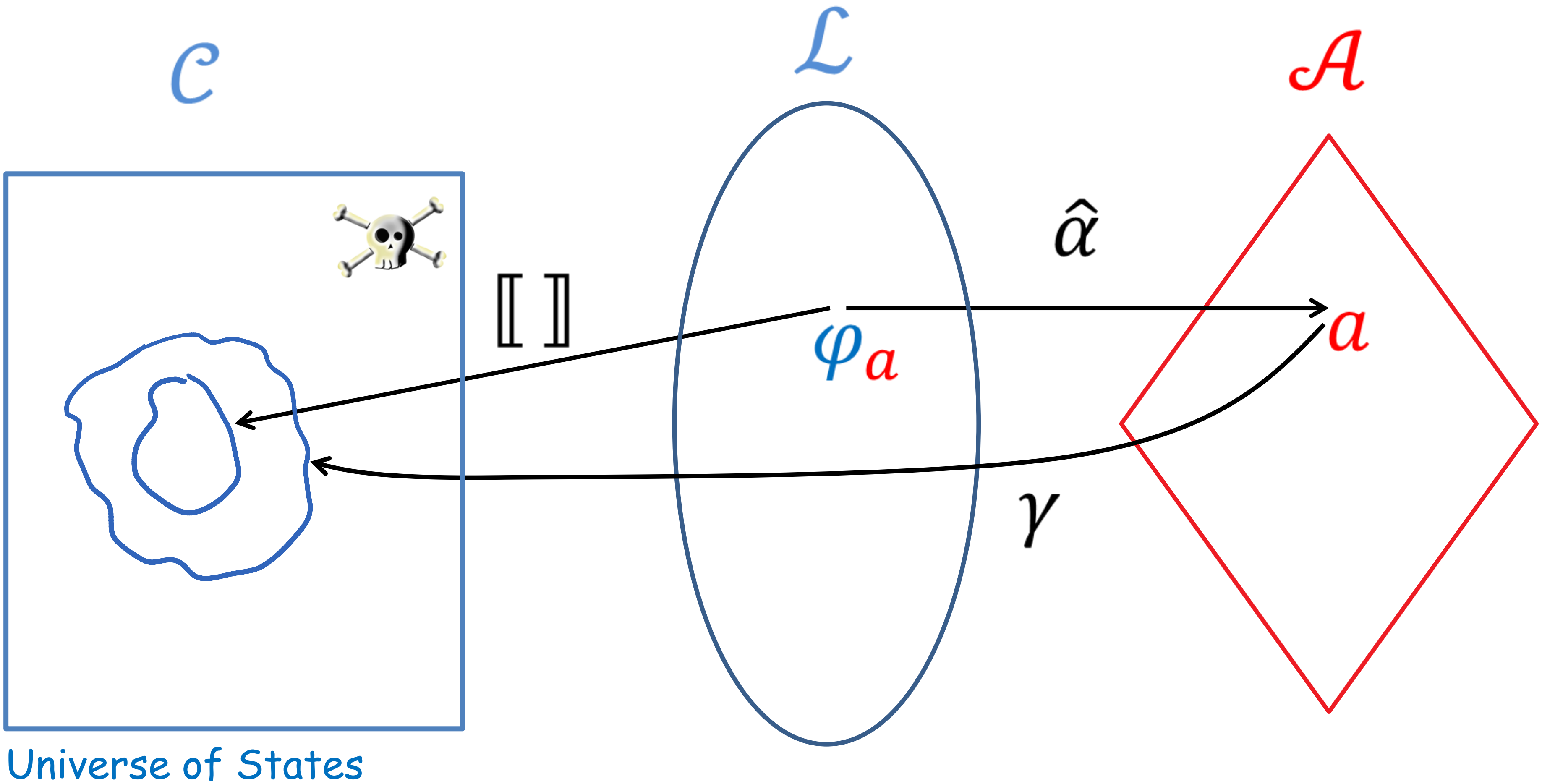
In 2004, Reps, Sagiv, and Yorsh gave us:



Symbolic Abstract Interpretation

Symbolic Concretization

In 2004, Reps, Sagiv, and Yorsh gave us:



Symbolic Abstraction

$\hat{\alpha}^{\uparrow}(\varphi)$ [VMCAI 2004]

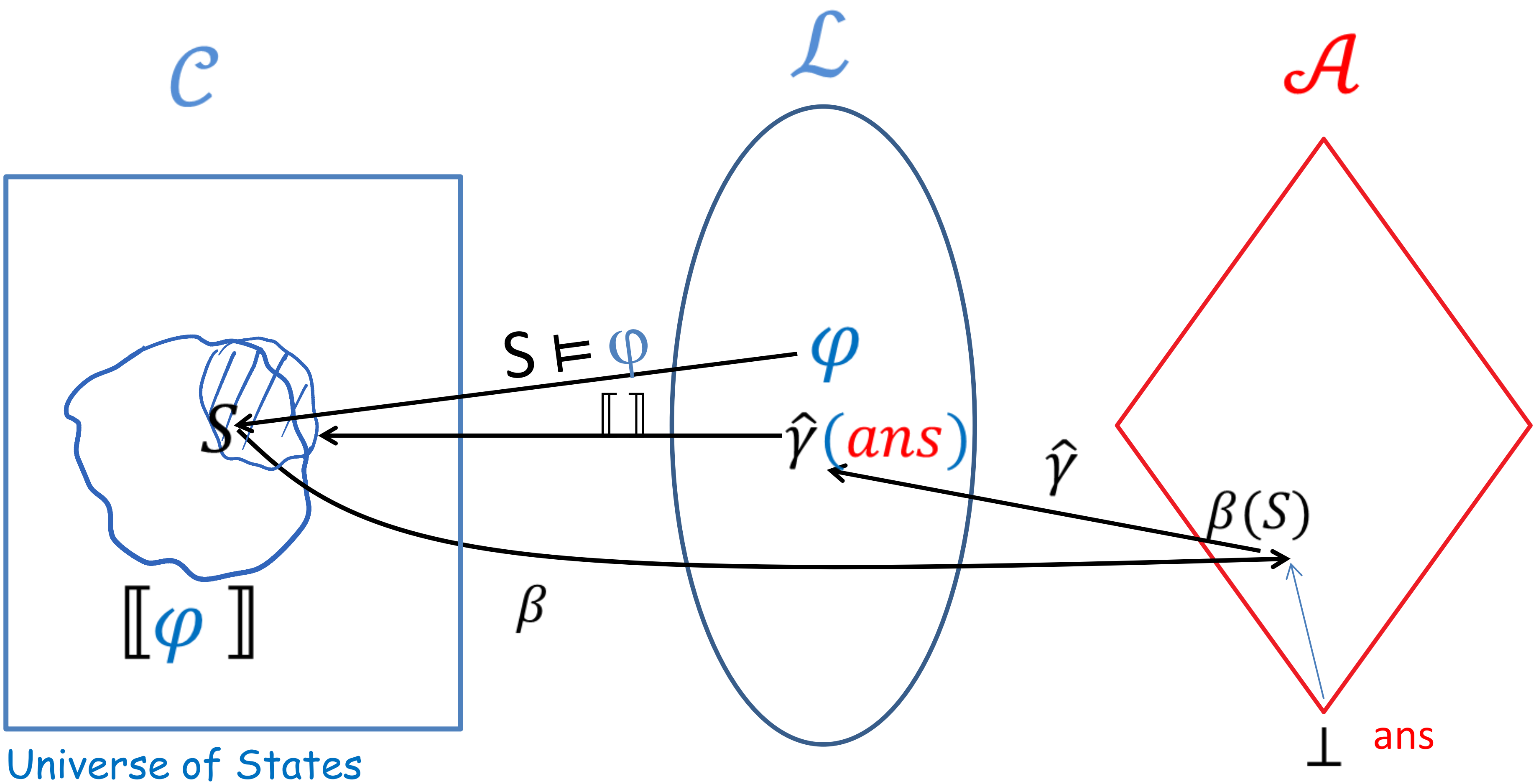
$\mathcal{C}$

Use SMT solvers to get leverage:
get models of φ

Universe of States

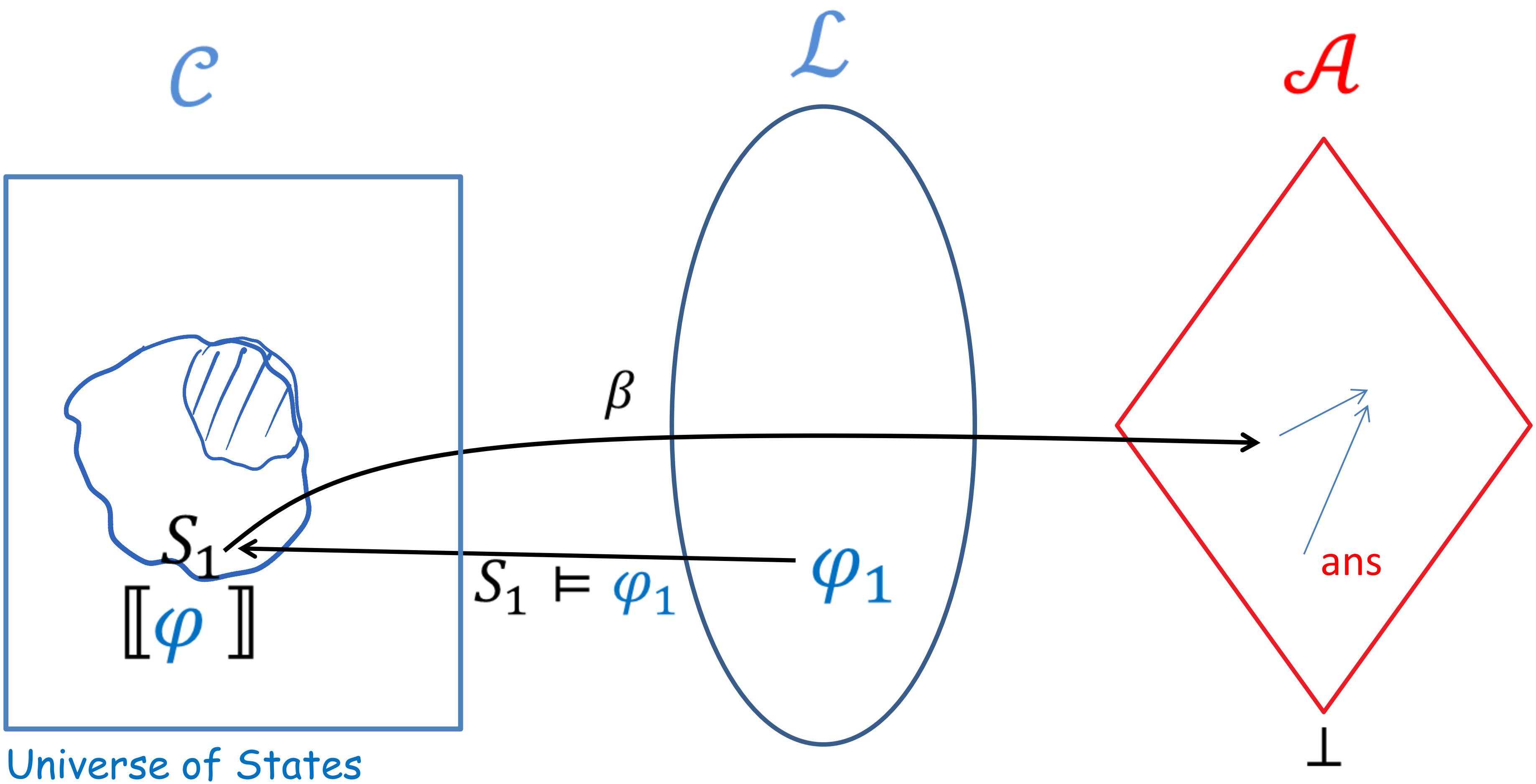
$\hat{\alpha}^{\uparrow}(\varphi)$

[VMCAI 2004]



$\hat{\alpha}^{\uparrow}(\varphi)$

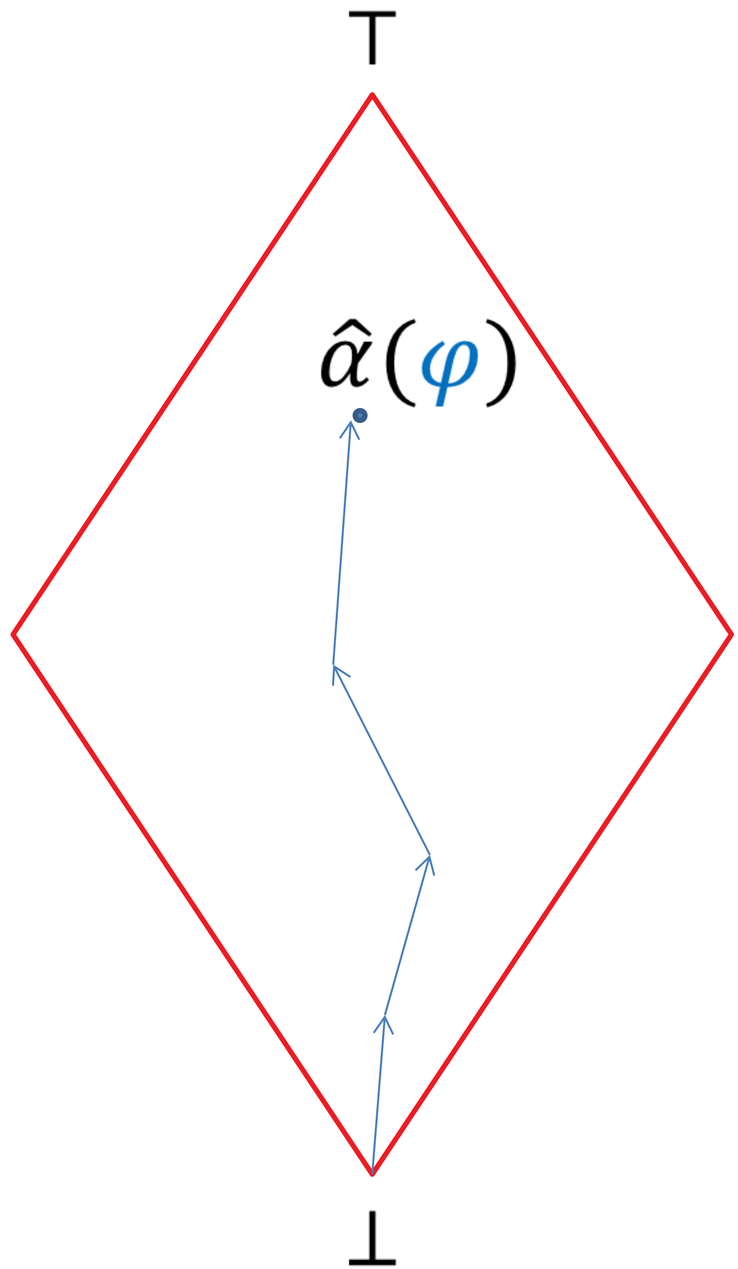
[VMCAI 2004]



$$\varphi_1 = \varphi \wedge \neg\hat{\gamma}(\text{ans})$$

$\hat{\alpha}^{\uparrow}(\varphi)$

[VMCAI 2004]

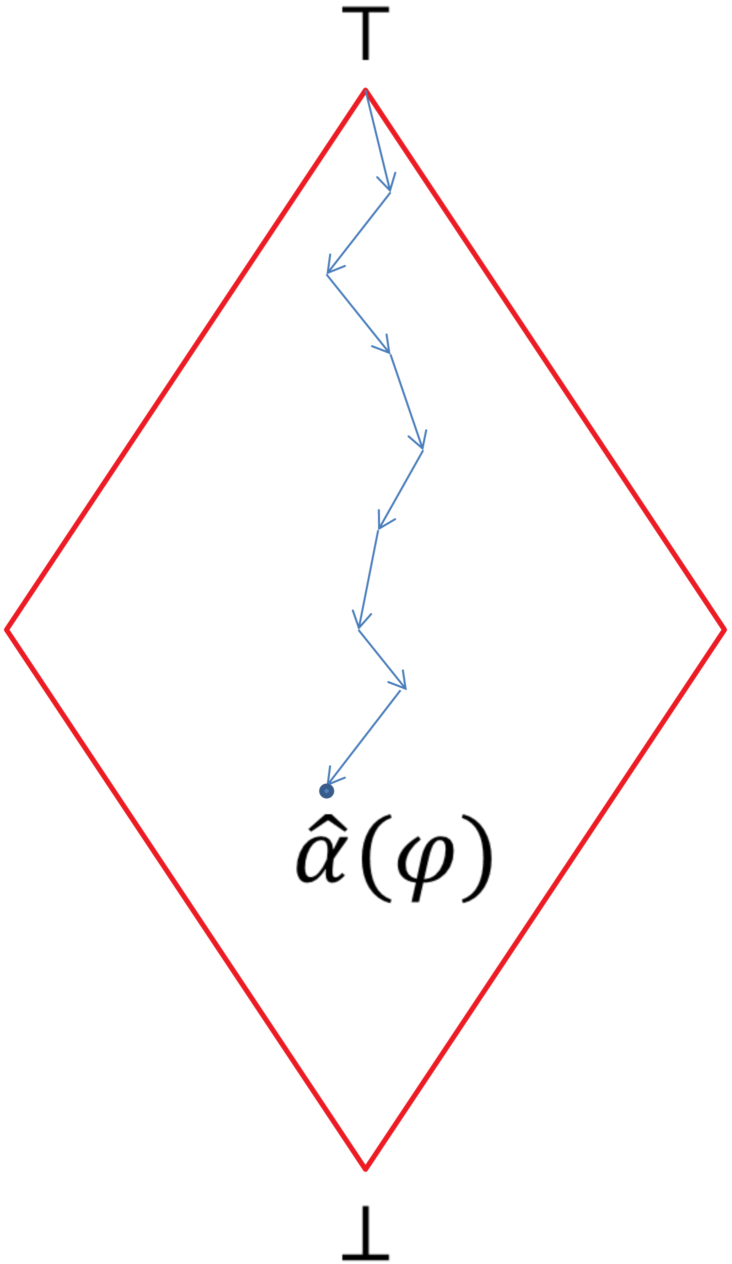


From "Below" vs. From "Above"

- Reps, Sagiv, and Yorsh 2004: approximation from "below"
- Desirable: approximation from "above"
  - always have safe over-approximation in hand
  - can stop algorithm at any time (e.g., if taking too long)
  - Thakur, A. and Reps, T., A method for symbolic computation of abstract operations. In *Proc. Computer-Aided Verification* (CAV), 2012

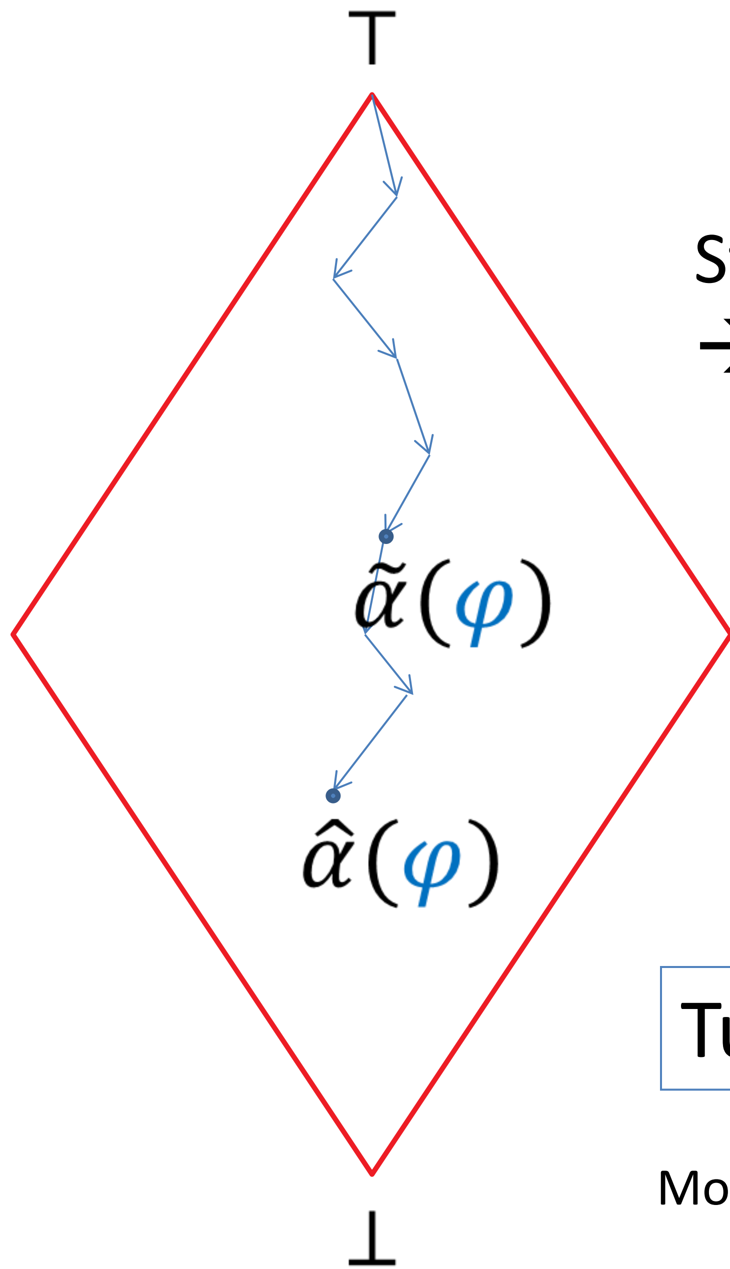
Aditya Thakur

[CAV 2012]



Key Feature

[CAV 2012]



Stop at any time
→ sound answer

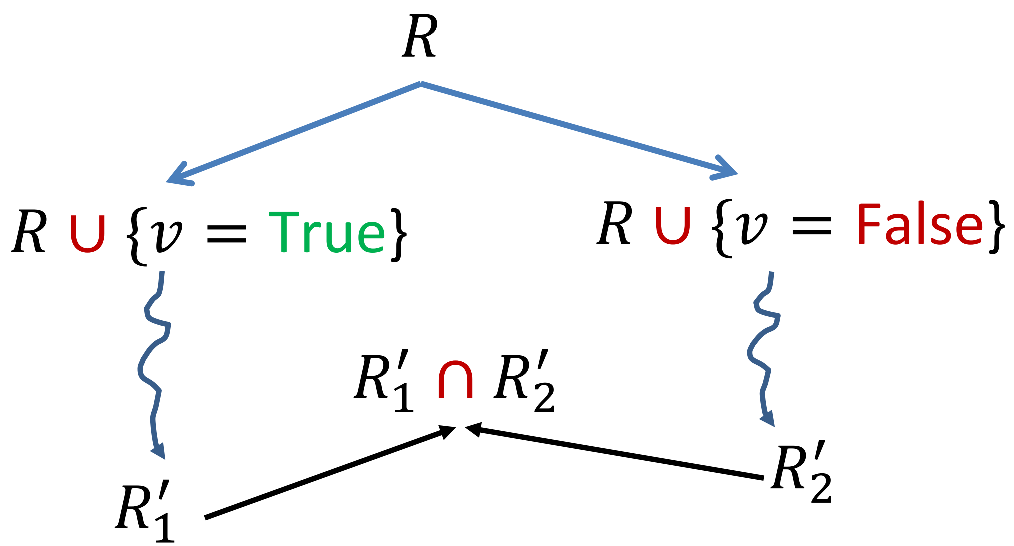
Tunable

More time → more precision

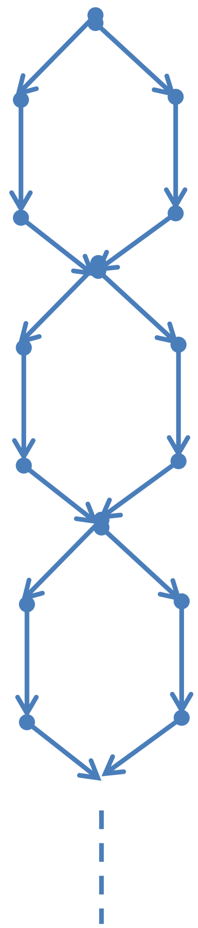
Stålmarck's method (1989)

Dilemma Rule

- Split
- Propagate
- Merge

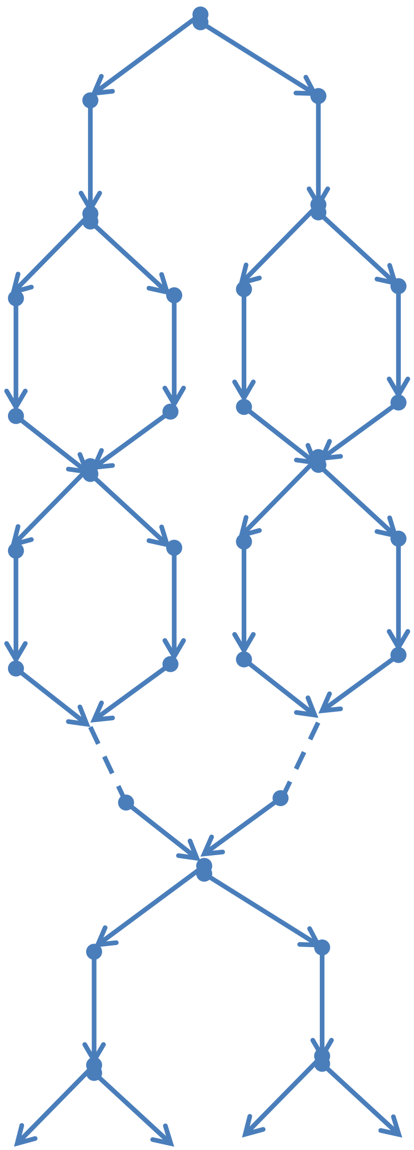


Stålmarck's method (1989)



1-saturation

Stålmarck's method (1989)

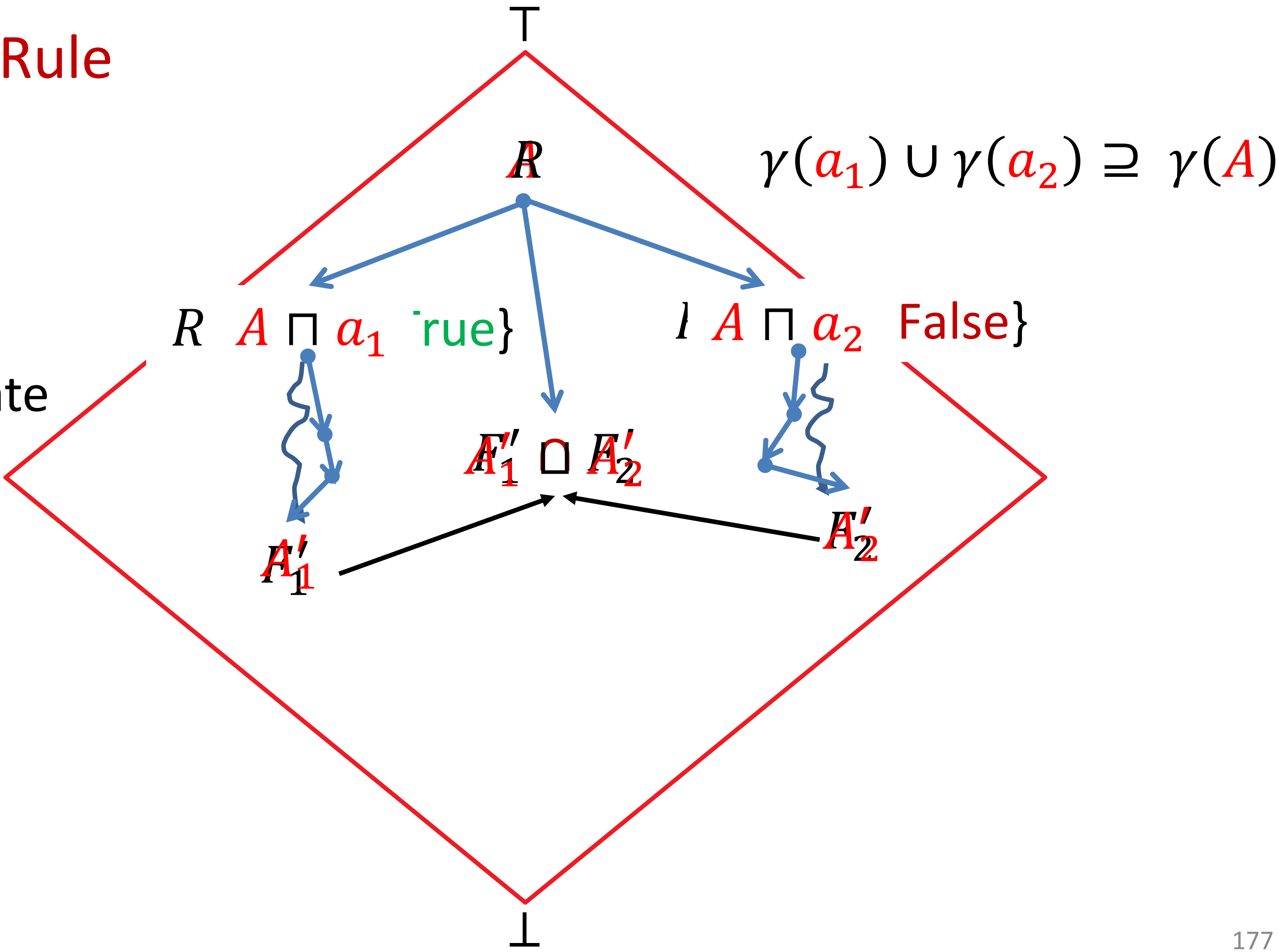


2-saturation

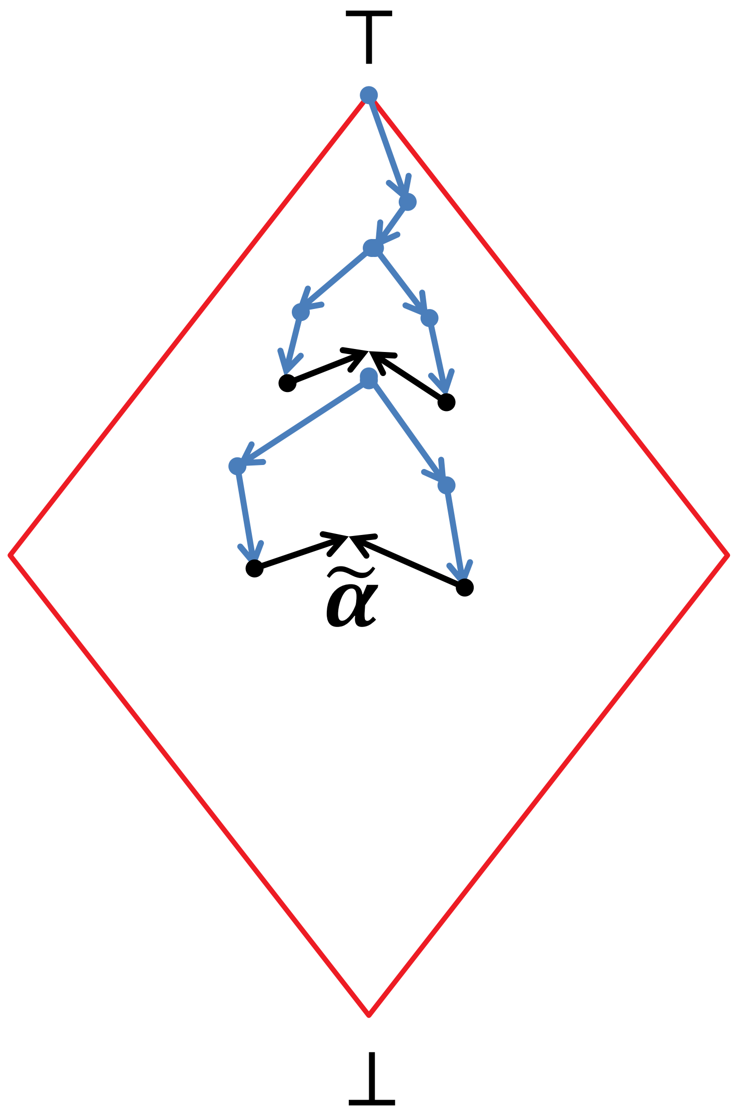
Stålmarck's method for $\tilde{\alpha}^{\downarrow}$

Dilemma Rule

- Split
- Propagate
- Merge



Stålmarck's method

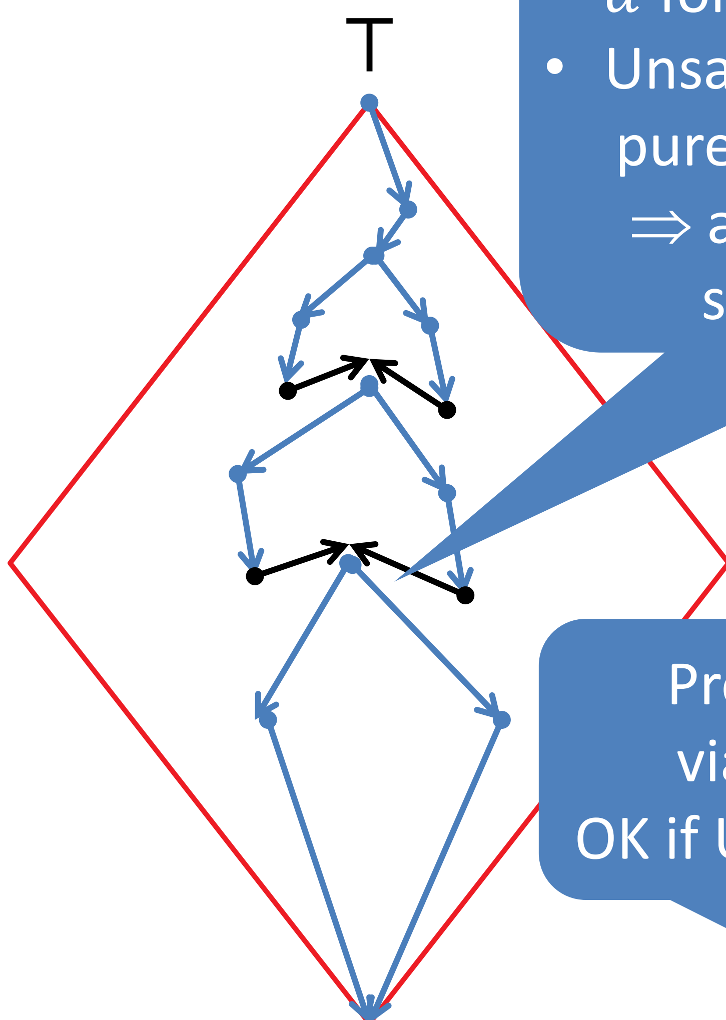


Key Feature

[CAV 2012]

Reasoning: Using $\tilde{\alpha}^{\downarrow}(\varphi)$

Dual use:
- $\tilde{\alpha}$ for abstract interpretation
- Unsat/validity checking for pure logical reasoning
  $\Rightarrow$ abstract interpretation in service to logic!

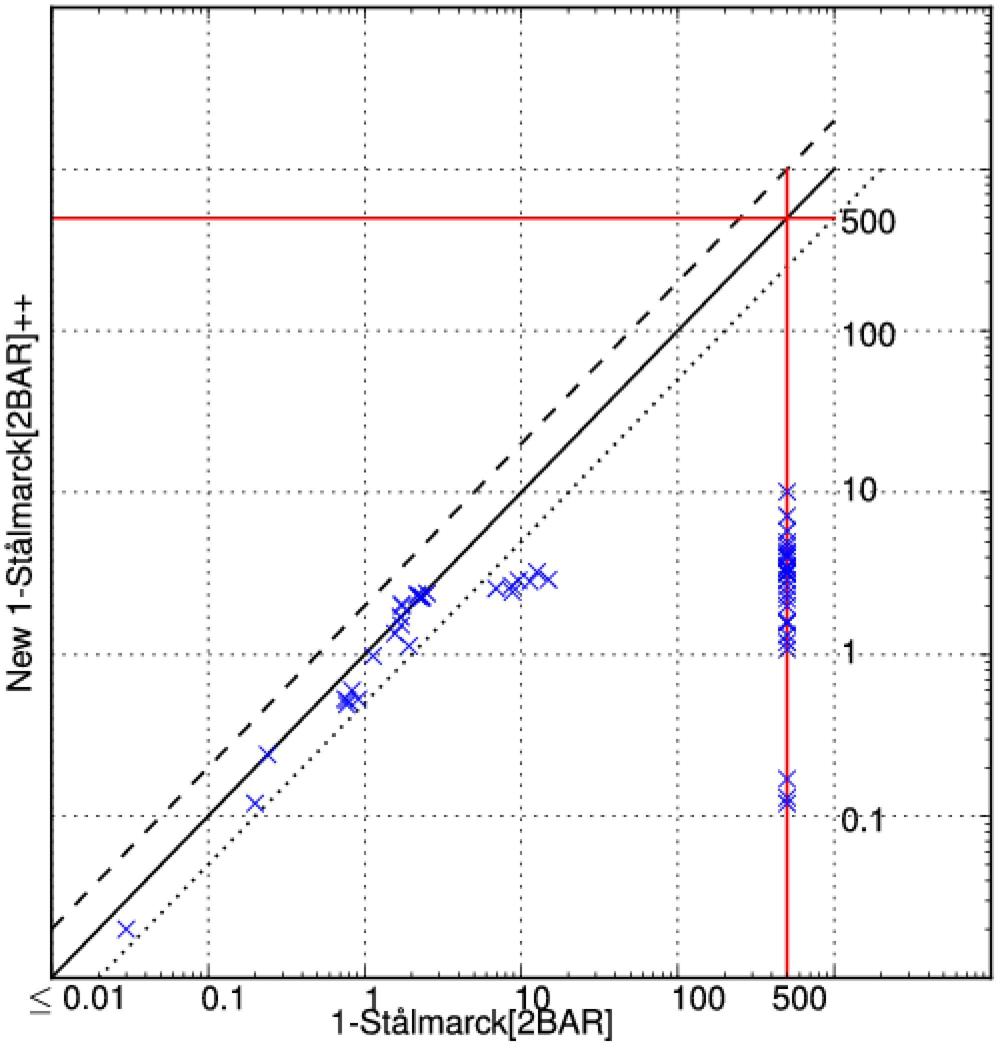


Property verification via model checking:
OK if Unsat(Program $\wedge$ Bad)

$\tilde{\alpha}^{\downarrow}(\varphi) = \bot$ $\quad\therefore \varphi$ is unsatisfiable

The importance of data structures

- Classic union-find
  - plus layers
  - plus least-upper bound
- Given UF_1 and UF_2, find the coarsest partition that is finer than UF_1 and UF_2
- Roughly, “confluent, partially-persistent union-find”



D

Extend WALi to use $\hat{\alpha}$

- Weighted Automaton Library (WALi):
  - supports context-sensitive interprocedural analysis
  - weights = dataflow transformers
  - weighted version of PDSs (a la material on specialized slicing)
- More precise results in abstract interpretation
- Easier implementation of analysis tools

Junghee Lim

Aditya Thakur

AlphaHat

- AlphaHat technique in three ways
  - WALi + AlphaHat (Aditya Thakur and Junghee Lim)
    - ~October 2012
  - Boogie + AlphaHat for source code (Akash Lal at Microsoft India)
    - ~November 2012
  - Boogie + AlphaHat for machine code (Aditya Thakur and Junghee Lim)
    - ~November 2012

Outline of Talk

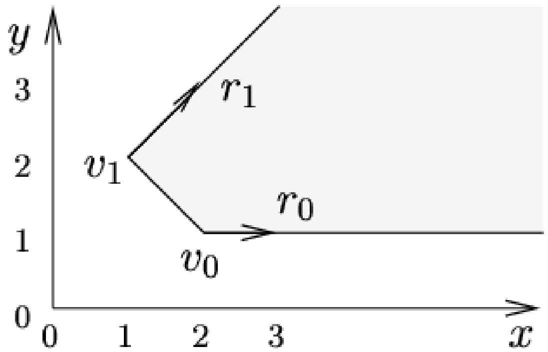
- Review of goals
- Progress (Oct. 2012 - May 2013)
  - Component identification
    - Recovering class hierarchies using dynamic analysis
  - Verifying component properties
    - Symbolic abstraction (BET + ONR STTR)
    - Domain-combination technique: combine results from multiple analysis methods
    - Abstract domain of bit-vector inequalities
    - Format-compatibility checking (ONR)
  - Component extraction
    - Specialization slicing
    - Partial evaluation of machine code
- Recap of publications/submissions
- Recap of plans for 2013

Possible-overflow example

```
char* concat(char* a, char* b)
{
    unsigned size = strlen(a)+strlen(b)+1;
    char* out = (char*)malloc(size*sizeof(char));      // Possible overflow
    for(unsigned i = 0; i < strlen(a); i++)  {
        out[i] = a[i];      // Potential memory corruption
    }
    for(unsigned i = 0; i < strlen(b); i++) {
        out[i+strlen(a)] = b[i];      // Potential memory corruption
    }
    out[i+strlen(a)] = '\0';
    return out;
}
```

Convex Polyhedra

[Figures from Halbwachs et al. FMSD97]



$$P = \left\{ (x,y) \mid \begin{pmatrix} y & \geq & 1 \\ x+y & \geq & 3 \\ -x+y & \leq & 1 \end{pmatrix} \right\}$$

$$V = \left\{ v_0 \begin{pmatrix} 2 \\ 1 \end{pmatrix}, v_1 \begin{pmatrix} 1 \\ 2 \end{pmatrix} \right\} \quad R = \left\{ r_0 \begin{pmatrix} 1 \\ 0 \end{pmatrix}, r_1 \begin{pmatrix} 1 \\ 1 \end{pmatrix} \right\}$$

Figure 1: A convex polyhedron and its 2 representations

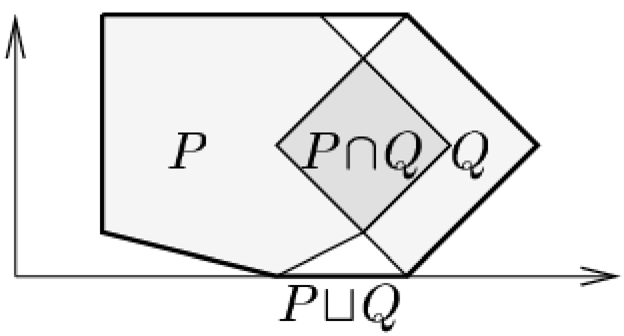


Figure 2: Intersection and convex hull

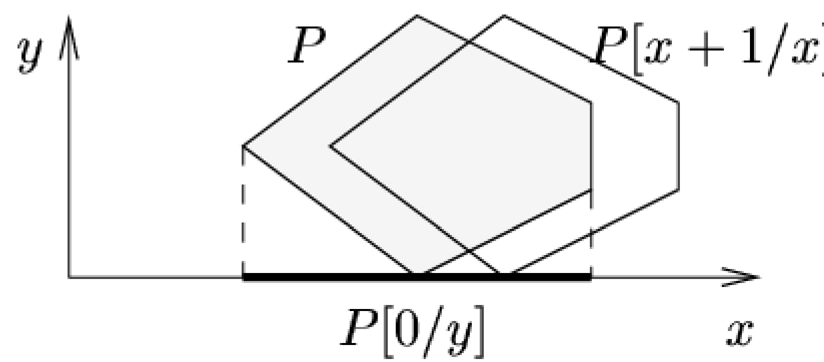


Figure 3: Linear transformations

Bitvector Inequality domain

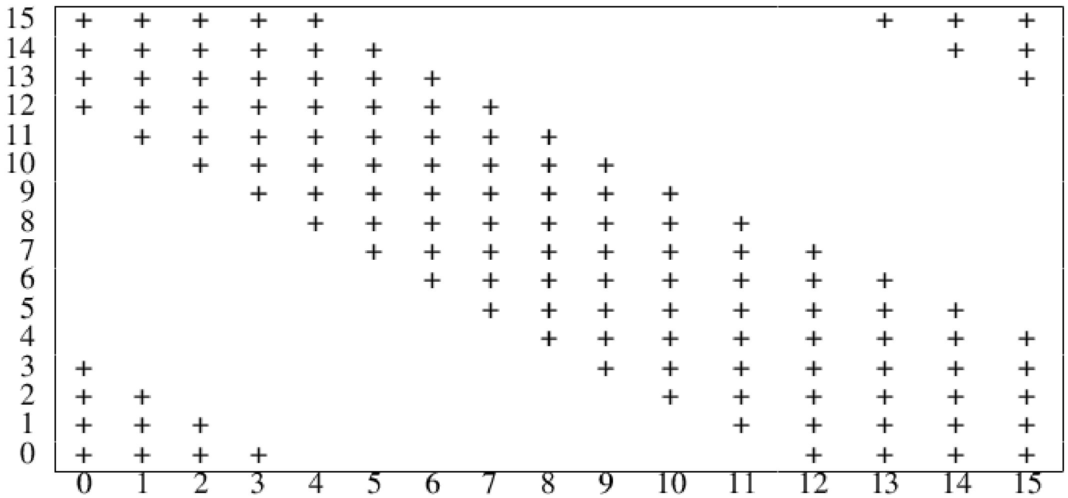
- Conventional domain for representing inequalities
  - polyhedra: conjunctions of linear inequalities

    $$a_1 x_1 + a_2 x_2 + \ldots + a_k x_k \leq c$$

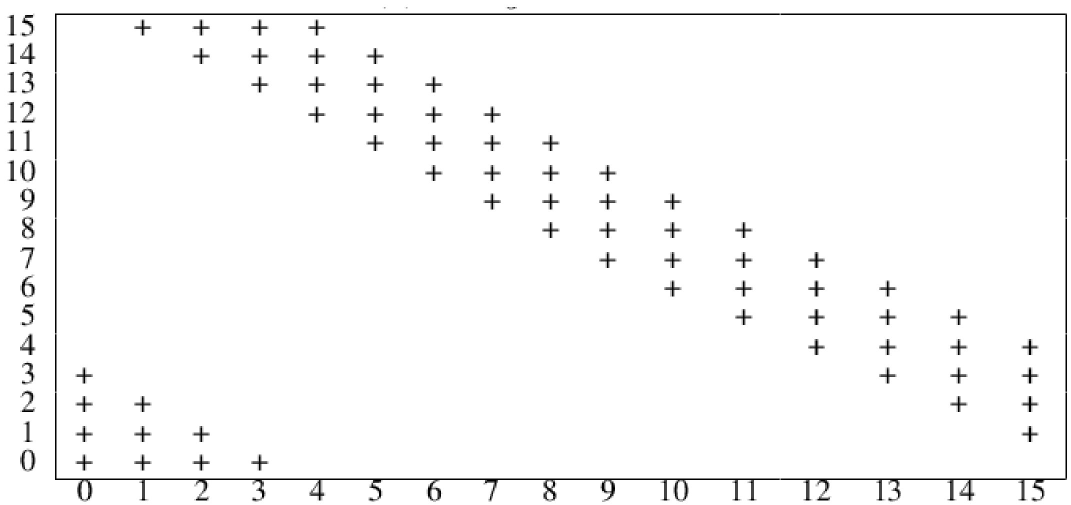
  - operations on polyhedra: linear transformations
    - unsound for machine arithmetic
    - machine integers wrap while mathematical integers do not
- Solution: Bitvector Inequality Domain

Tushar Sharma

Not so well-behaved . . .



(a) $x + y + 4 \leq 7$



(b) $x + y \leq 3$