태초부터 자동으로:
반복문 불변식을 자동 유추할 때 사용할 부품식들의 자동 유추

ROSAEC 여름 워크샵

2010년 8월 27일

이원찬

wclee@ropas.snu.ac.kr

(이 연구는 Bow-Yaw Wang 교수님, 정영범 연구원, 이광근 교수님과의 공동 연구입니다.)

Inferring Loop Invariant Automatically

Problem: Loop Invariant Generation

For the annotated loop

$$\{\delta\}\ \texttt{while}\ \rho\ \texttt{do}\ S\ \texttt{end}\ \{\epsilon\}$$

Find an invariant ι satisfying the following conditions:

(A) $\delta \Rightarrow \iota$ (ι holds when entering the loop)

(B) $\iota \wedge \neg\rho \Rightarrow \epsilon$ (ι gives ϵ after leaving the loop)

(C) $\iota \wedge \rho \Rightarrow Pre(\iota, S)$ (ι holds at each iteration)

Problem: Loop Invariant Generation

For the annotated loop

$$\{\delta\}\ \texttt{while}\ \rho\ \texttt{do}\ S\ \texttt{end}\ \{\epsilon\}$$

Find an invariant ι satisfying the following conditions:

(A) $\delta \Rightarrow \iota$ (ι holds when entering the loop)

(B) $\iota \wedge \neg\rho \Rightarrow \epsilon$ (ι gives ϵ after leaving the loop)

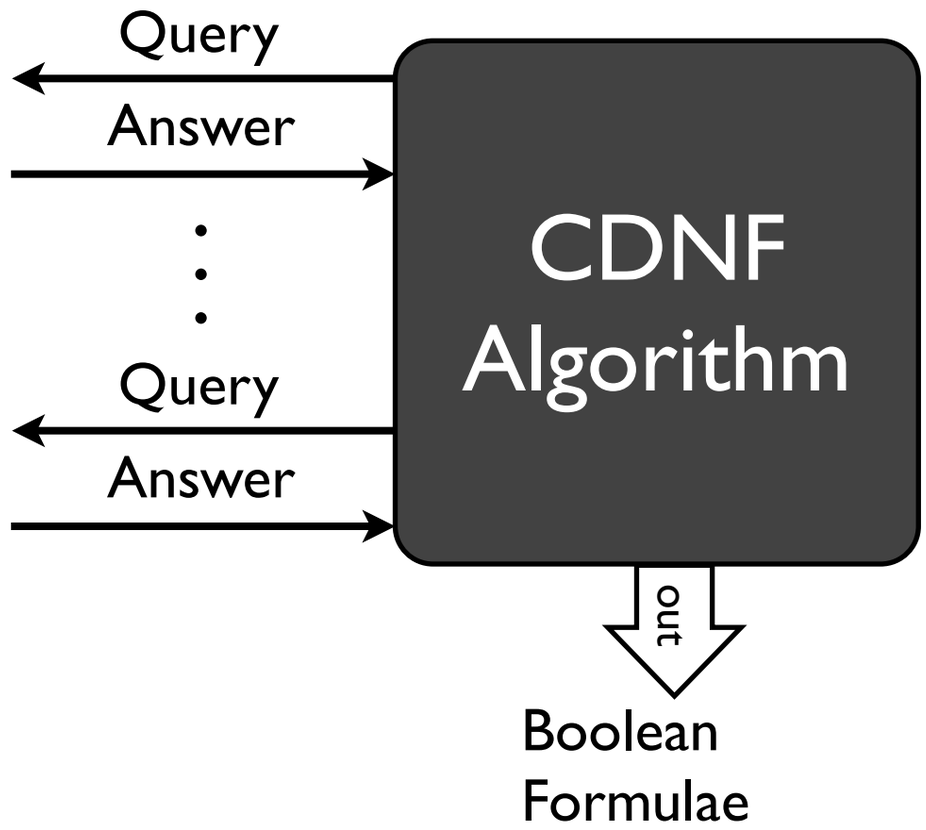
(C) $\iota \wedge \rho \Rightarrow Pre(\iota, S)$ (ι holds at each iteration)

$$\underline{\delta} \Rightarrow \iota \Rightarrow \overline{\epsilon \vee \rho}$$

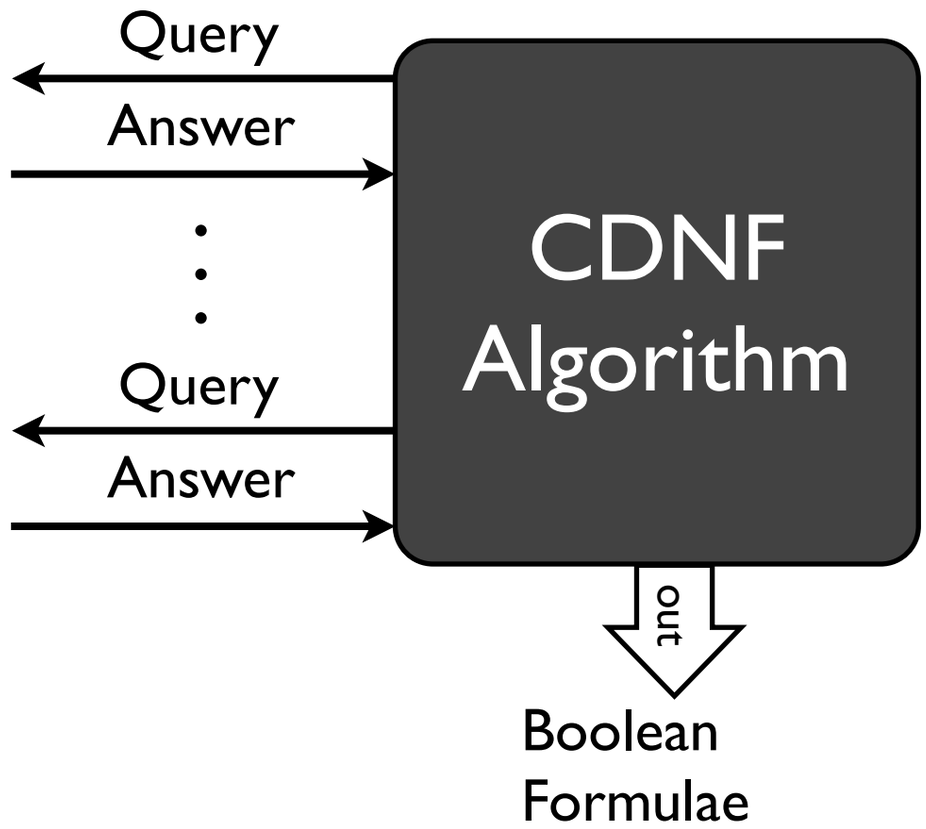
$\underline{\iota}$ strongest under-approximation of an loop invariant

$\overline{\iota}$ weakest over-approximation of an loop invariant

CDNF Learning Algorithm



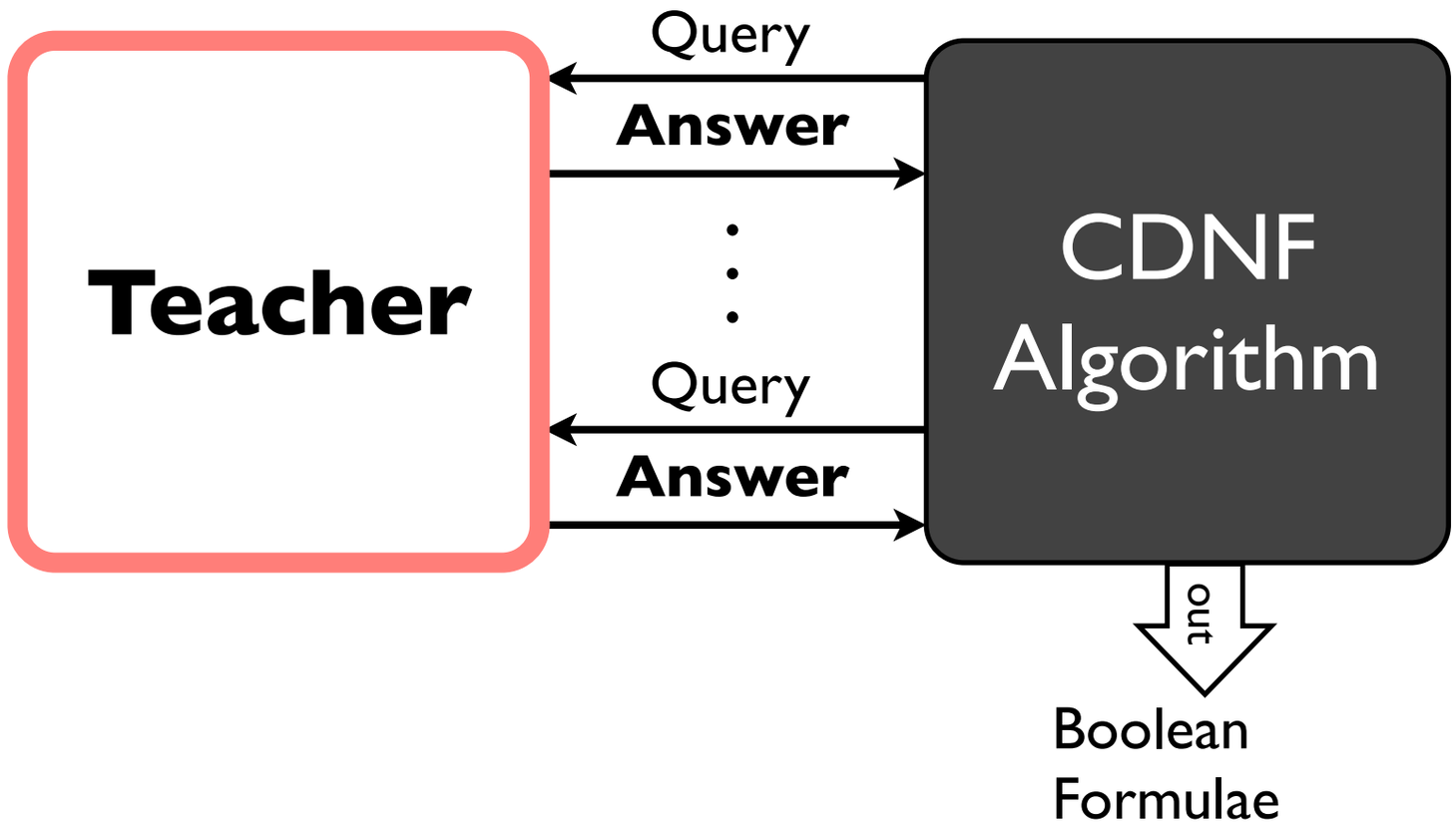
CDNF Learning Algorithm



Membership query $MEM(\mu)$ Can μ be a model of boolean formula?
=> Yes / No

Equivalence query $EQ(\beta)$ Is β equivalent to boolean formula?
=> Yes / counterexample

Solution:
Giving Answers to CDNF



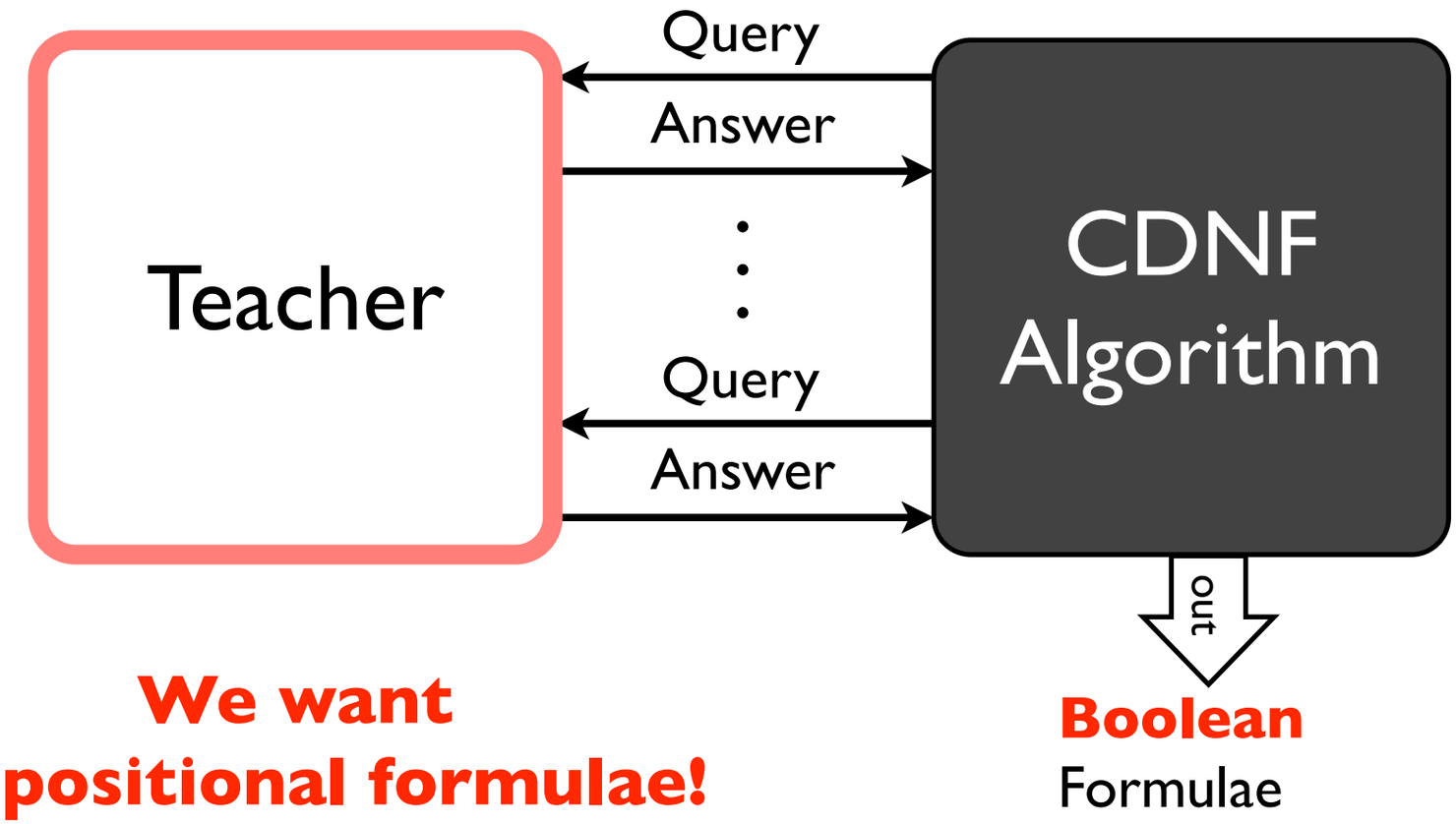
Membership query $MEM(\mu)$ Can μ be a model of boolean formula?

=> Yes / No

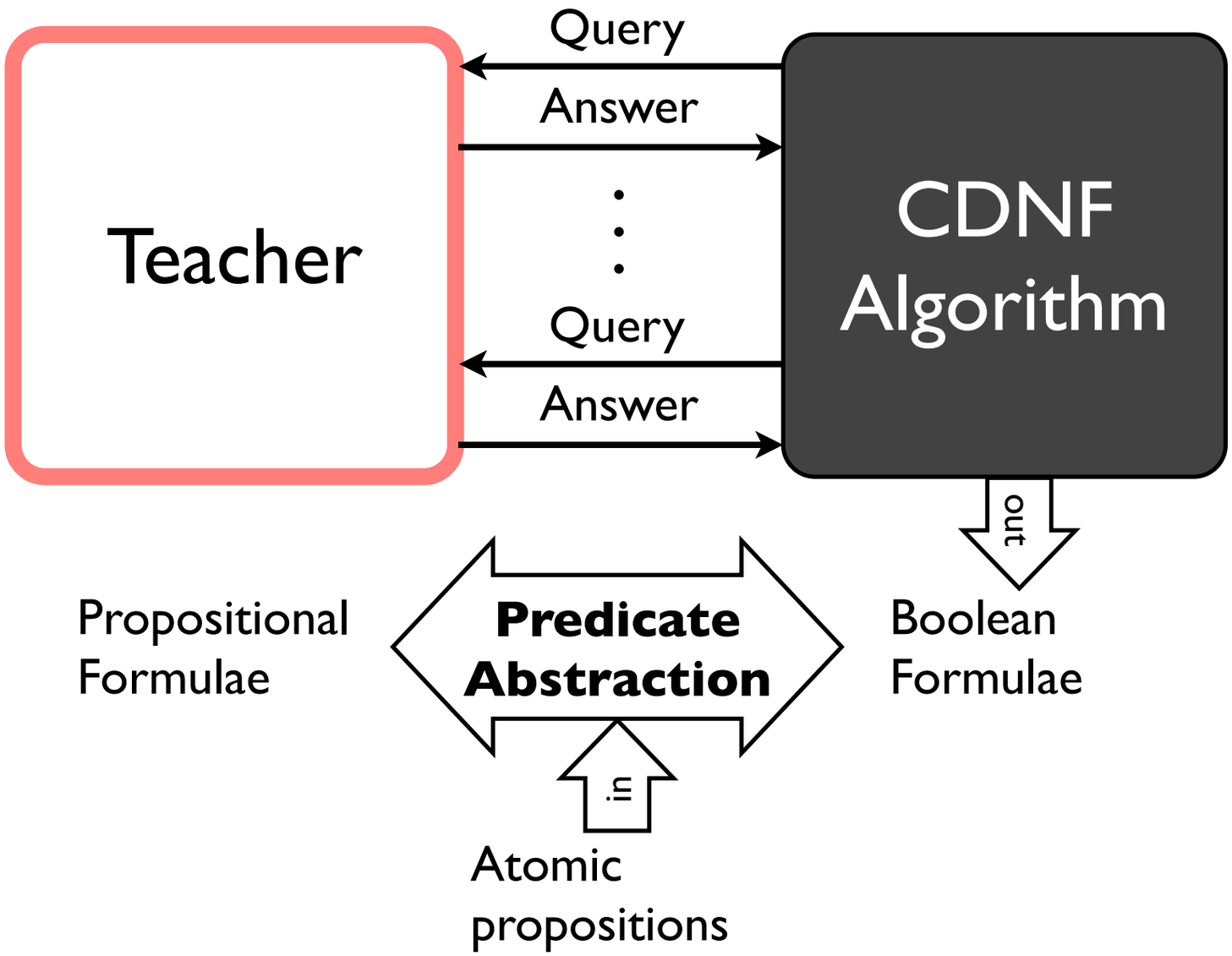
Equivalence query $EQ(\beta)$ Is β equivalent to boolean formula?

=> Yes / counterexample

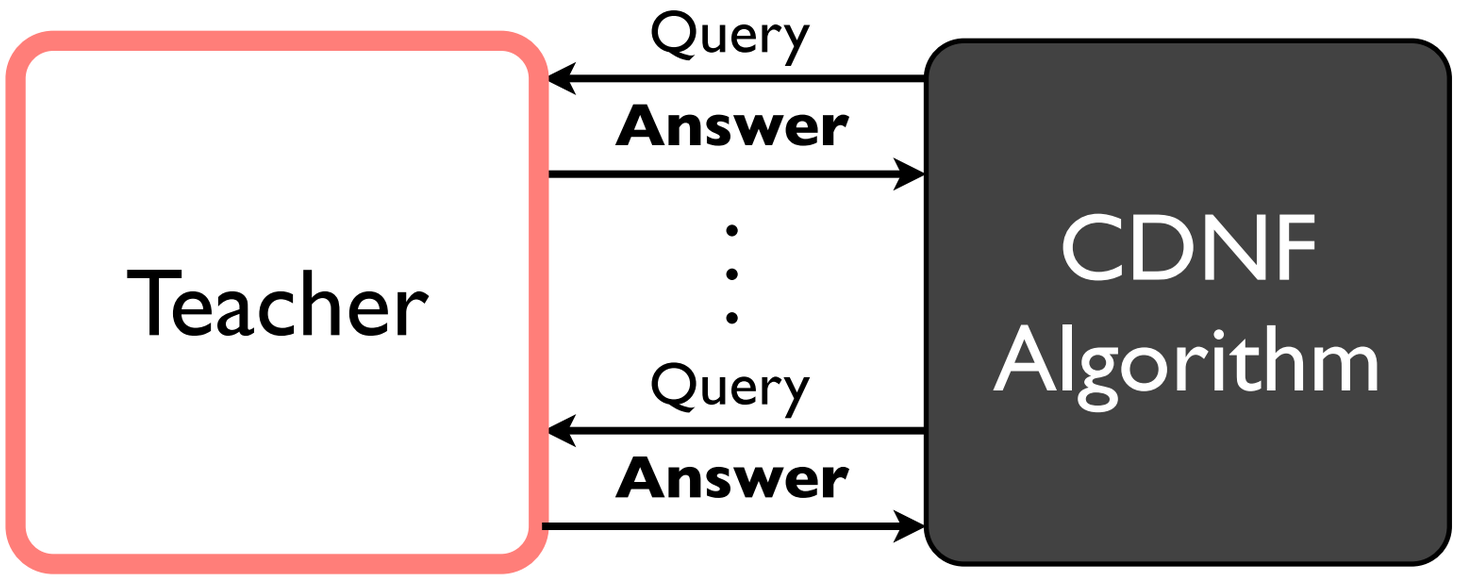
But, Boolean Formulae?



Algorithmic Learning with Predicate Abstraction



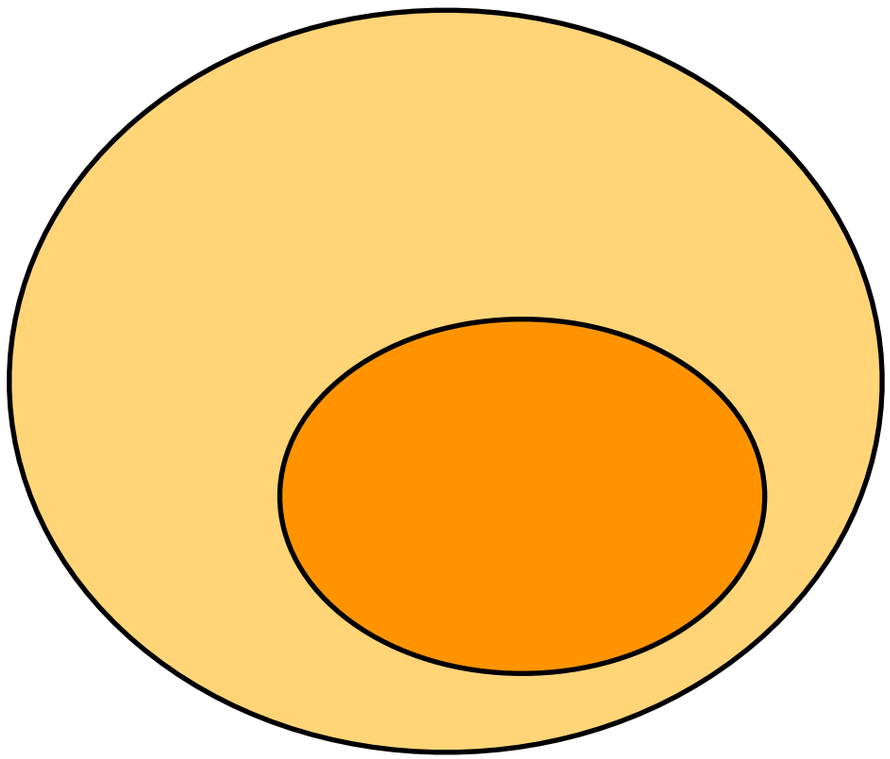
How about query resolution?



We have to answers about invariants
without knowing invariants!

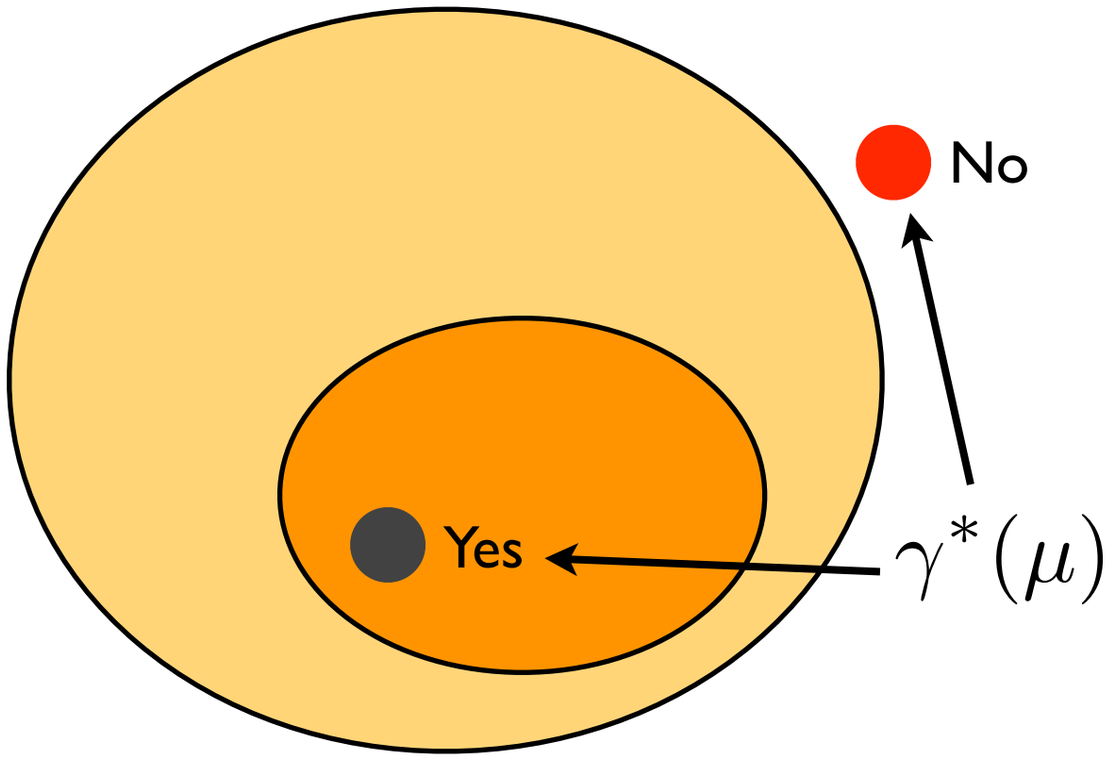
How about query resolution?

Search space of invariants



How about query resolution?

Membership query $MEM(\mu)$

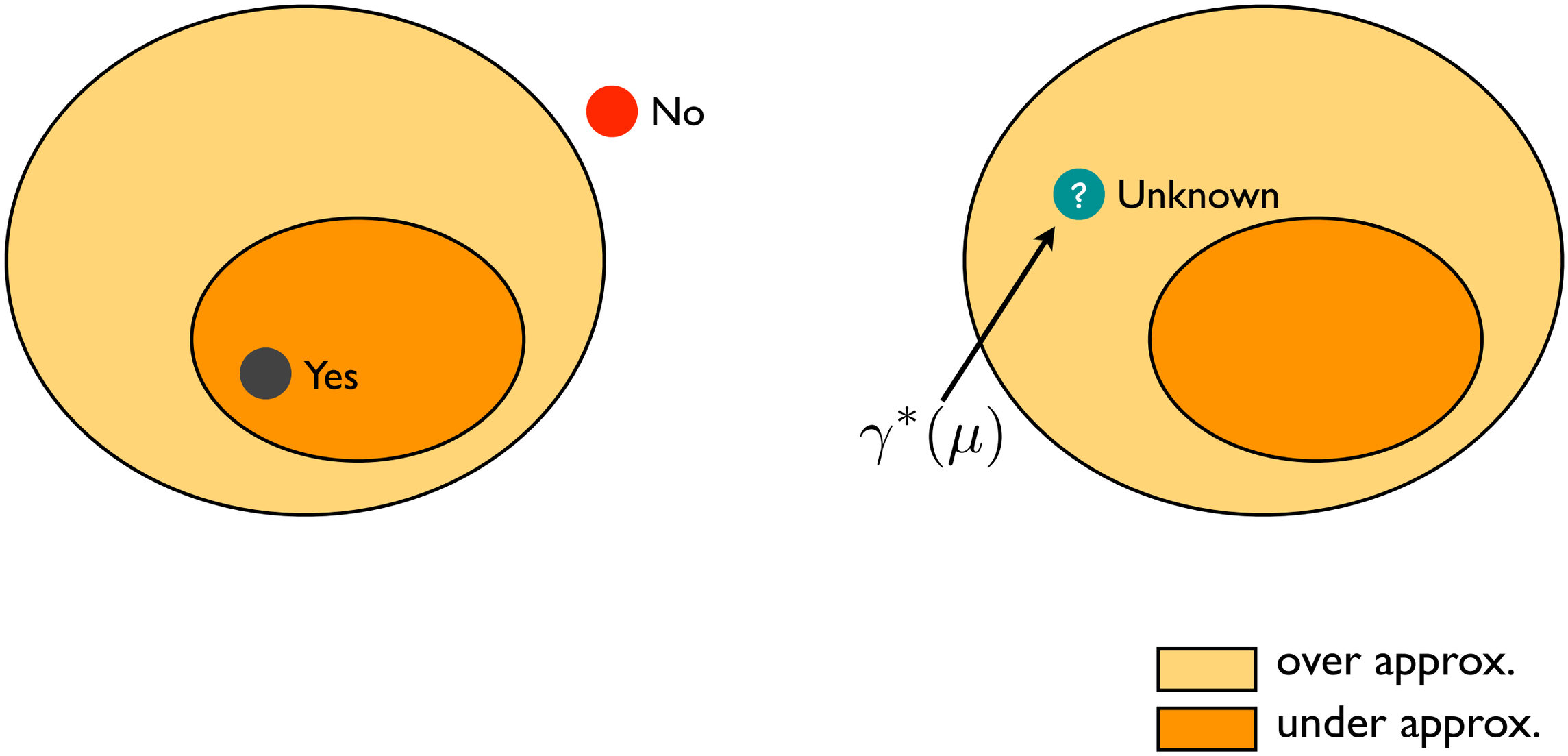


over approx.

under approx.

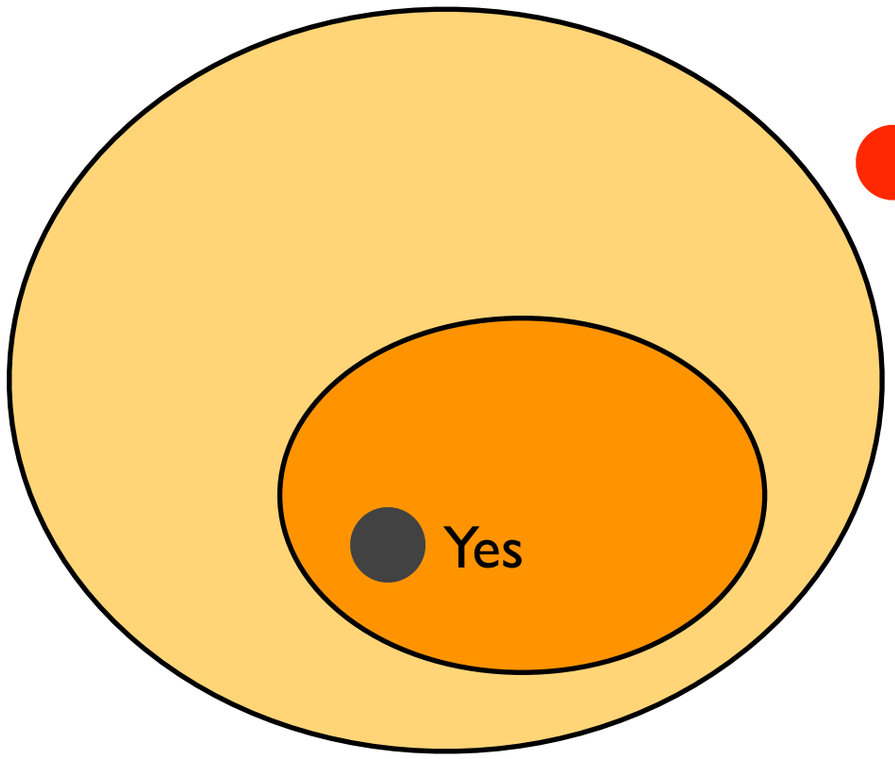
How about query resolution?

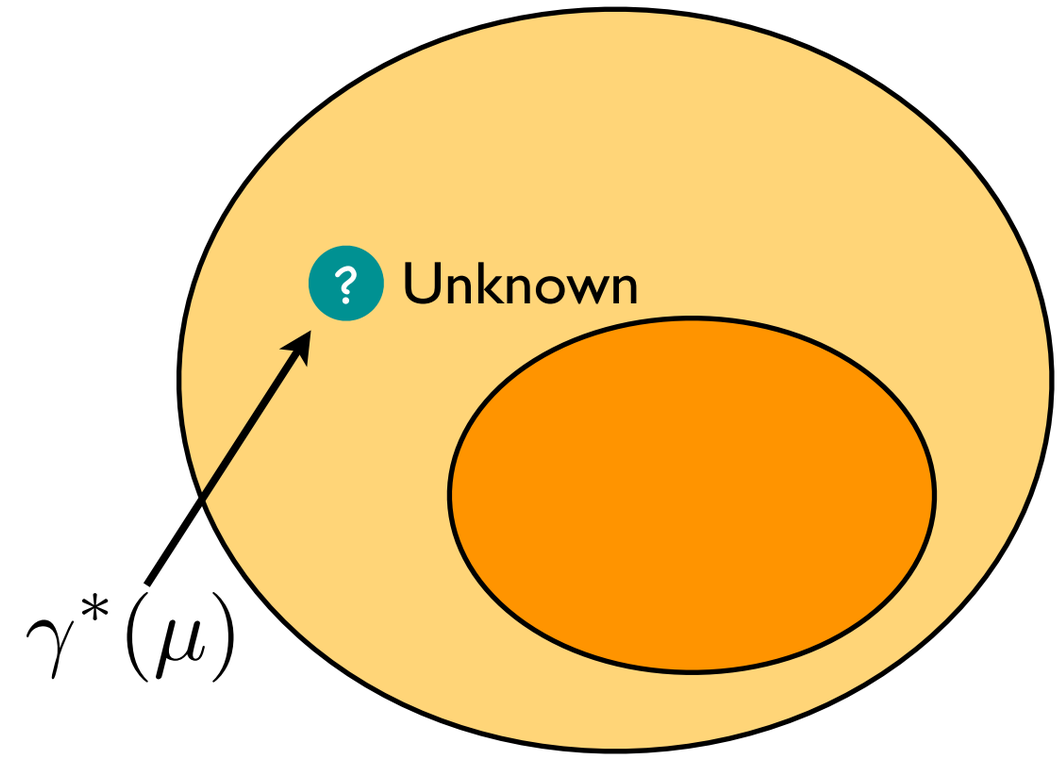
Membership query $MEM(\mu)$



How about query resolution?

Membership query $MEM(\mu)$





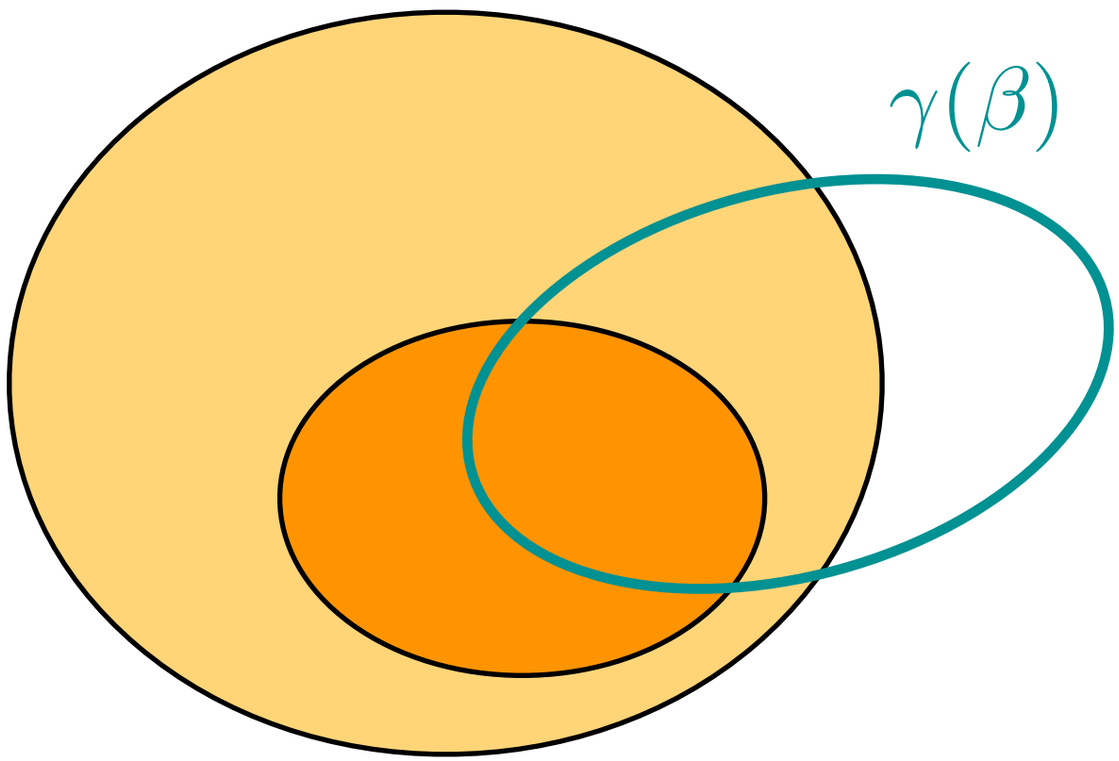
Random answer!

over approx.

under approx.

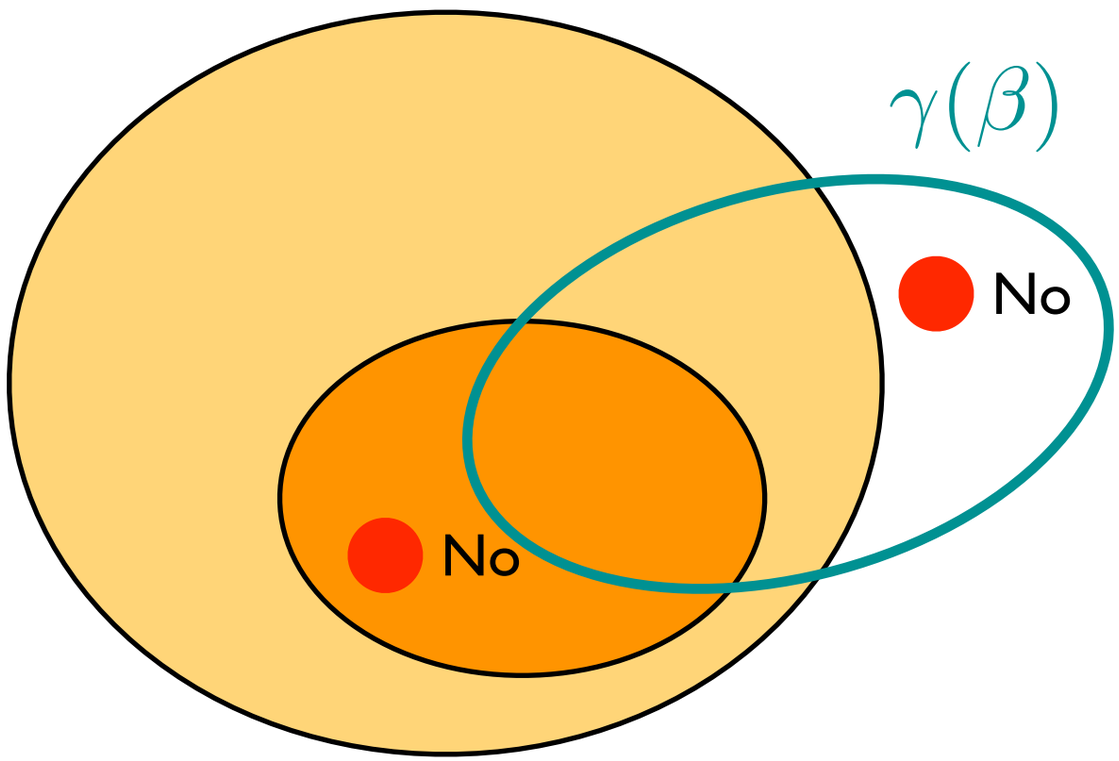
How about query resolution?

Equivalence query $EQ(\beta)$



How about query resolution?

Equivalence query $EQ(\beta)$

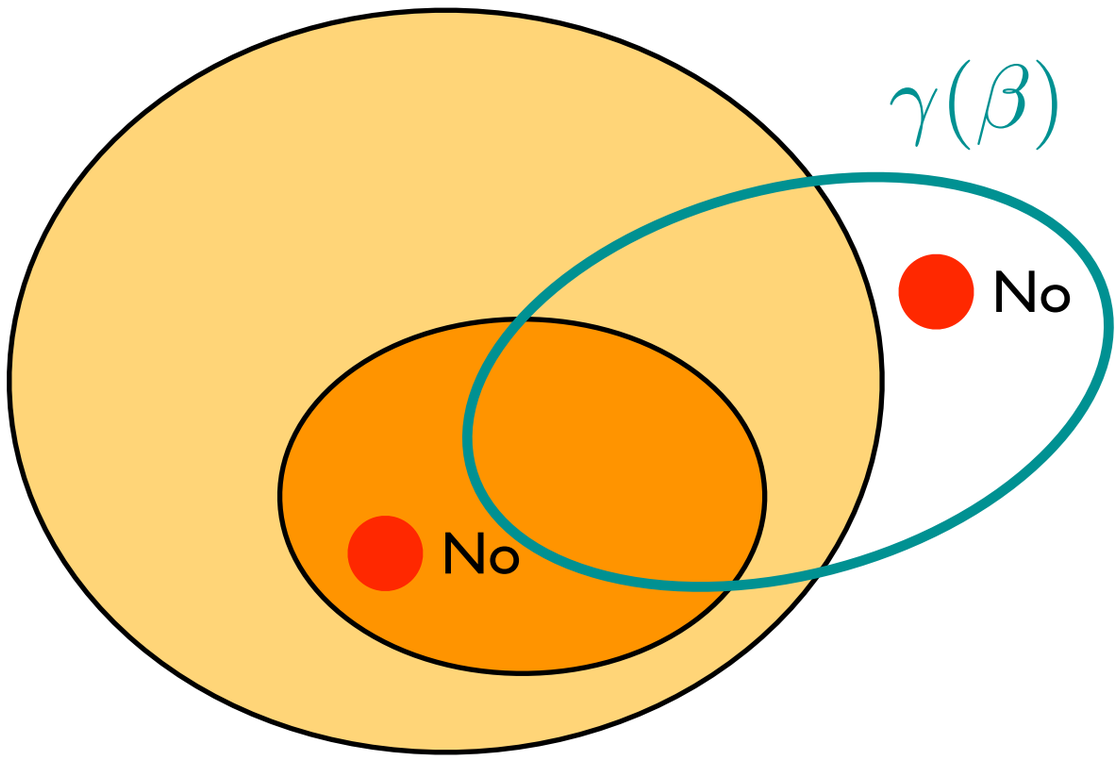


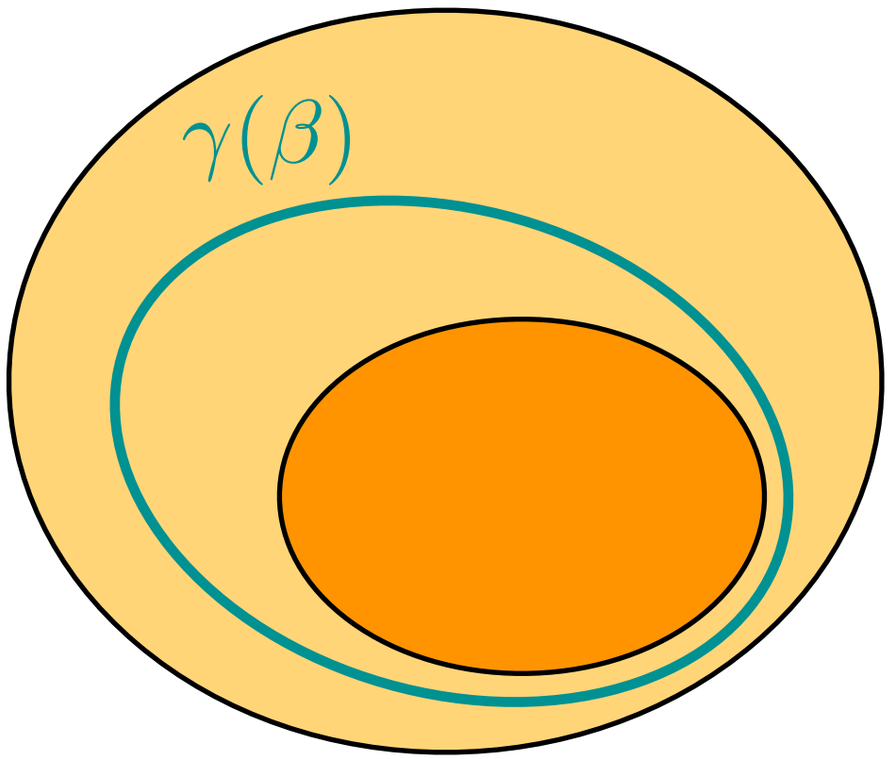
over approx.

under approx.

How about query resolution?

Equivalence query $EQ(\beta)$





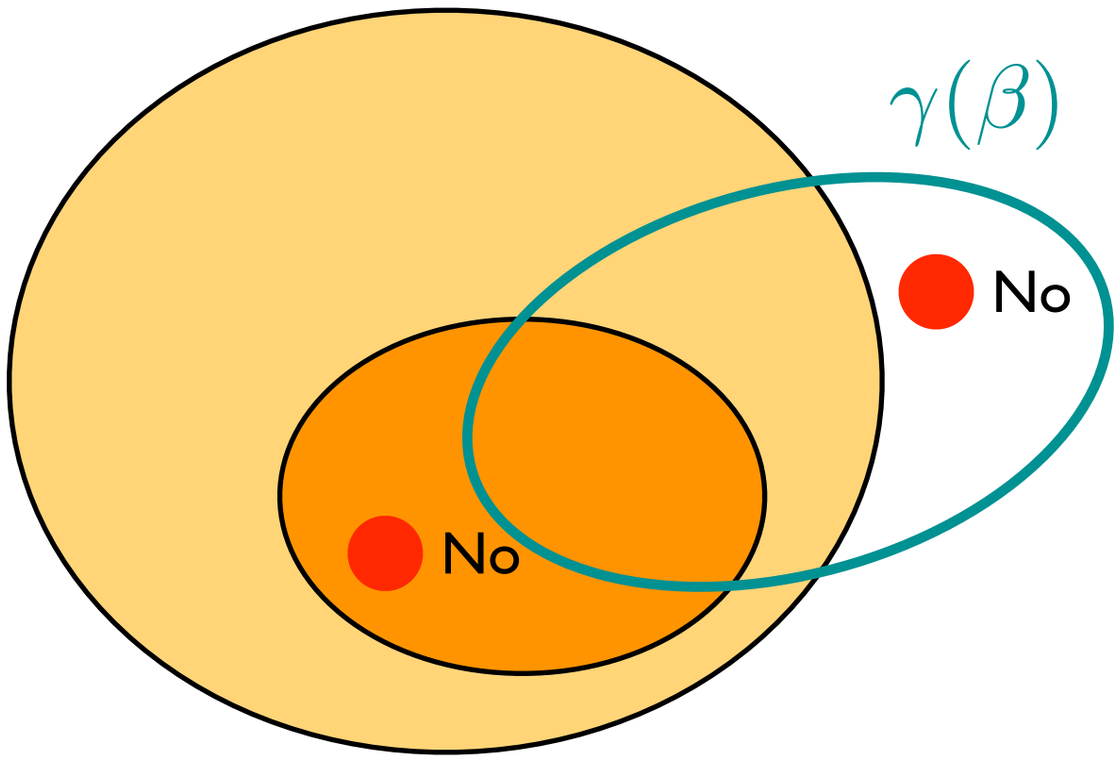
what if $\gamma(\beta) \wedge \rho \nRightarrow Pre(\gamma(\beta), S)$?

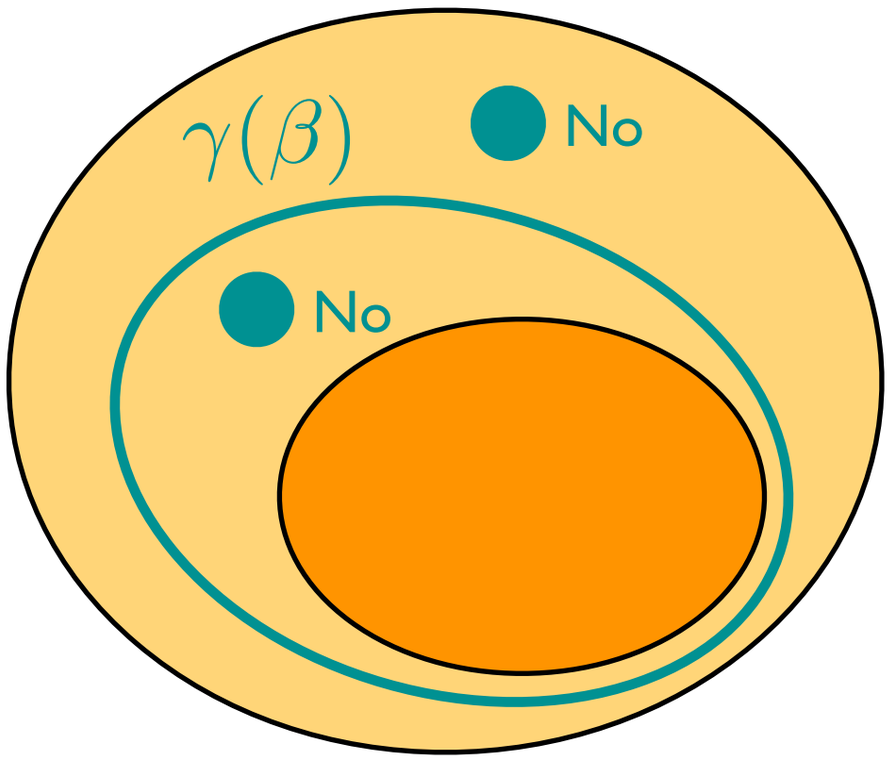
over approx.

under approx.

How about query resolution?

Equivalence query $EQ(\beta)$





Random counterexample!

over approx.

under approx.

VMCAI'10

Deriving Invariants by Algorithmic Learning, Decision Procedures, and Predicate Abstraction*

Yungbum Jung[1], Soonho Kong[1], Bow-Yaw Wang[2], and Kwangkeun Yi[1]

[1] School of Computer Science and Engineering, Seoul National University
{dreameye,soon,kwang}@ropas.snu.ac.kr
[2] Institute of Information Science, Academia Sinica
bywang@iis.sinica.edu.tw

Abstract. By combining algorithmic learning, decision procedures, and predicate abstraction, we present an automated technique for finding loop invariants in propositional formulae. Given invariant approximations derived from pre- and post-conditions, our new technique exploits the flexibility in invariants by a simple randomized mechanism. The proposed technique is able to generate invariants for some Linux device drivers and SPEC2000 benchmarks in our experiments.

1 Introduction

Algorithmic learning has been applied to assumption generation in compositional reasoning [9]. In contrast to traditional techniques, the learning approach does not derive assumptions in an off-line manner. It instead finds assumptions by interacting with a model checker progressively. Since assumptions in compositional reasoning are generally not unique, algorithmic learning can exploit the flexibility in assumptions to attain preferable solutions. Applications in formal verification and interface synthesis have also been reported [9, 1, 2, 18, 7].

Finding loop invariants follows a similar pattern. Invariants are often not unique. Indeed, programmers derive invariants incrementally. They usually have their guesses of invariants in mind, and gradually refine their guesses by observing program behavior more. Since in practice there are many invariants for given pre- and post-conditions, programmers have more freedom in deriving invariants. Yet traditional invariant generation techniques do not exploit the flexibility. They have a similar impediment to traditional assumption generation.

This article reports our first findings in applying algorithmic learning to invariant generation. We show that the three technologies (algorithmic learning,

* This work was supported by (A) the Engineering Research Center of Excellence Program of Korea Ministry of Education, Science and Technology(MEST) / Korea Science and Engineering Foundation(KOSEF) Grant Number R11-2008-007-01002-0, (B) the Brain Korea 21 Project, School of Electrical Engineering and Computer Science, Seoul National University, (C) SK Telecom, and (D) National Science Council of Taiwan Grant Numbers 95-2221-E-001-024-MY3 and 97-2221-E-001-006-MY3.

APLAS'10

Automatically Inferring Quantified Loop Invariants by Algorithmic Learning from Simple Templates

Soonho Kong[1], Yungbum Jung[1], Cristina David[2], Bow-Yaw Wang[3], and Kwangkeun Yi[1]

[1] Seoul National University
[2] National University of Singapore
[3] INRIA, Tsinghua University, and Academia Sinica

Abstract. By combining algorithmic learning, decision procedures, predicate abstraction, and simple templates, we present an automated technique for finding quantified loop invariants. Our technique can find arbitrary first-order invariants (modulo a fixed set of atomic propositions and an underlying SMT solver) in the form of the given template and exploits the flexibility in invariants by a simple randomized mechanism. The proposed technique is able to find quantified invariants for loops from the Linux source, as well as for the benchmark code used in the previous works. Our contribution is a simpler technique than the previous works yet with the same derivation power.

1 Introduction

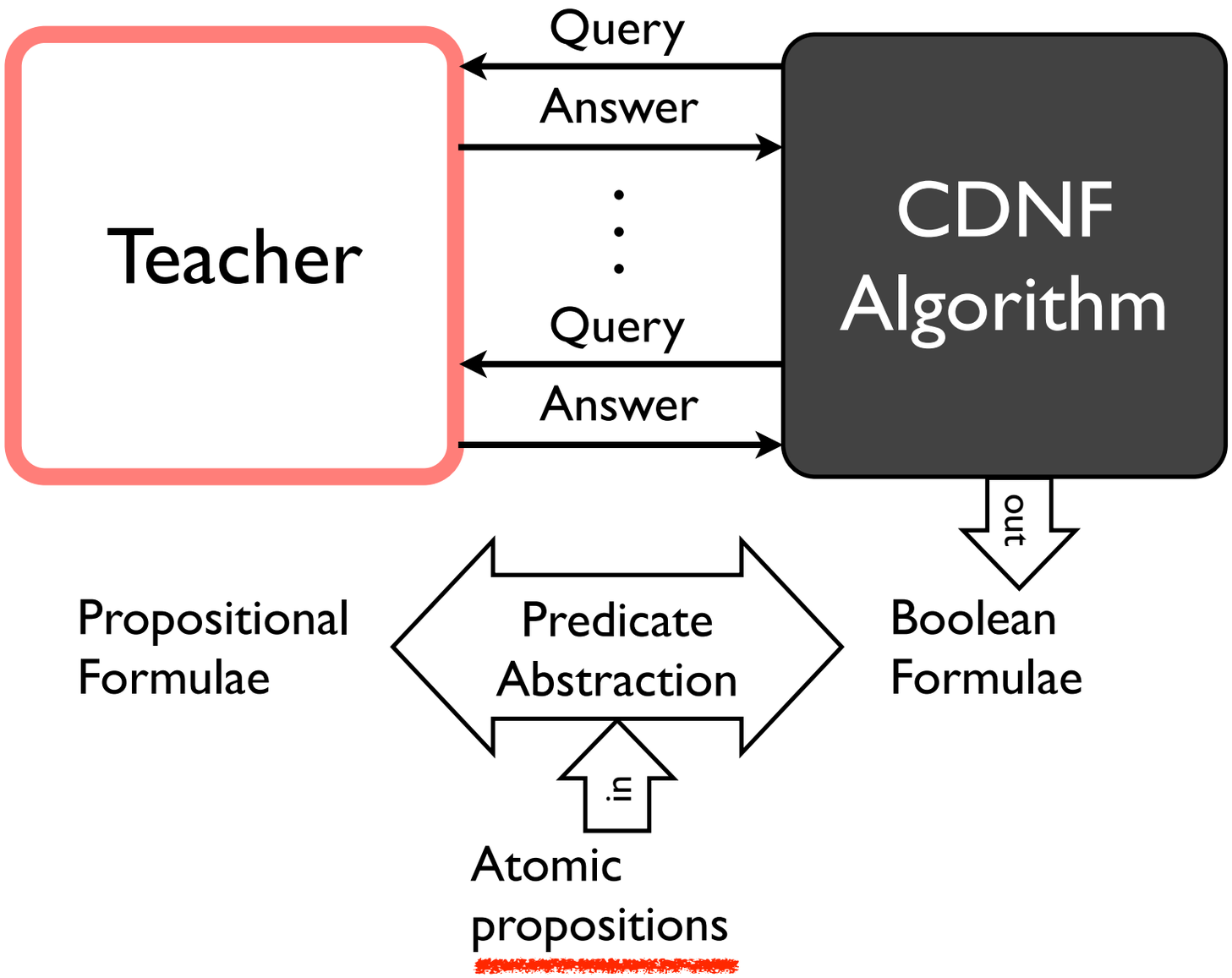
Recently, algorithmic learning has been successfully applied to invariant generation. The new approach formalizes the invariant generation problem as an instance of algorithmic learning: to generate an invariant is to learn a concept from a teacher. Using a learning algorithm as a black box, one only needs to design design a mechanical teacher that guides the learning algorithm to invariants. The learning-based framework not only simplifies the design of invariant generation algorithms, the new approach can also automatically generate invariants for realistic C loops at a reasonable cost [15].

Figure 1 shows the new framework proposed in [15]. In the figure, the CDNF algorithm is used to drive the search of quantifier-free invariants. The CDNF algorithm is an exact learning algorithm for Boolean formulae. It computes a representation of an unknown target formula by asking a teacher two types of queries. A membership query asks if a valuation to Boolean variables satisfies the unknown target; an equivalence query asks if a candidate formula is equivalent to the target. With predicate abstraction, the new approach formulates an unknown quantifier-free invariant as the unknown target Boolean formula. One only needs to automate the query resolution process to infer an invariant.

If an invariant was known, it would be easy to design a mechanical teacher to resolve queries. In the context of invariant generation, no invariant is known. However, a simple randomized automatic teacher is proposed in [15]. With the

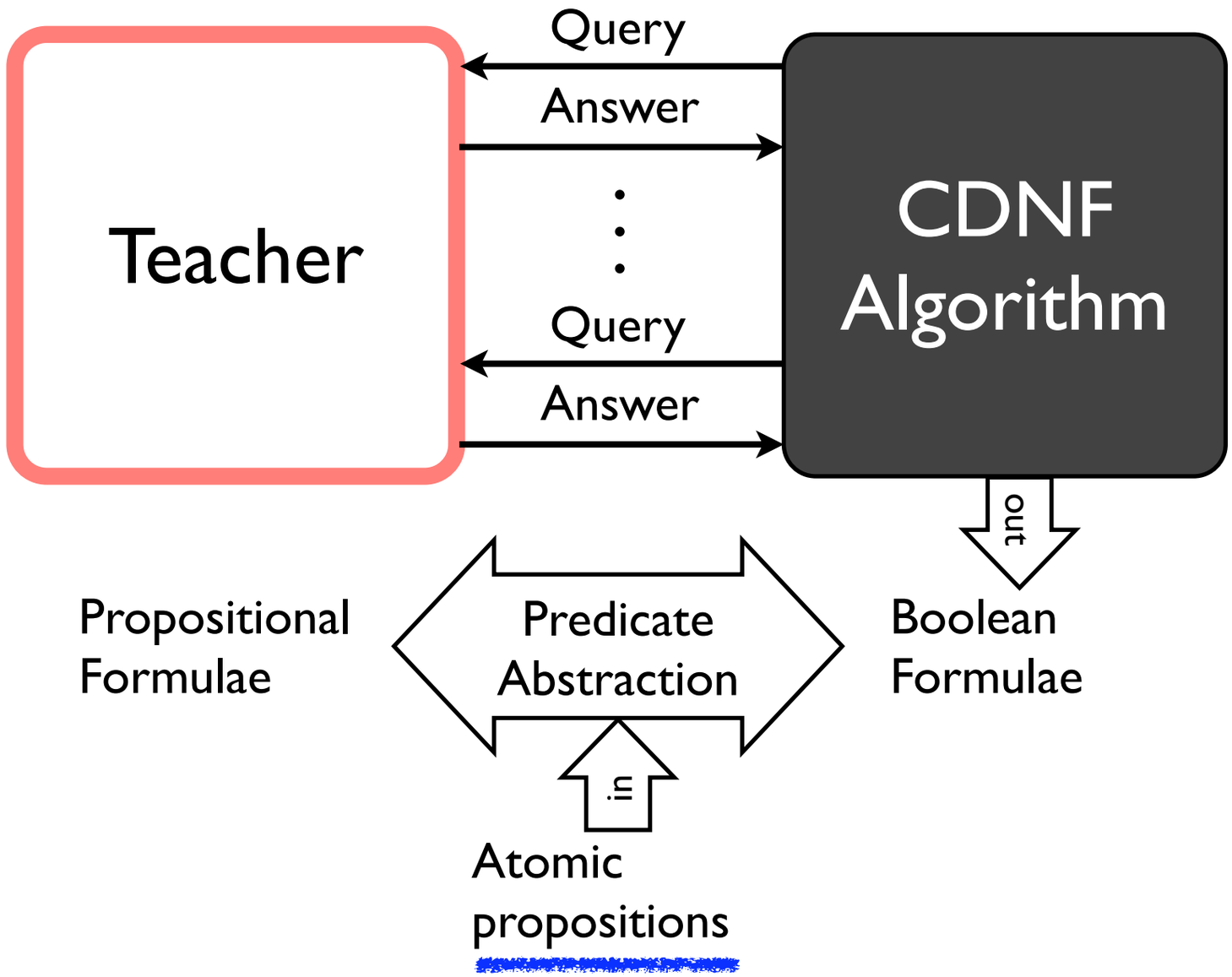
Generating Atomic Propositions

One Drawback of Previous Approach



manually chosen!

Our Thesis



can be generated using **interpolation**!

Why bothers?

Simply use all atomic propositions in the program.

Why bothers?

Simply use all atomic propositions in the program.

Counterexample:

```
{x = n /\ y = n /\ n >= 0}
while(x > 0) {
  x := x - 1;
  y := y - 1;
}
assert(x + y = 0)
```

Why bothers?

Simply use all atomic propositions in the program.

Counterexample:

```
{x = n /\ y = n /\ n >= 0}
while(x > 0) {
  x := x - 1;
  y := y - 1;
}
assert(x + y = 0)
```

Why bothers?

Simply use all atomic propositions in the program.

Counterexample:

```
{x = n /\ y = n /\ n >= 0}
while(x > 0) {
  x := x - 1;
  y := y - 1;
}
assert(x + y = 0)
```

We cannot generate any invariants even as simple as these: $x \geq 0$, $y \geq 0$

Why bothers?

Simply use all atomic propositions in the program.

Counterexample:

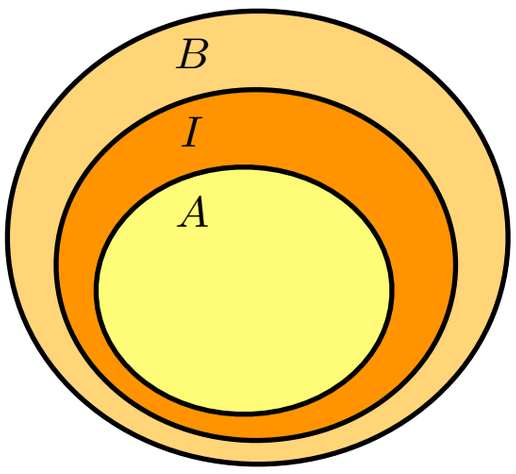
```
{x = n /\ y = n /\ n >= 0}
while(x > 0) {
  x := x - 1;
  y := y - 1;
}
assert(x + y = 0)
```

Invariant:

$x = y \wedge x \geq 0$

Interpolation could generate an atomic proposition like this: $x = y$

Craig's Interpolation Theorem†



Interpolant I of $A \Rightarrow B$ is defined as,

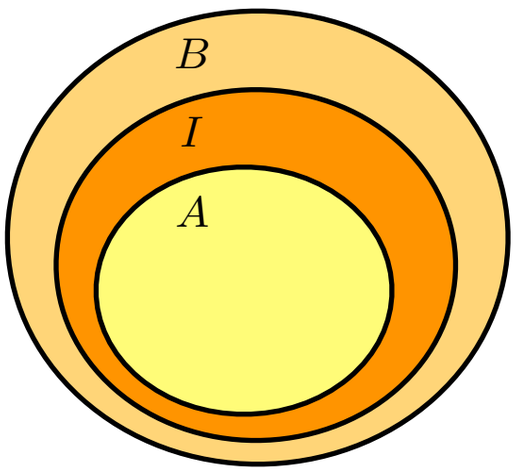
(A) $A \Rightarrow I$

(B) $I \Rightarrow B$

(C) $Var(I) \subseteq Var(A) \cap Var(B)$

†W. Craig, *Linear Reasoning. A New Form of the Herbrand-Gentzen Theorem*, The Journal of Symbolic Logic, Vol. 22, No. 3 (Sep., 1957), pp. 250–268

Craig's Interpolation Theorem†



Interpolant I of $A \Rightarrow B$ is defined as,

(A) $A \Rightarrow I$

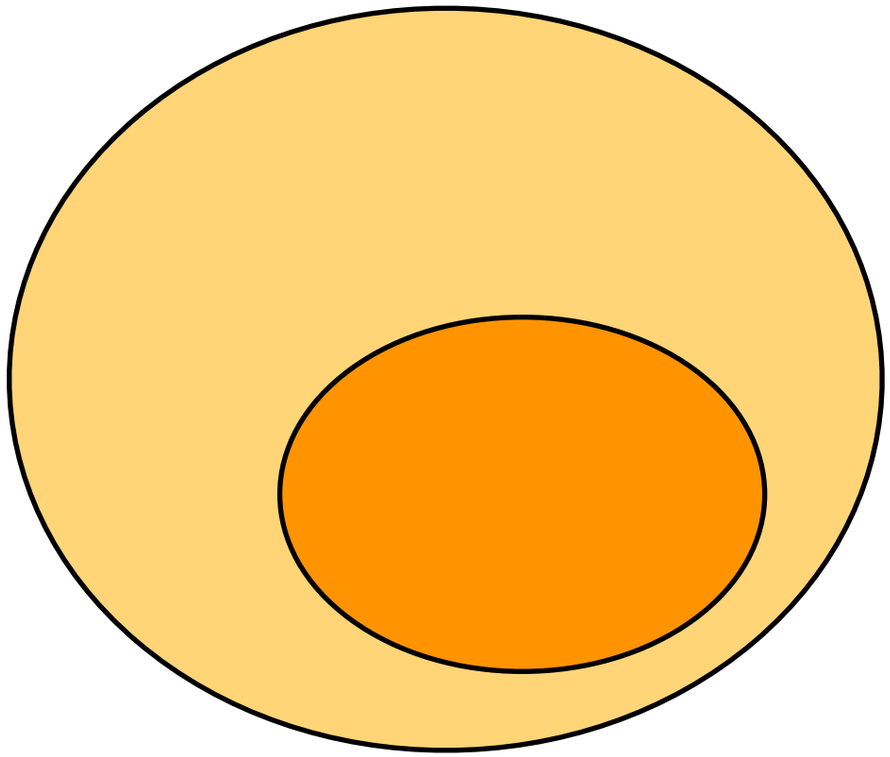
(B) $I \Rightarrow B$

(C) $Var(I) \subseteq Var(A) \cap Var(B)$

†W. Craig, *Linear Reasoning. A New Form of the Herbrand-Gentzen Theorem*, The Journal of Symbolic Logic, Vol. 22, No. 3 (Sep., 1957), pp. 250–268

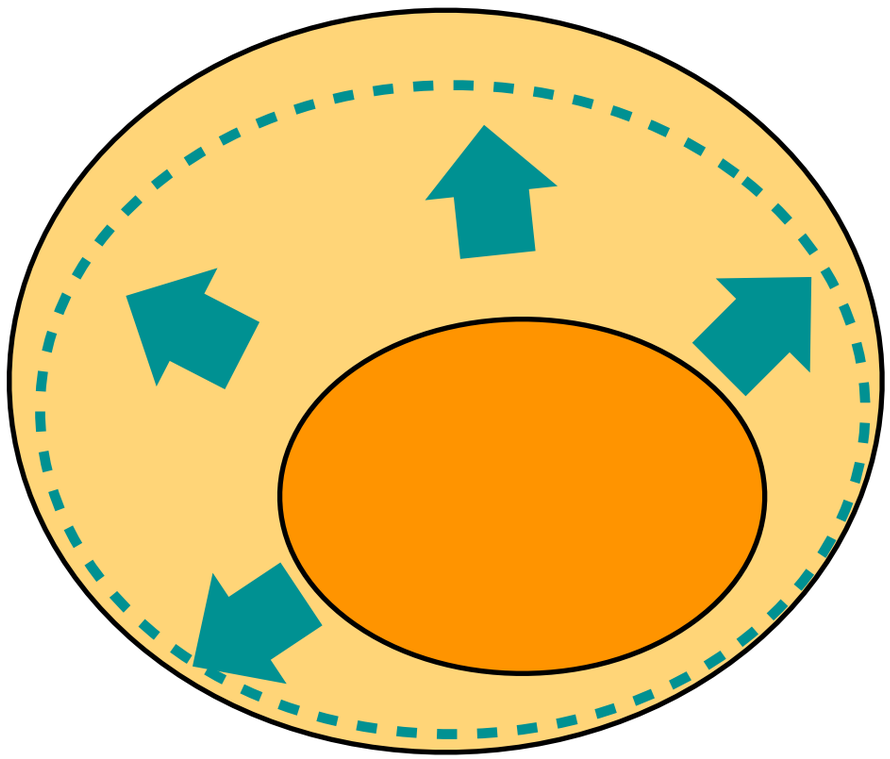
Initial Atomic Propositions

Interpolation using $\underline{\iota} \Rightarrow \overline{\iota}$



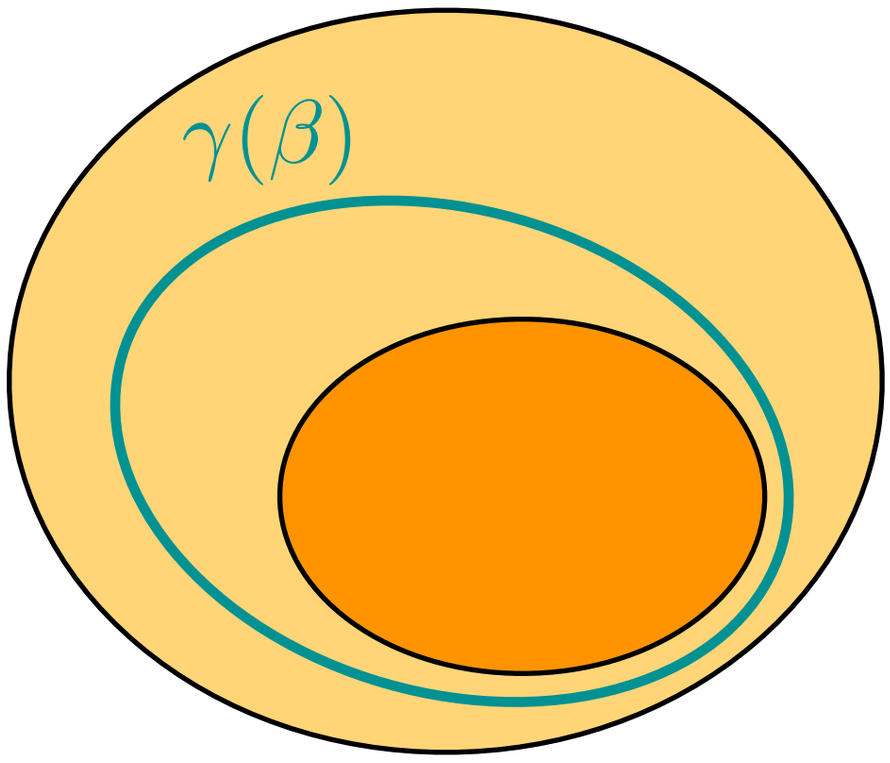
Initial Atomic Propositions

Interpolation using $\underline{\iota} \Rightarrow \overline{\iota}$



Interpolating Failed Conjecture

Equivalence query $EQ(\beta)$



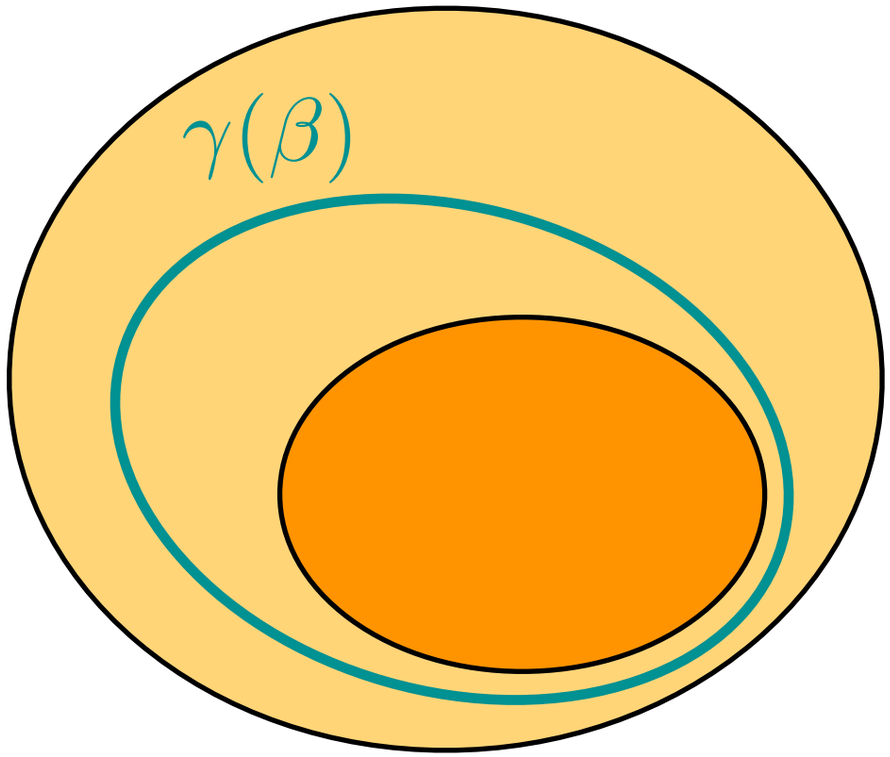
$\gamma(\beta) \wedge \rho \Rightarrow Pre(\gamma(\beta), S)$

over approx.

under approx.

Interpolating Failed Conjecture

Equivalence query $EQ(\beta)$



$\gamma(\beta) \wedge \rho \Rightarrow Pre(\gamma(\beta), S)$

Two possibilities

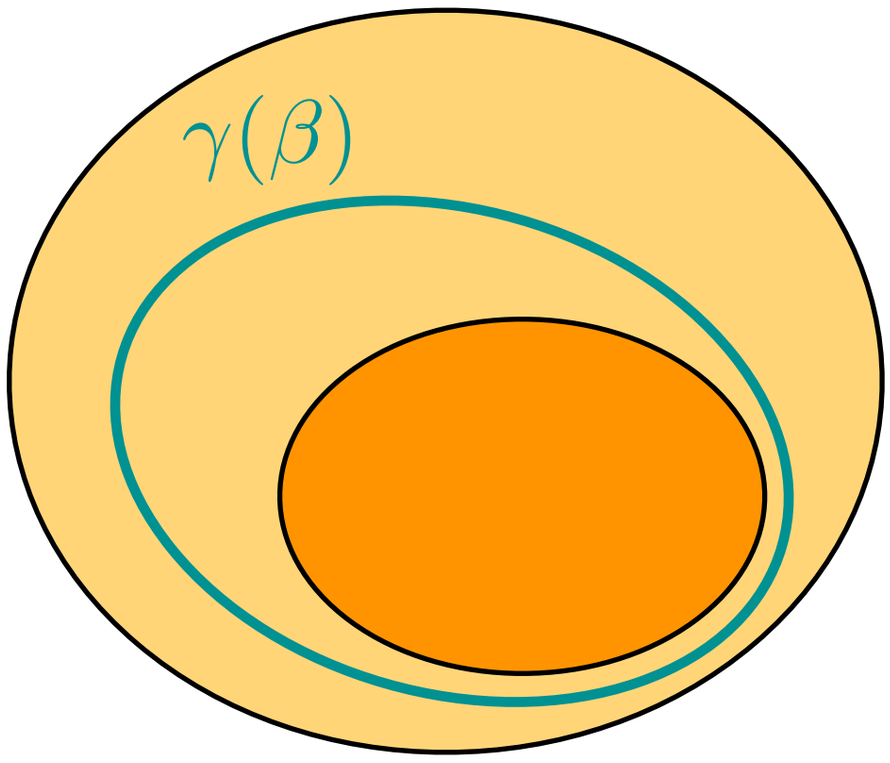
(A) sufficient atomic propositions
(in the middle of computation)

(B) insufficient atomic propositions
➡ need to generate more!

over approx.
under approx.

Interpolating Failed Conjecture

Equivalence query $EQ(\beta)$



$\gamma(\beta) \Rightarrow \bar{\iota}$

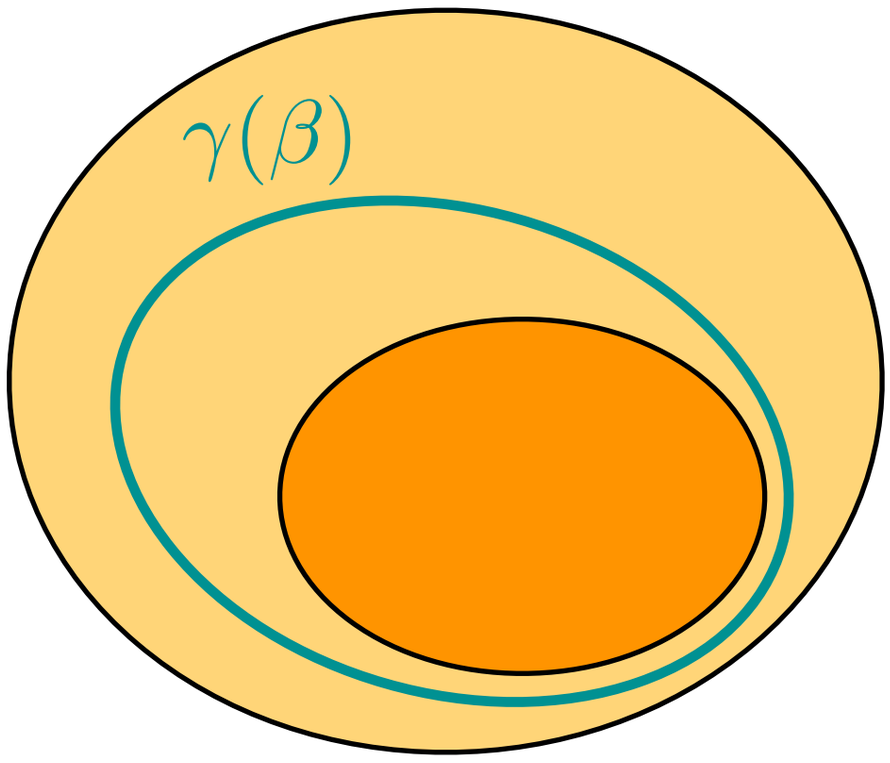
$\gamma(\beta) \wedge \rho \nRightarrow Pre(\gamma(\beta), S)$

over approx.

under approx.

Interpolating Failed Conjecture

Equivalence query $EQ(\beta)$



$\gamma(\beta) \Rightarrow \bar{\iota}$

Add no information

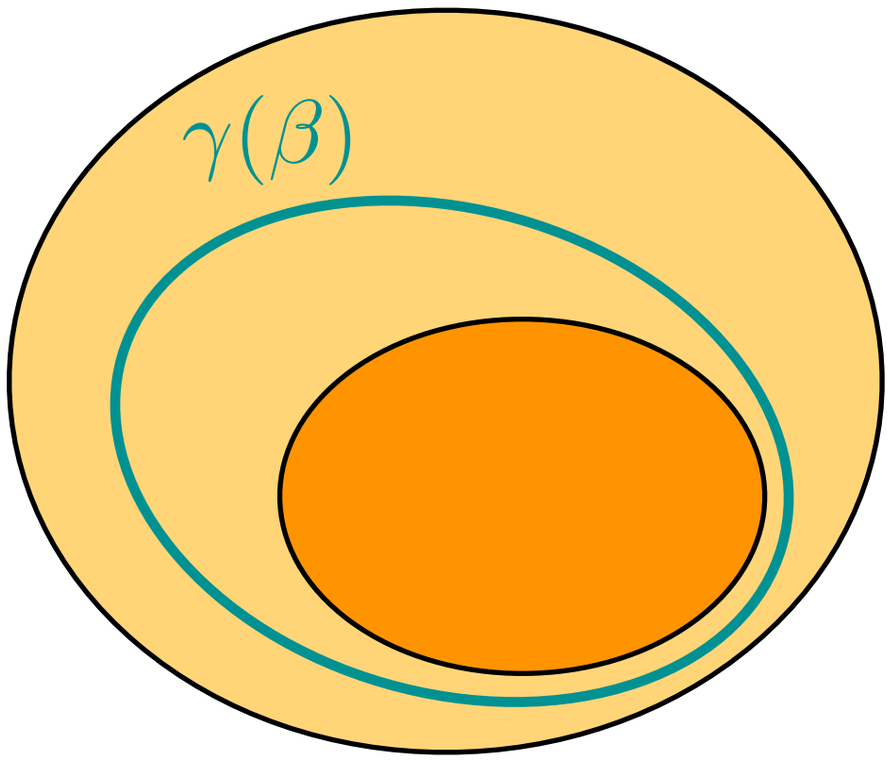
$\gamma(\beta) \wedge \rho \nRightarrow Pre(\gamma(\beta), S)$

over approx.

under approx.

Interpolating Failed Conjecture

Equivalence query $EQ(\beta)$



$\gamma(\beta) \wedge \rho \wedge [\![S]\!] \Rightarrow ?$

transition formula

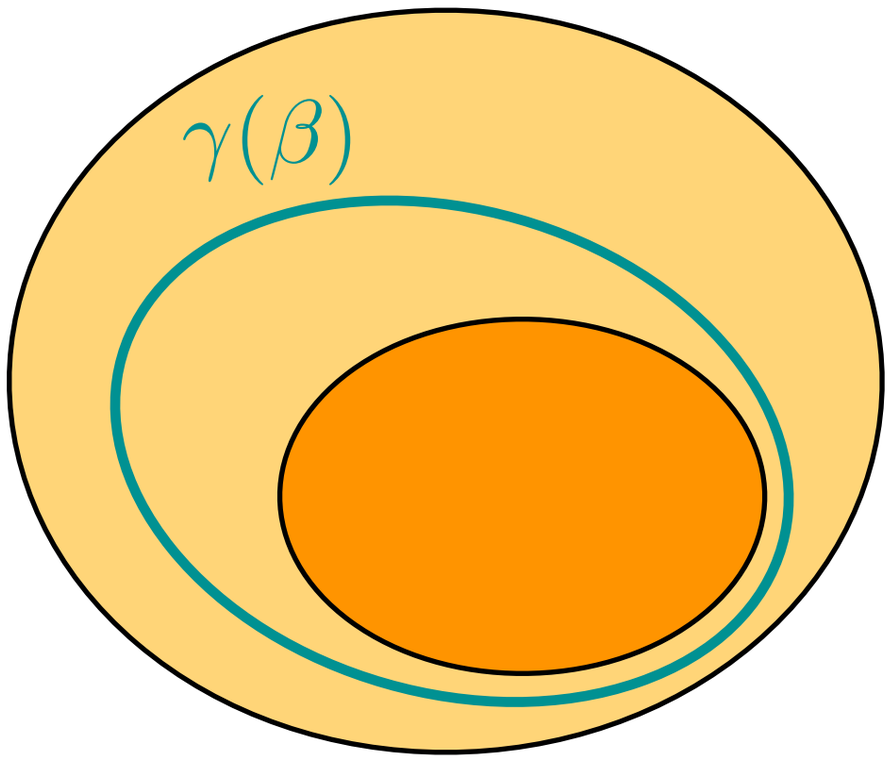
$\gamma(\beta) \wedge \rho \nRightarrow Pre(\gamma(\beta), S)$

over approx.

under approx.

Interpolating Failed Conjecture

Equivalence query $EQ(\beta)$



$$\gamma(\beta) \wedge \rho \wedge [\![S]\!] \Rightarrow \bar{\imath}'$$

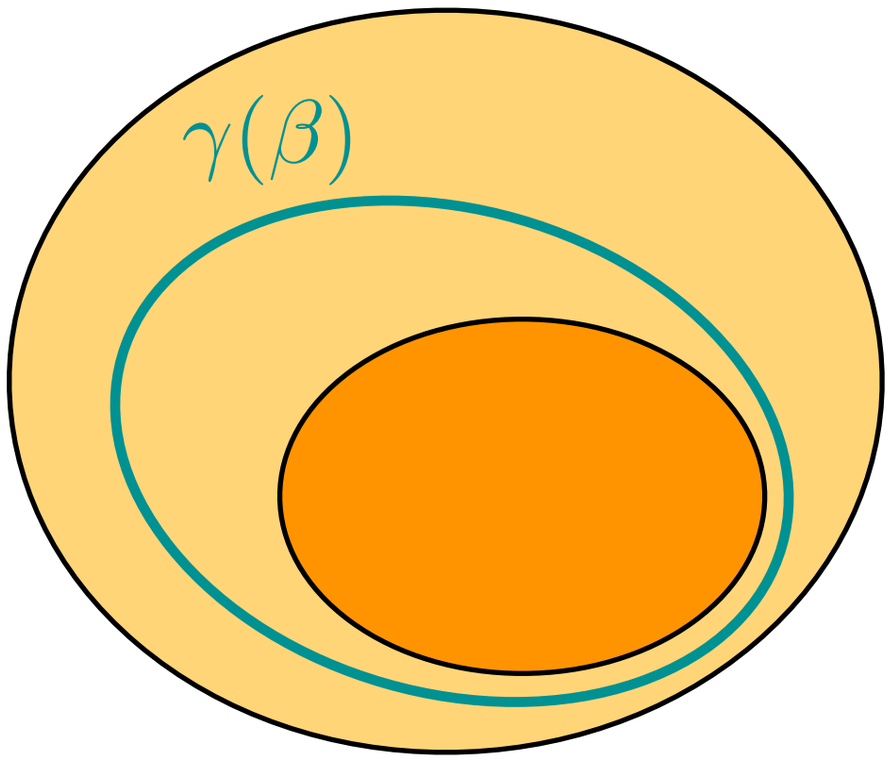
$$\gamma(\beta) \wedge \rho \nRightarrow Pre(\gamma(\beta), S)$$

over approx.

under approx.

Interpolating Failed Conjecture

Equivalence query $EQ(\beta)$



$\gamma(\beta) \wedge \rho \nRightarrow Pre(\gamma(\beta), S)$

$\gamma(\beta) \wedge \rho \wedge [\![S]\!] \Rightarrow \bar{\iota}'$

For every loop invariant ι

$\iota \wedge \rho \Rightarrow Pre(\iota, S)$ and $\iota \Rightarrow \bar{\iota}$

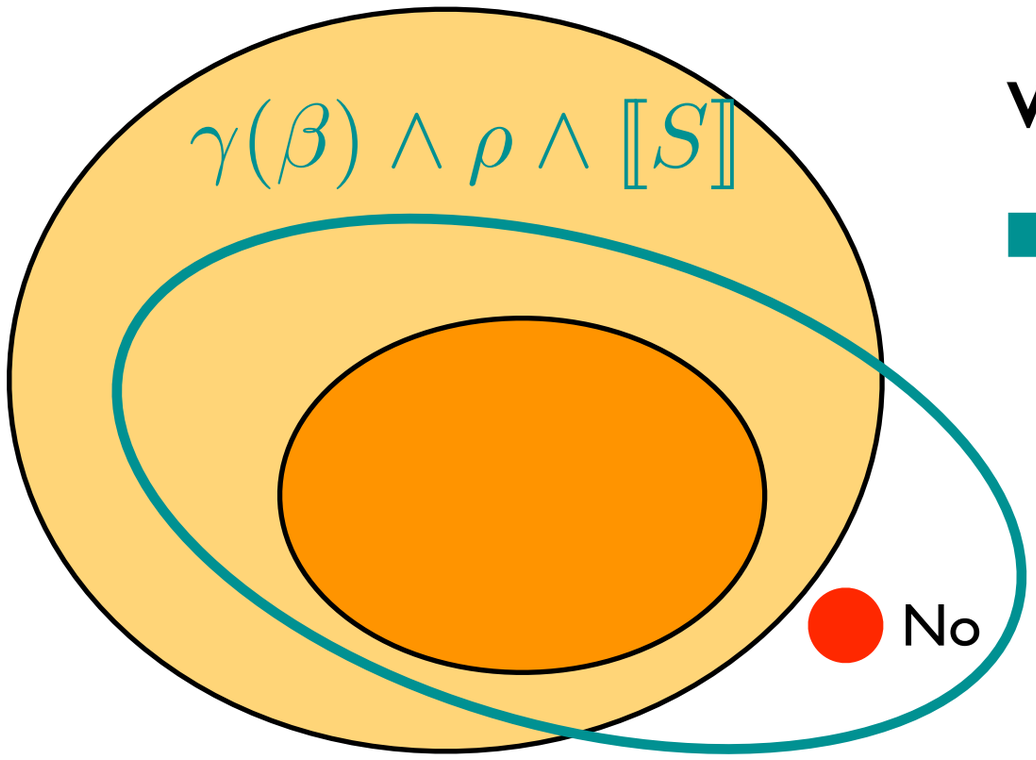
thus, $\iota \wedge \rho \Rightarrow Pre(\bar{\iota}, S)$

over approx.

under approx.

Interpolating Failed Conjecture

Equivalence query $EQ(\beta)$



what if $\gamma(\beta) \wedge \rho \wedge [\![S]\!] \not\Rightarrow \bar{\iota}'$ **?**

➡ we can give a correct counterexample here!

over approx.

under approx.

Experiment Results

case	$SIZE$	FOCI					CSIsat				
		AP	MEM	EQ	RE	$Time(s)$	AP	MEM	EQ	RE	$Time(s)$
`ide-ide-tape`	16	6	9	5	1	0.05	6	9	5	1	0.05
`ide-wait-ireason`	9	5	120	89	7	1.02	5	130	96	7	1.24
`parser`	37	12	118	33	1	0.56	12	119	33	1	0.56
`usb-message`	18	3	7	6	1	0.03	3	7	6	1	0.04
`vpr`	8	1	1	3	1	0.01	1	1	3	1	0.01

case	$SIZE$	AP	MEM	EQ	coin tossing	iterations	time (sec)
`ide-ide-tape`	16	6	18.2	5.2	4.1	1.2	0.055
`ide-wait-ireason`	9	6	216.1	111.8	47.2	9.9	0.602
`parser`	37	20	6694.5	819.4	990.3	12.5	32.120
`usb-message`	18	10	20.1	6.8	1.0	1.0	0.128
`vpr`	8	7	14.5	8.9	11.8	2.9	0.055

Table 2. VMCAI'10 Performance Numbers, the average of 500 runs

Experiment Results

case	$SIZE$	FOCI					CSIsat				
		AP	MEM	EQ	RE	$Time(s)$	AP	MEM	EQ	RE	$Time(s)$
`ide-ide-tape`	16	6	9	5	1	0.05	6	9	5	1	0.05
`ide-wait-ireason`	9	5	120	89	7	1.02	5	130	96	7	1.24
`parser`	37	12	118	33	1	0.56	12	119	33	1	0.56
`usb-message`	18	3	7	6	1	0.03	3	7	6	1	0.04
`vpr`	8	1	1	3	1	0.01	1	1	3	1	0.01

case	$SIZE$	AP	MEM	EQ	coin tossing	iterations	time (sec)
`ide-ide-tape`	16	6	18.2	5.2	4.1	1.2	0.055
`ide-wait-ireason`	9	6	216.1	111.8	47.2	9.9	0.602
`parser`	37	20	6694.5	819.4	990.3	12.5	32.120
`usb-message`	18	10	20.1	6.8	1.0	1.0	0.128
`vpr`	8	7	14.5	8.9	11.8	2.9	0.055

Table 2. VMCAI'10 Performance Numbers, the average of 500 runs

Experiment Results

case	$SIZE$	FOCI					CSIsat				
		AP	MEM	EQ	RE	$Time(s)$	AP	MEM	EQ	RE	$Time(s)$
`ide-ide-tape`	16	6	9	5	1	0.05	6	9	5	1	0.05
`ide-wait-ireason`	9	5	120	89	7	1.02	5	130	96	7	1.24
`parser`	37	12	118	33	1	0.56	12	119	33	1	0.56
`usb-message`	18	3	7	6	1	0.03	3	7	6	1	0.04
`vpr`	8	1	1	3	1	0.01	1	1	3	1	0.01

case	$SIZE$	AP	MEM	EQ	coin tossing	iterations	time (sec)
`ide-ide-tape`	16	6	18.2	5.2	4.1	1.2	0.055
`ide-wait-ireason`	9	6	216.1	111.8	47.2	9.9	0.602
`parser`	37	20	6694.5	819.4	990.3	12.5	32.120
`usb-message`	18	10	20.1	6.8	1.0	1.0	0.128
`vpr`	8	7	14.5	8.9	11.8	2.9	0.055

Table 2. VMCAI'10 Performance Numbers, the average of 500 runs

Thank you!