SYNTHESIZING ABSTRACT TRANSFORMERS

OOPSLA ‘22

PANKAJ KUMAR KALITA, Indian Institute of Technology Kanpur, India
SUJIT KUMAR MUDULI, Indian Institute of Technology Kanpur, India
LORIS D’ANTONI, University of Wisconsin–Madison, USA
THOMAS REPS, University of Wisconsin–Madison, USA
SUBHAJIT ROY, Indian Institute of Technology Kanpur, India

Presentor: Shaurya Gomber (1st yr MS CS, UIUC)

Topics

1. What is Abstract Interpretation?
2. What are Abstract Transformers?
3. Soundness and Precision of Abstract Transformers
4. AMURTH: The Abstract Transformer Synthesizer
    a. High level idea
    b. Working
    c. Results

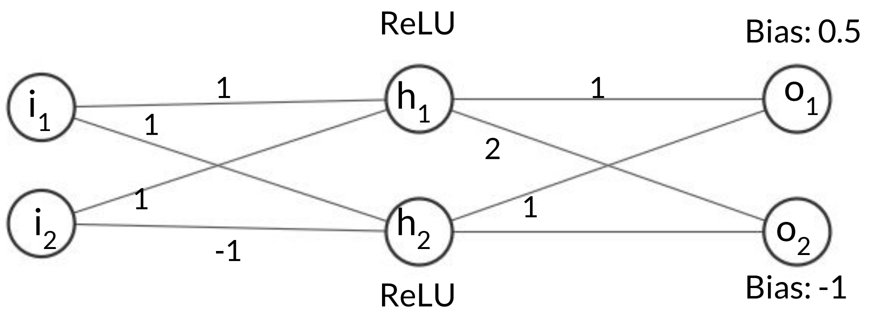
Introduction

- **Static Analysis**: Method of reasoning (verifying, debugging ..) about computer programs without *explicitly* executing them.

- **Abstract Interpretation**: A static-analysis framework that guarantees that the information gathered about a program is a safe approximation to the program's semantics.

- **Basic Idea:** Approximate the program's behavior by using an abstract domain, which is a simplified representation of the values that the program can manipulate.

Example




```
def f(i1, i2):
    h1 = i1 + i2
    h2 = i1 - i2
    h1 = max(0, h1)
    h2 = max(0, h2)
    o1 = h1 + h2 + 0.5
    o2 = 2*h1 - h2 - 1
    return o1, o2
```

Prove that if i_1 ranges from [0, 0.3] and i_2 ranges from [0.1, 0.4], then $o_2 \geq -1$

Example

Let's try and reason about this by keeping track of the lower and upper bounds of the variables!

```
def f(i1, i2):

    h1 = i1 + i2

    h2 = i1 - i2

    h1 = max(0, h1)

    h2 = max(0, h2)

    o1 = h1 + h2 + 0.5

    o2 = 2*h1 - h2 - 1

    return o1, o2
```

Example

Let's try and reason about this by keeping track of the lower and upper bounds of the variables!

```
def f(i1, i2):
```

{i_1 -> [0, 0.3], i_2 -> [0.1, 0.4]}   (Given)

```
    h1 = i1 + i2

    h2 = i1 - i2

    h1 = max(0, h1)

    h2 = max(0, h2)

    o1 = h1 + h2 + 0.5

    o2 = 2*h1 - h2 - 1

    return o1, o2
```

Example

Let's try and reason about this by keeping track of the lower and upper bounds of the variables!

```
def f(i1, i2):
    {i1 -> [0, 0.3], i2 -> [0.1, 0.4]}        (Given)
    h1 = i1 + i2
    {h1 -> [0.1, 0.7]}        (To add intervals, add the lower bounds and upper bounds)
    h2 = i1 - i2

    h1 = max(0, h1)

    h2 = max(0, h2)

    o1 = h1 + h2 + 0.5

    o2 = 2*h1 - h2 - 1

    return o1, o2
```

Example

Let's try and reason about this by keeping track of the lower and upper bounds of the variables!

```
def f(i1, i2):
```

{i_1 -> [0, 0.3], i_2 -> [0.1, 0.4]} (Given)

```
    h1 = i1 + i2
```

{h_1 -> [0.1, 0.7]} (To add intervals, add the lower bounds and upper bounds)

```
    h2 = i1 - i2
```

{h_2 -> [-0.4, 0.2]} (-i_2 -> [-0.4, -0.1] and $i_1 - i_2 = i_1 + (-i_2)$)

```
    h1 = max(0, h1)

    h2 = max(0, h2)

    o1 = h1 + h2 + 0.5

    o2 = 2*h1 - h2 - 1

    return o1, o2
```

Example

Let's try and reason about this by keeping track of the lower and upper bounds of the variables!

```
def f(i1, i2):
```

$\{i_1 \text{ -> } [0, 0.3],\ i_2 \text{ -> } [0.1, 0.4]\}$ (Given)

```
    h1 = i1 + i2
```

$\{h_1 \text{ -> } [0.1, 0.7]\}$ (To add intervals, add the lower bounds and upper bounds)

```
    h2 = i1 - i2
```

$\{h_2 \text{ -> } [-0.4, 0.2]\}$ ($-i_2$ -> $[-0.4, -0.1]$ and $i_1 - i_2 = i_1 + (-i_2)$)

```
    h1 = max(0, h1)
```

$\{h_1 \text{ -> } [0.1, 0.7]\}$ (Max has no effect on h_1 as it is already more than 0)

```
    h2 = max(0, h2)

    o1 = h1 + h2 + 0.5

    o2 = 2*h1 - h2 - 1

    return o1, o2
```

Example

Let's try and reason about this by keeping track of the lower and upper bounds of the variables!

```
def f(i1, i2):
```
{i_1 -> [0, 0.3], i_2 -> [0.1, 0.4]} (Given)
```
    h1 = i1 + i2
```
{h_1 -> [0.1, 0.7]} (To add intervals, add the lower bounds and upper bounds)
```
    h2 = i1 - i2
```
{h_2 -> [-0.4, 0.2]} (-i_2 -> [-0.4, -0.1] and $i_1 - i_2 = i_1 + (-i_2)$)
```
    h1 = max(0, h1)
```
{h_1 -> [0.1, 0.7]} (Max has no effect on h_1 as it is already more than 0)
```
    h2 = max(0, h2)
```
{h_2 -> [0, 0.2]} (Max prunes the negative part from h_2)
```
    o1 = h1 + h2 + 0.5

    o2 = 2*h1 - h2 - 1

    return o1, o2
```

Example

Let's try and reason about this by keeping track of the lower and upper bounds of the variables!

```
def f(i1, i2):
```

$\{i_1 \text{ -> } [0, 0.3],\ i_2 \text{ -> } [0.1, 0.4]\}$ (Given)

```
    h1 = i1 + i2
```

$\{h_1 \text{ -> } [0.1, 0.7]\}$ (To add intervals, add the lower bounds and upper bounds)

```
    h2 = i1 - i2
```

$\{h_2 \text{ -> } [-0.4, 0.2]\}$ ($-i_2 \text{ -> } [-0.4, -0.1]$ and $i_1 - i_2 = i_1 + (-i_2)$)

```
    h1 = max(0, h1)
```

$\{h_1 \text{ -> } [0.1, 0.7]\}$ (Max has no effect on h_1 as it is already more than 0)

```
    h2 = max(0, h2)
```

$\{h_2 \text{ -> } [0, 0.2]\}$ (Max prunes the negative part from h_2)

```
    o1 = h1 + h2 + 0.5
```

$\{o_1 \text{ -> } [0.6, 1.4]\}$ ($[0.1 + 0 + 0.5,\ 0.7 + 0.2 + 0.5]$)

```
    o2 = 2*h1 - h2 - 1

    return o1, o2
```

Example

Let's try and reason about this by keeping track of the lower and upper bounds of the variables!

```python
def f(i1, i2):
```

$\{i_1 \text{->} [0, 0.3],\ i_2 \text{->} [0.1, 0.4]\}$ (Given)

```python
    h1 = i1 + i2
```

$\{h_1 \text{->} [0.1, 0.7]\}$ (To add intervals, add the lower bounds and upper bounds)

```python
    h2 = i1 - i2
```

$\{h_2 \text{->} [-0.4, 0.2]\}$ ($-i_2 \text{->} [-0.4, -0.1]$ and $i_1 - i_2 = i_1 + (-i_2)$)

```python
    h1 = max(0, h1)
```

$\{h_1 \text{->} [0.1, 0.7]\}$ (Max has no effect on h_1 as it is already more than 0)

```python
    h2 = max(0, h2)
```

$\{h_2 \text{->} [0, 0.2]\}$ (Max prunes the negative part from h_2)

```python
    o1 = h1 + h2 + 0.5
```

$\{o_1 \text{->} [0.6, 1.4]\}$ ($[0.1 + 0 + 0.5,\ 0.7 + 0.2 + 0.5]$)

```python
    o2 = 2*h1 - h2 - 1
```

$\{o_2 \text{->} [-1, 0.4]\}$ ($-h2 \text{->} [-0.2, 0]$ and then $o2 \text{->} [2*0.1 - 0.2 - 1,\ 2*0.7 + 0 - 1]$)

```python
    return o1, o2
```

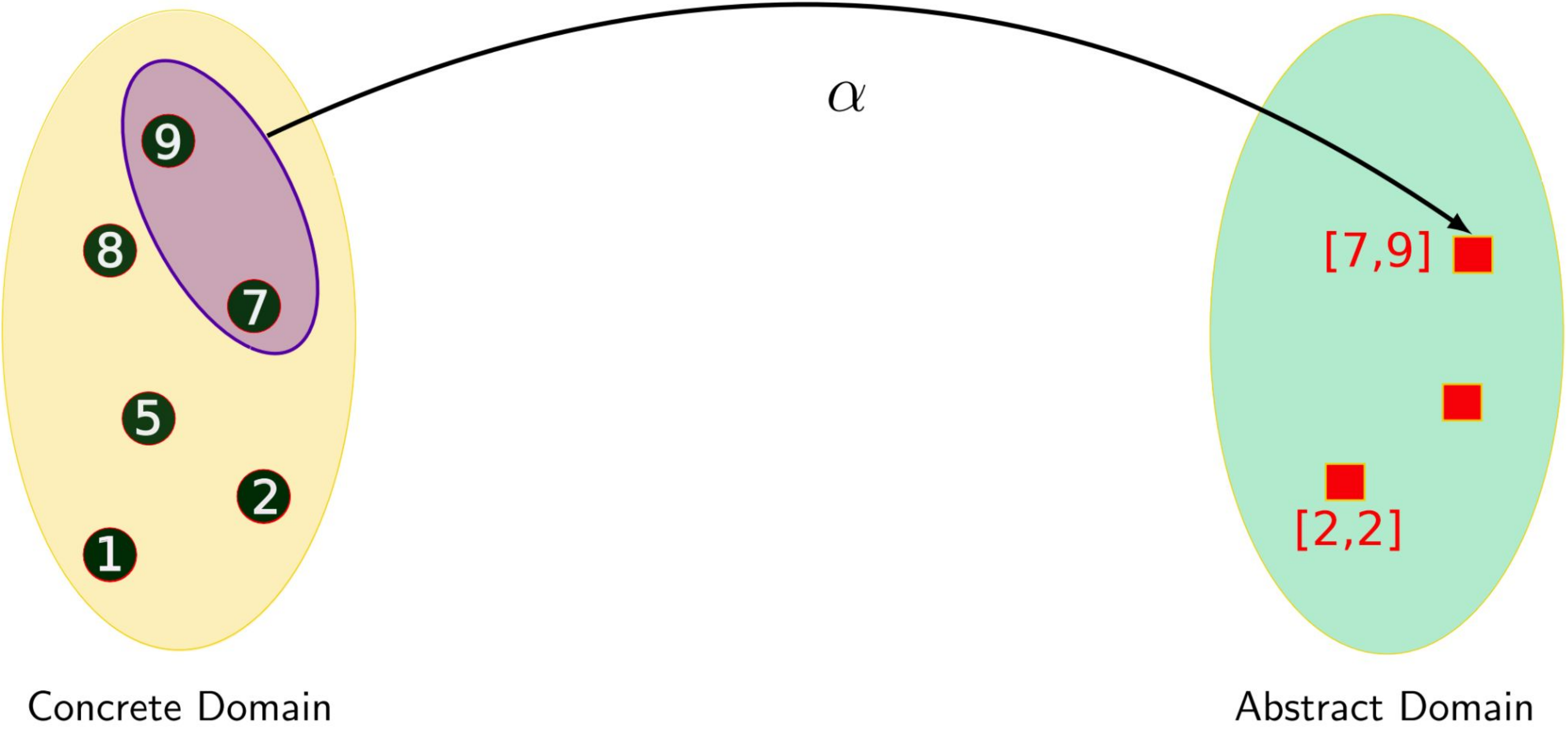
Example analysis

- Had to prove that if i_1 ranges from [0, 0.3] and i_2 ranges from [0.1, 0.4], then $o_2 \geq -1$
- Have proved that: o_1 -> [0.6, 1.4] and o_2 -> [-1, 0.4]. This helps us to prove that:
    - $o_2 \geq -1$
    - Other similar properties : $o_1 \geq 0.6$
    - More complex properties : $o_1 > o_2$
- Maintaining intervals for variables enabled us to reason about all possible program states together.
- This was possible by interpreting the program states in another abstract domain (intervals here).

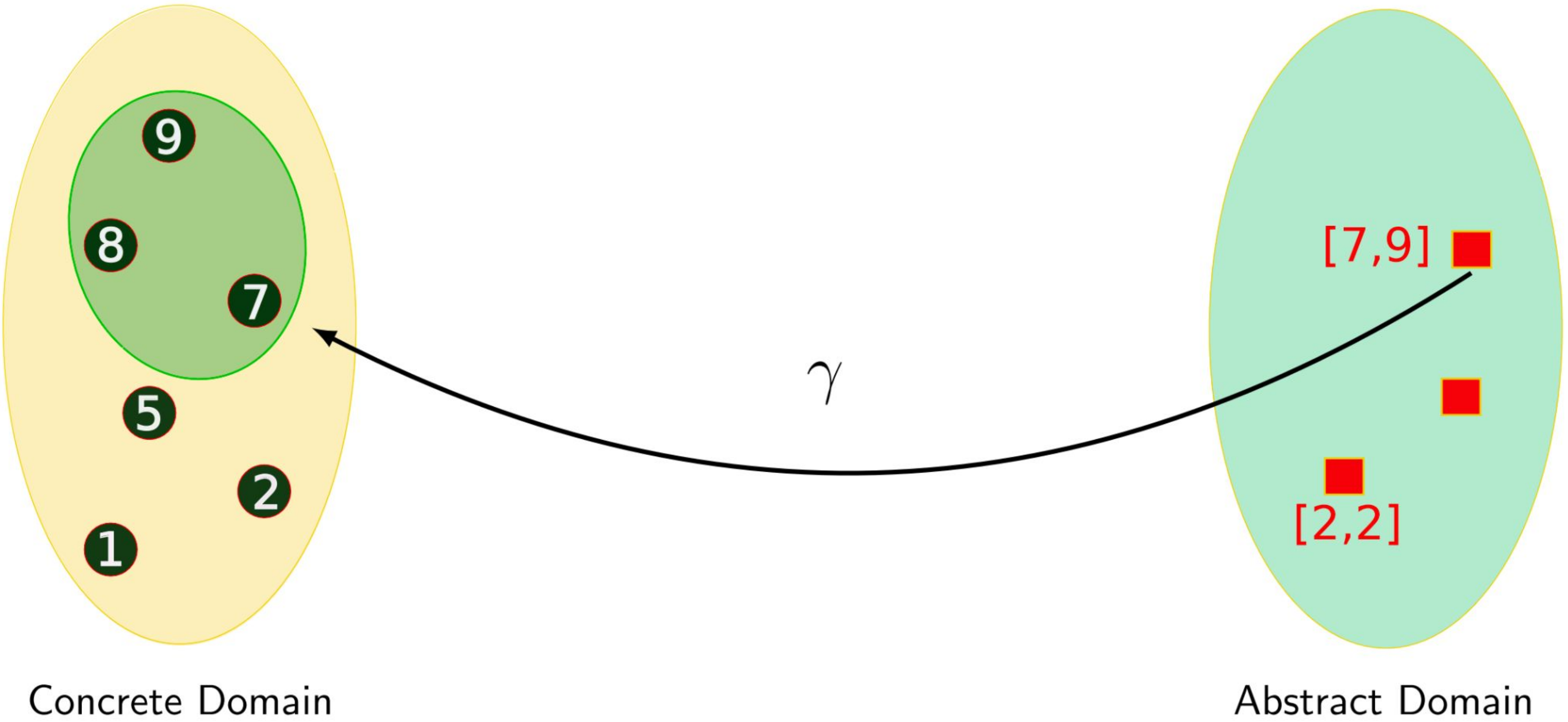
Abstract Domains

- **Abstract Domains (A):** Domain of values that are used to keep track of the program states (the concrete domain C) *succinctly*.
- Some examples: Interval, Zonotopes, Octagon, Polyhedra

Abstraction Function

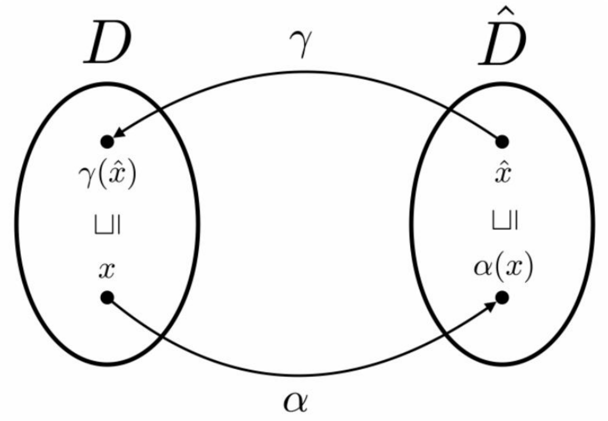


Concretization Function



Galois Connection

$$\forall x \in D, \forall \hat{x} \in \hat{D}.\ \alpha(x) \sqsubseteq \hat{x} \Leftrightarrow x \sqsubseteq \gamma(\hat{x})$$



Intuitively, this says that α, γ respect the orderings of $D, \hat{D}$

Introduction

- Consider the + operation and the code line z = x + y.

- When interpreting the program on interval domain:

  If $x^\#$ = [a, b] and $y^\#$ = [c, d], then we need a operator $+^\#$ that gives us $z^\#$

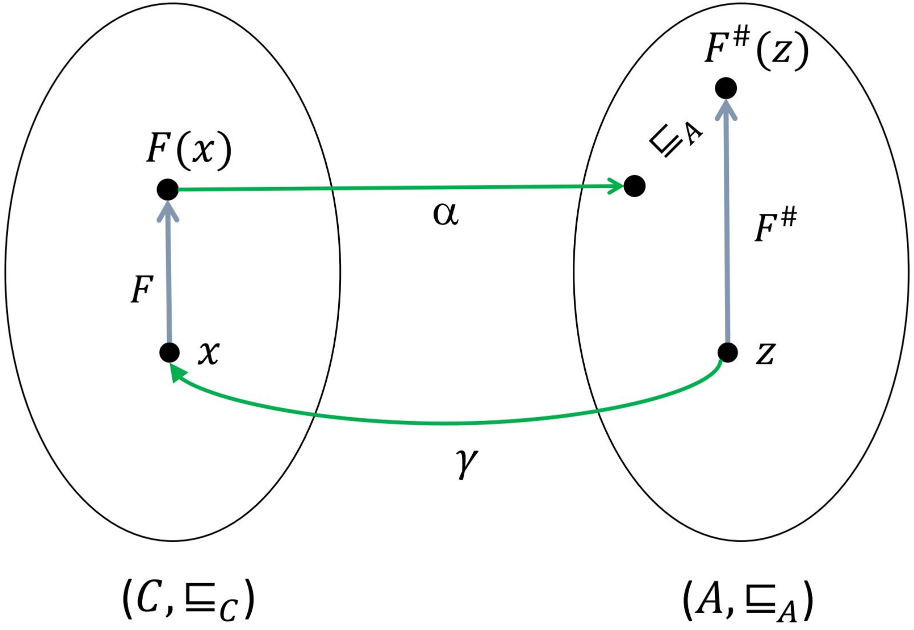
  $z^\# = x^\# +^\# y^\# = [a, b] +^\# [c, d] = [a+b,\ c+d]$

- We call $+^\#$ the *abstract transformer* for +

- We need abstract transformers for all operations in the language.

Soundness

$$\forall z \in A.\, \alpha\left(F\left(\gamma(z)\right)\right) \sqsubseteq_A F^{\#}(z)$$



- **Necessary** condition for transformer correctness.

- We define the best transformer $F_{best}^{\#}(z)$ as:
  - Concretize the z to get the set of concrete values mapped to it
  - Apply F to all those concrete values to get a set C' (= F(x)) of concrete values
  - Get the abstract values for the set C'

- Any transformer $F^{\#}(z)$ is sound if it over-approximates $F_{best}^{\#}(z)$ (includes all abstract values computed by the best transformer)

- If $[a, b] +^{\#} [c,d] = [e, f]$, then $+^{\#}$ is sound if:

$$\forall x \in [a, b], \forall y \in [c, d] \Rightarrow x + y \in [e, f]$$

Precision

- Important for the practical applicability of abstract interpretation.
- Can be thought of as the measure of succinctness of the transformer's output.

Consider $[a, b] +^{\#} [c,d] = [e, f]$, $+^{\#}$ is sound if $\forall\, x \in [a, b], \forall\, y \in [c, d] \Rightarrow x + y \in [e, f]$

Now consider the following possible transformers:

	SOUND?	PRECISE?
$[-\infty, \infty]$		
$[a + c - 5, b + d + 6]$		
$[a + c + 1, b + d]$		
$[a + c, b + d]$		

Precision

- Important for the practical applicability of abstract interpretation.
- Can be thought of as the measure of succinctness of the transformer's output.

Consider $[a, b] +^{\#} [c,d] = [e, f]$, $+^{\#}$ is sound if $\forall\ x \in [a, b], \forall\ y \in [c, d] \Rightarrow x + y \in [e, f]$

Now consider the following possible transformers:

	SOUND?	PRECISE?
$[-\infty, \infty]$	YES	
$[a + c - 5, b + d + 6]$		
$[a + c + 1, b + d]$		
$[a + c, b + d]$		

Precision

- Important for the practical applicability of abstract interpretation.
- Can be thought of as the measure of succinctness of the transformer's output.

Consider $[a, b] +^{\#} [c,d] = [e, f]$, $+^{\#}$ is sound if $\forall\ x \in [a, b], \forall\ y \in [c, d] \Rightarrow x + y \in [e, f]$

Now consider the following possible transformers:

	SOUND?	PRECISE?
$[-\infty, \infty]$	**YES**	**NO**
$[a + c - 5, b + d + 6]$		
$[a + c + 1, b + d]$		
$[a + c, b + d]$		

Precision

- Important for the practical applicability of abstract interpretation.
- Can be thought of as the measure of succinctness of the transformer's output.

Consider $[a, b] +^{\#} [c,d] = [e, f]$, $+^{\#}$ is sound if $\forall\, x \in [a, b],\ \forall\, y \in [c, d] \Rightarrow x + y \in [e, f]$

Now consider the following possible transformers:

	SOUND?	PRECISE?
$[-\infty, \infty]$	**YES**	**NO**
$[a + c - 5, b + d + 6]$	**YES**	
$[a + c + 1, b + d]$		
$[a + c, b + d]$		

Precision

- Important for the practical applicability of abstract interpretation.
- Can be thought of as the measure of succinctness of the transformer's output.

Consider $[a, b] +^{\#} [c,d] = [e, f]$, $+^{\#}$ is sound if $\forall\ x \in [a, b], \forall\ y \in [c, d] \Rightarrow x + y \in [e, f]$

Now consider the following possible transformers:

	SOUND?	PRECISE?
$[-\infty, \infty]$	YES	NO
$[a + c - 5, b + d + 6]$	YES	NO
$[a + c + 1, b + d]$		
$[a + c, b + d]$		

Precision

- Important for the practical applicability of abstract interpretation.
- Can be thought of as the measure of succinctness of the transformer's output.

Consider $[a, b] +^{\#} [c,d] = [e, f]$, $+^{\#}$ is sound if $\forall\ x \in [a, b], \forall\ y \in [c, d] \Rightarrow x + y \in [e, f]$

Now consider the following possible transformers:

	SOUND?	PRECISE?
$[-\infty, \infty]$	**YES**	**NO**
$[a + c - 5, b + d + 6]$	**YES**	**NO**
$[a + c + 1, b + d]$	**NO**	<don't care>
$[a + c, b + d]$		

Precision

- Important for the practical applicability of abstract interpretation.
- Can be thought of as the measure of succinctness of the transformer's output.

Consider $[a, b] +^{\#} [c,d] = [e, f]$, $+^{\#}$ is sound if $\forall\ x \in [a, b], \forall\ y \in [c, d] \Rightarrow x + y \in [e, f]$

Now consider the following possible transformers:

	SOUND?	PRECISE?
$[-\infty, \infty]$	YES	NO
$[a + c - 5, b + d + 6]$	YES	NO
$[a + c + 1, b + d]$	NO	<don't care>
$[a + c, b + d]$	YES	YES

Motivation

- Abstract transformers are often non-trivial even for a simple operation.

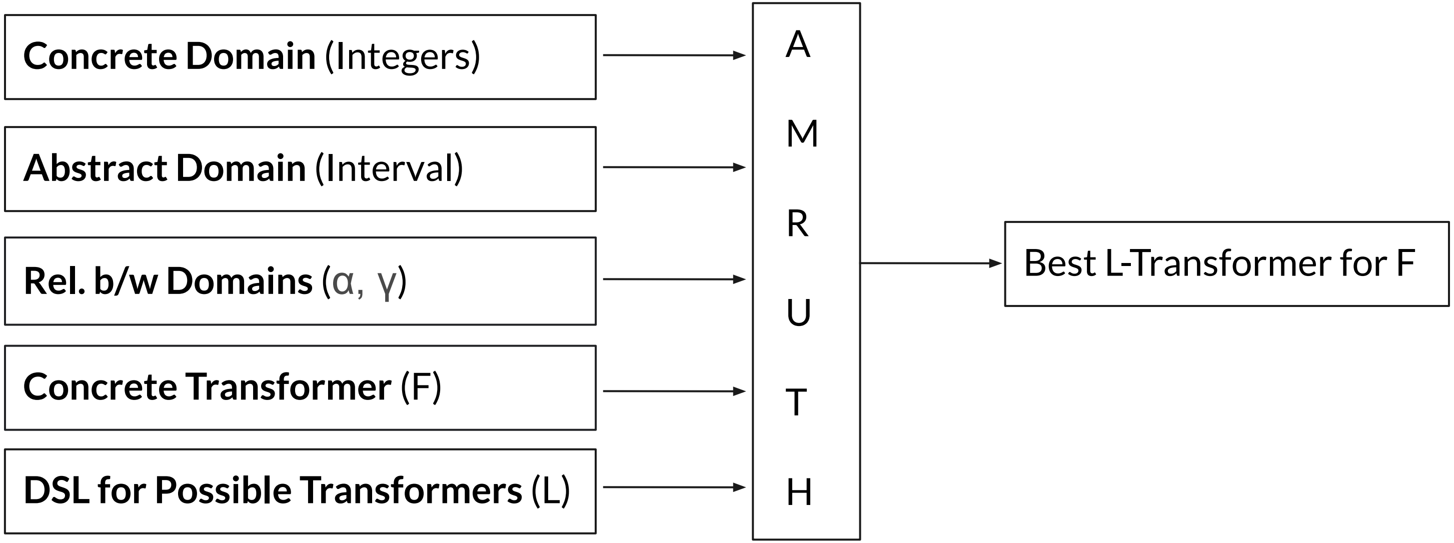
  E.g.: The most precise transformer for abs(x) in the interval domain is:

$$\mathrm{abs}^{\sharp}(a) = [\mathtt{max}(\mathtt{max}(0, a.l), -a.r),\ \mathtt{max}(-a.l, a.r)].$$

- Manually written abstract transformers error-prone (unsound) and can be imprecise.
- AMURTH found multiple bugs in abstract transformers in the existing abstract interpretation engines.

AMURTH can synthesize non-trivial transformers in reasonable time (< 2000 seconds).

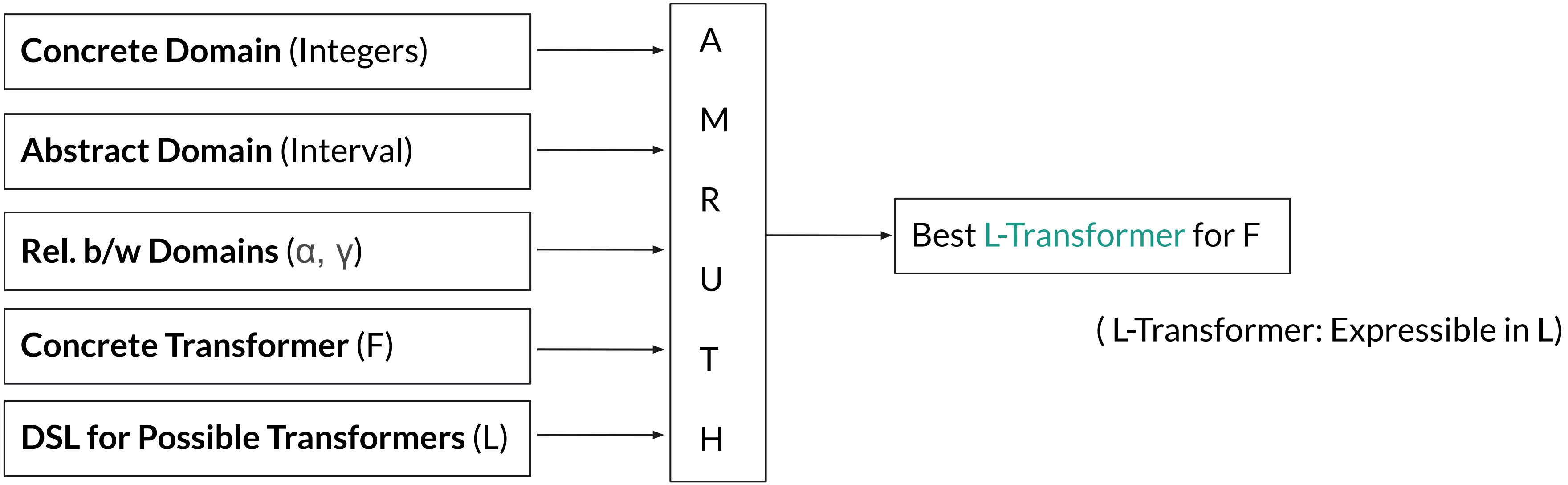
High Level Diagram



$$
\begin{aligned}
\mathit{Transformer} &::= \lambda \mathsf{a}.[E, E] \\
E &::= \mathtt{a.l} \mid \mathtt{a.r} \mid 0 \mid -E \mid +\infty \mid -\infty \mid E + E \mid E - E \mid E * E \mid \mathtt{min}(E, E) \mid \mathtt{max}(E, E)
\end{aligned}
$$

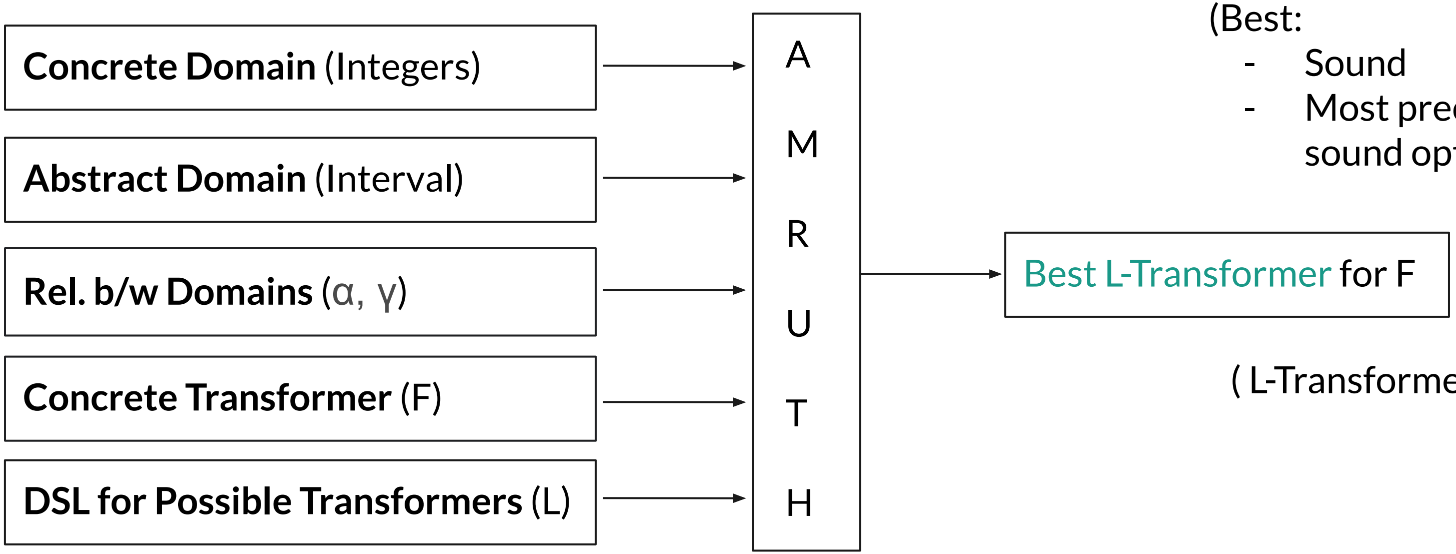
AMURTH

High Level Diagram



$$\begin{aligned} \mathit{Transformer} &::= \lambda \mathrm{a}.[E, E] \\ E &::= \texttt{a.l} \mid \texttt{a.r} \mid 0 \mid -E \mid +\infty \mid -\infty \mid E+E \mid E-E \mid E*E \mid \texttt{min}(E,E) \mid \texttt{max}(E,E) \end{aligned}$$

High Level Diagram



(Best:
- Sound
- Most precise among other sound options expressible in L)

(L-Transformer: Expressible in L)

$$\begin{aligned} \mathit{Transformer} &::= \lambda \mathtt{a}.[E, E] \\ E &::= \mathtt{a.l} \mid \mathtt{a.r} \mid 0 \mid -E \mid +\infty \mid -\infty \mid E + E \mid E - E \mid E * E \mid \mathtt{min}(E, E) \mid \mathtt{max}(E, E) \end{aligned}$$

Soundness (or +ve) Counterexamples

- AMURTH works by guessing potential transformers from the DSL.
- These guesses are then corrected/guided by counterexamples.

For the abs(x) case, say AMURTH guesses the transformer: $abs^{\#}([l, r]) = [0, l+r]$

Consider **[-2, 2]:**

- $abs^{\#}[-2,2]$ should capture all values between [0,2]
- But $abs^{\#}[-2,2]$ computes to [0, 0] (is missing the concrete value 2)
- <[-2, 2], 2> is a soundness counterexample

General form: $\langle a, c' \rangle$ such that $a \in A$ and $c' \in \gamma(F_{best}^{\#}(a))$ but $c' \notin \gamma(F^{\#}(a))$

Soundness (or +ve) Counterexamples

- What if we only use Soundness counterexamples to refine our guesses?

  Are there some drawbacks?

Soundness (or +ve) Counterexamples

- What if we only use Soundness counterexamples to refine our guesses?

  Are there some drawbacks?

Ans: YES! If we only use Soundness counterexamples, nothing is stopping the tool to synthesize $[-\infty, \infty]$ everytime.

Reason: There are no preciseness constraints!

Precision (-ve) Counterexamples

For the abs(x) case, say AMURTH guesses the transformer: $abs^{\#}([l, r]) = [0, l+r]$

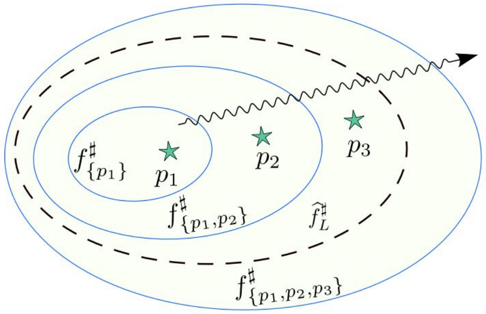
Consider **[2, 4]:**

- $abs^{\#}[2, 4]$ should capture all values between [2, 4]
- But $abs^{\#}[2, 4]$ computes to [0, 6] (has many redundant values, lets pick 5)
- <[2, 4], 5> is a precision counterexample

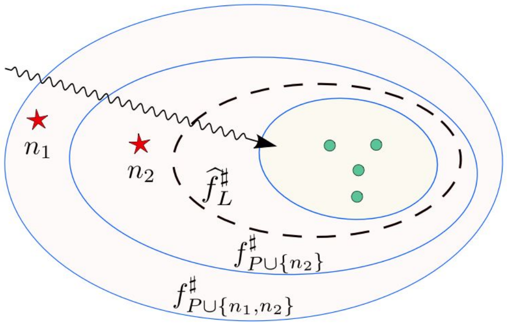
General form: $\langle a, c' \rangle$ such that $a \in A$ and $\exists\ F_{L\text{-}best}^{\#}(a)$ such that $c' \notin \gamma(F_{L\text{-}best}^{\#}(a))$

Algorithm Overview

- Amurth uses counterexample-guided inductive synthesis (CEGIS) strategy
- Attempts to meet the *dual objectives* of soundness and precision



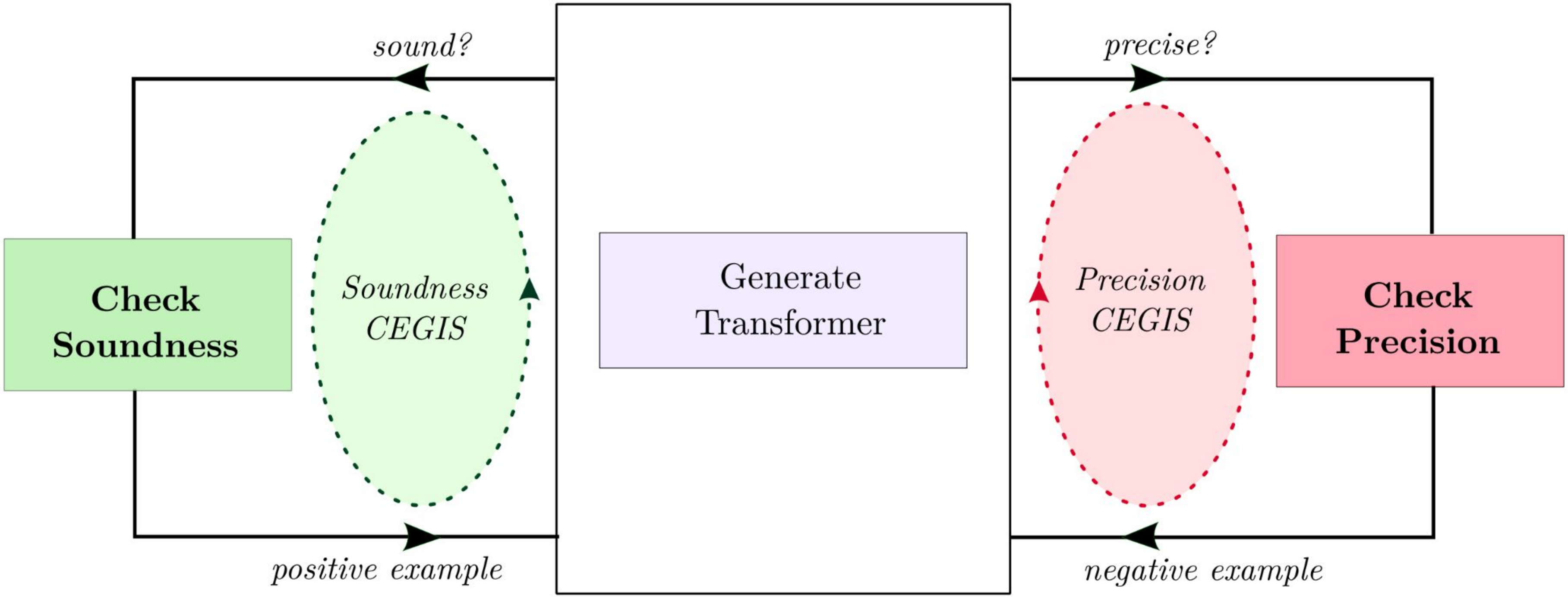
(a) Adding positive counterexamples



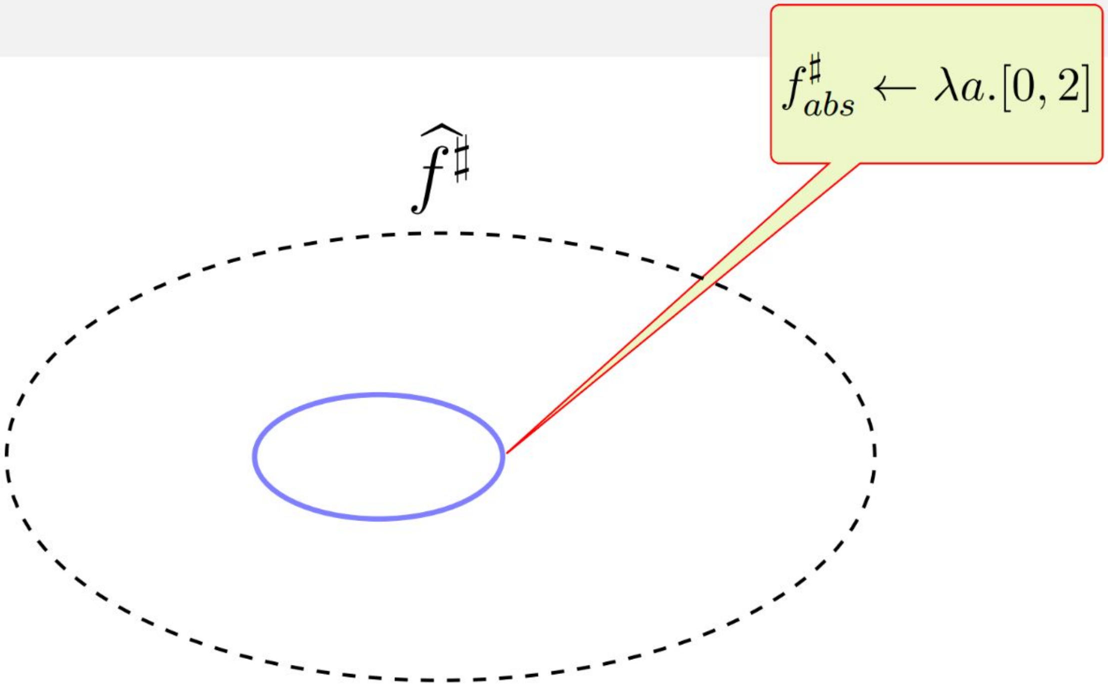
(b) Adding negative counterexamples. P is a set of positive examples (•) .

- Counterexamples generated by soundness and precision verifiers drive two CEGIS loops.

Algorithm Overview



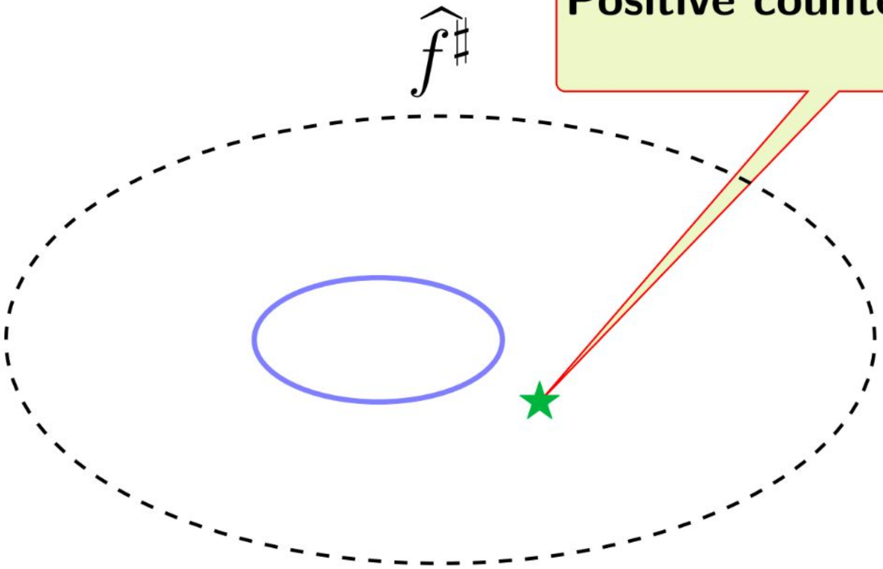
AMURTH in action!



AMURTH in action!

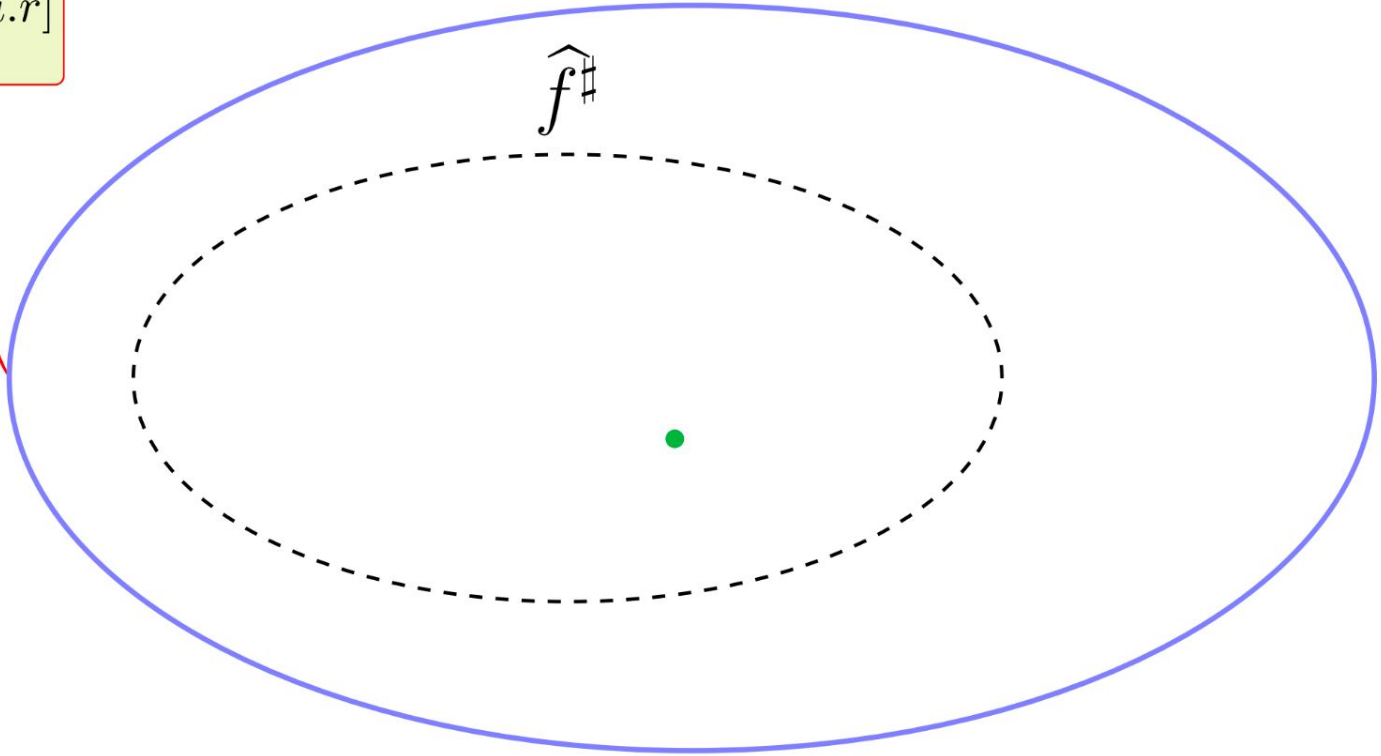
$$f^{\sharp}_{abs} \leftarrow \lambda a.[0, 2]$$

Positive counterexample: $\langle [0, 5], 3 \rangle$



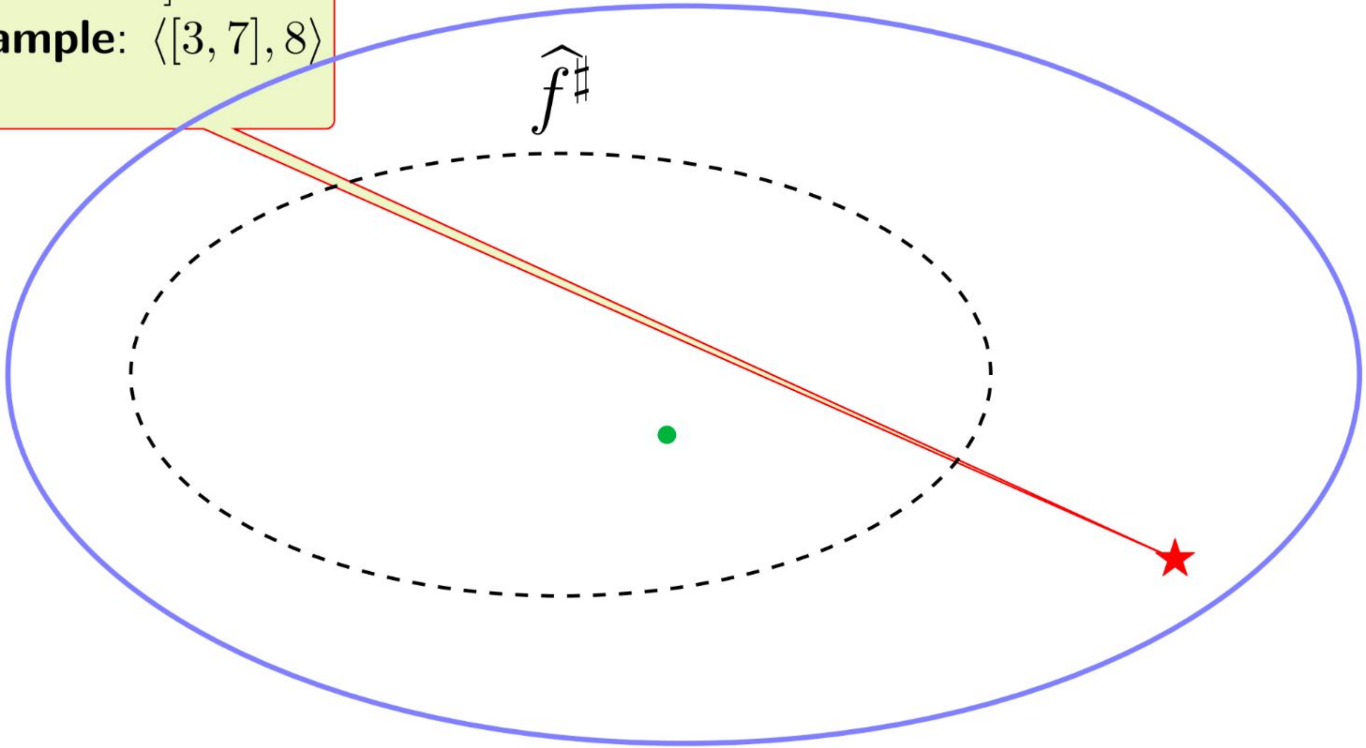
AMURTH in action!

$$f^\sharp_{abs} \leftarrow \lambda a.[0, a.l + a.r]$$

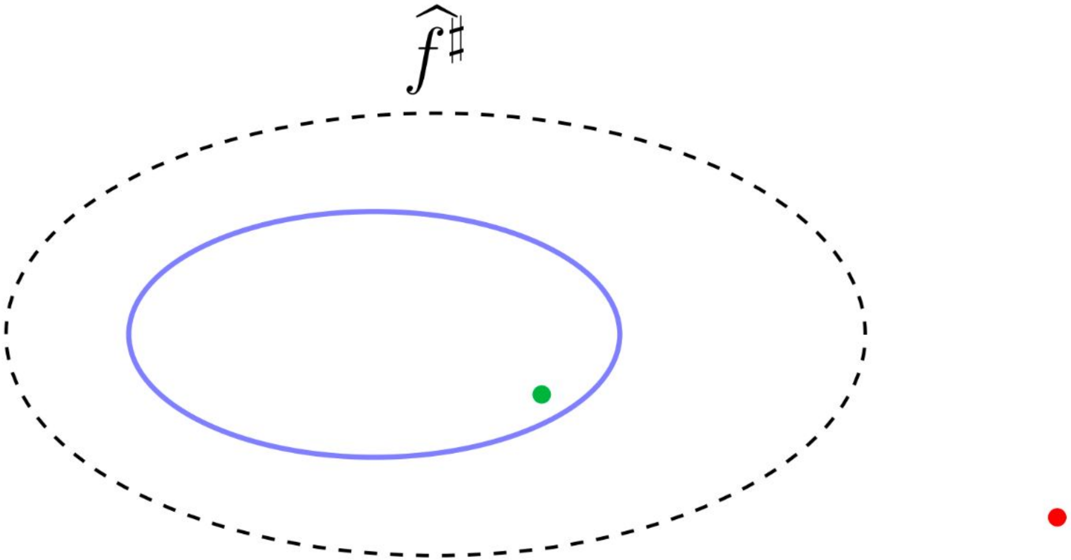


$$f^{\sharp}_{abs} \leftarrow \lambda a.[0, a.l + a.r]$$

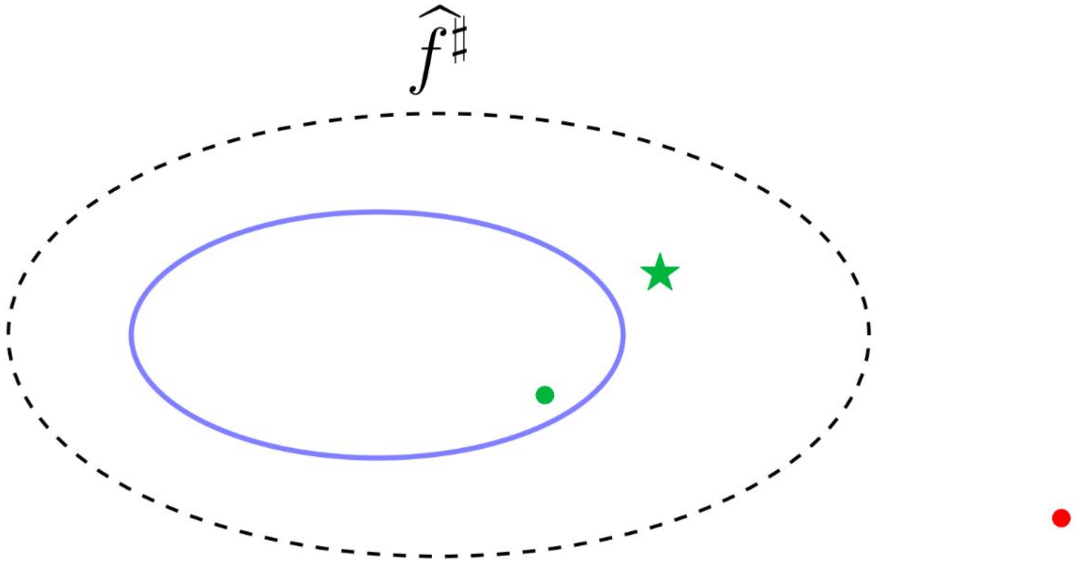
Negative counterexample: $\langle [3,7], 8 \rangle$



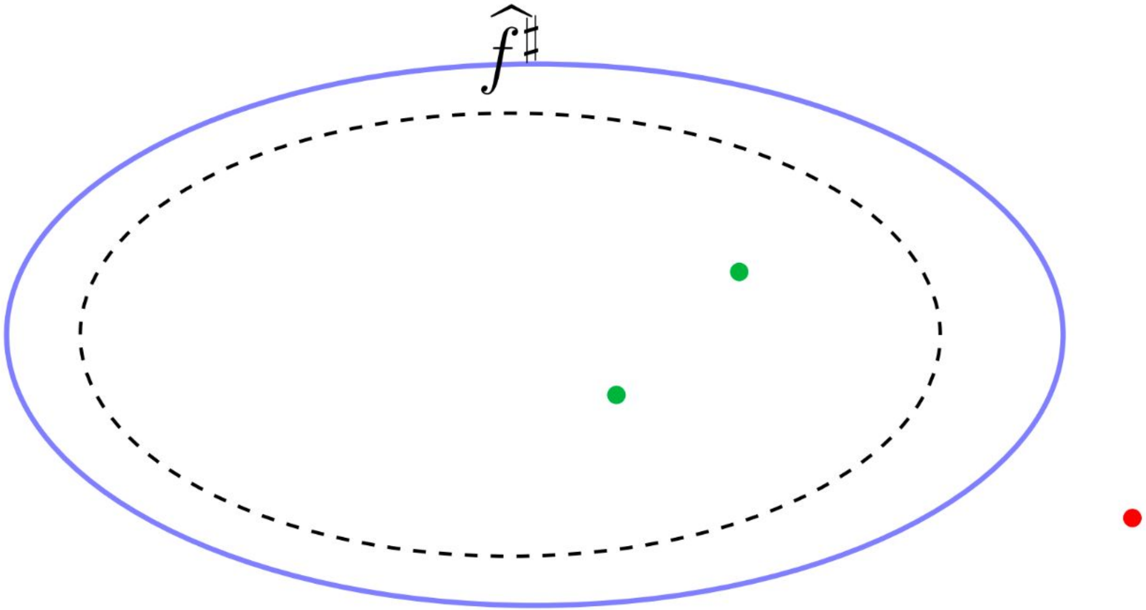
AMURTH in action!



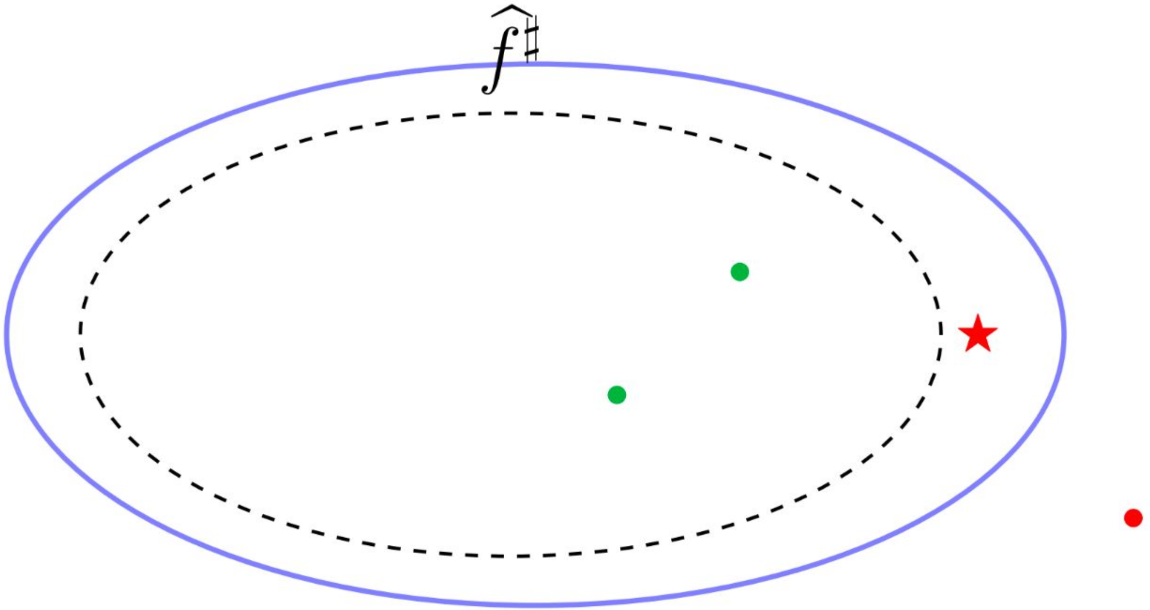
AMURTH in action!



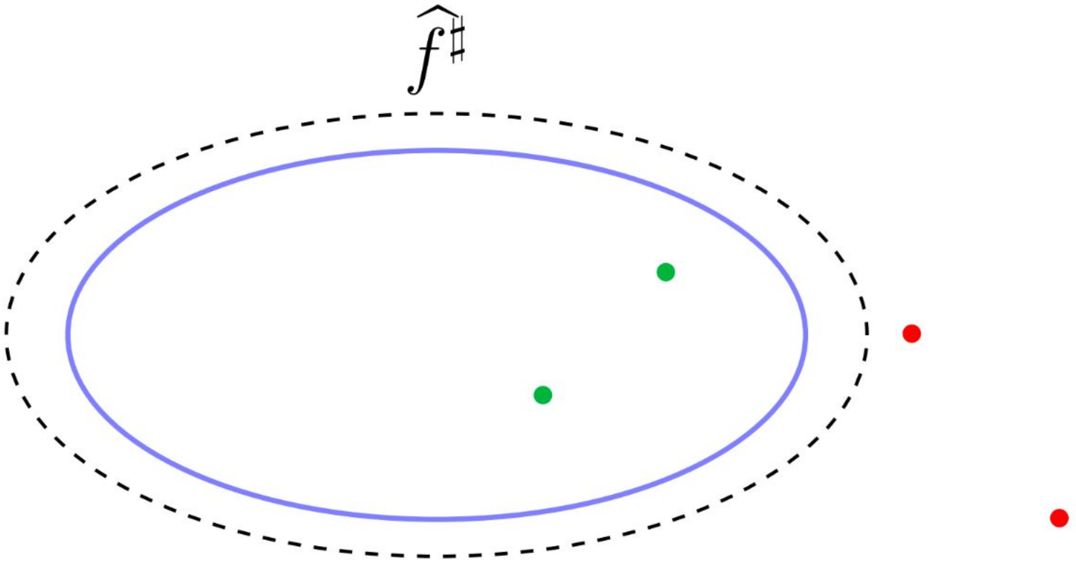
Amurth in action!



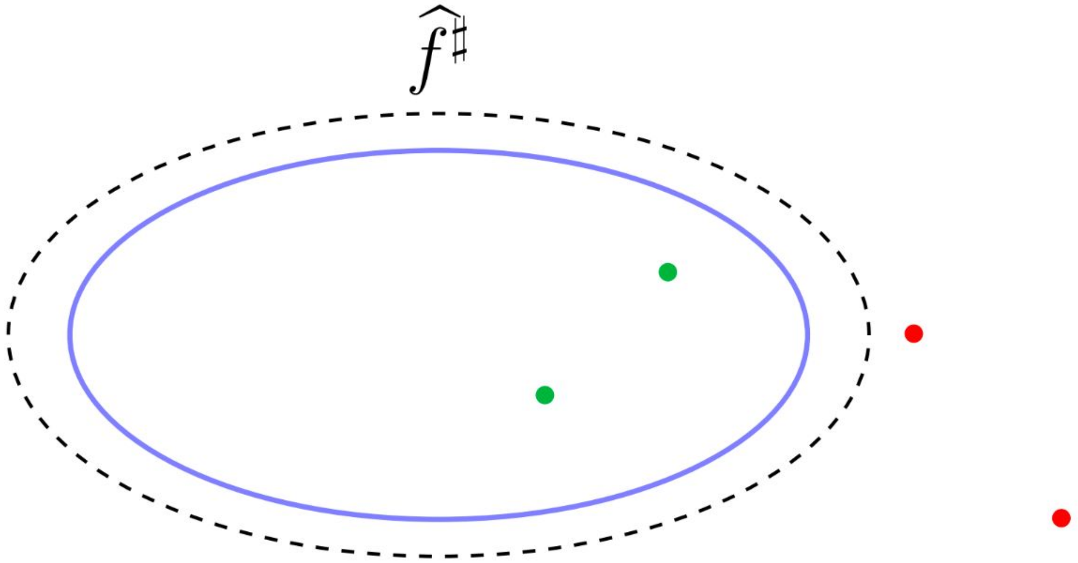
AMURTH in action!



AMURTH in action!



AMURTH in action!



This stops when there are no more soundness and precision counterexamples.

Theorems for Correctness

Theorem 1

If Algorithm terminates, it returns a best L-transformer for the concrete function f.

Theorem 2

If the DSL L is finite, Algorithm always terminates.

Evaluation

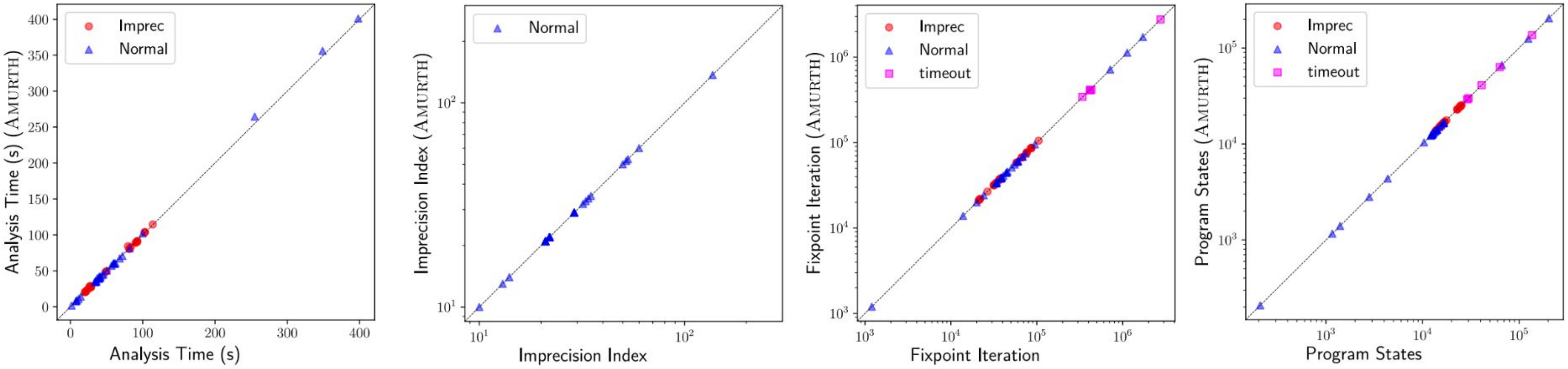
Domain Type	Abstract Domains	Operations
String	Constant String ($\mathcal{CS}$) String Set (size k) ($\mathcal{SS}_k$) Char Inclusion ($\mathcal{CI}$) Prefix-Suffix ($\mathcal{PS}$) String Hash ($\mathcal{SH}$)	$\texttt{charAt}^\sharp$ $\texttt{concat}^\sharp$, $\texttt{contains}^\sharp$, $\texttt{toLower}^\sharp$, $\texttt{toUpper}^\sharp$, $\texttt{trim}^\sharp$
Fixed Bitwidth Interval	Unsigned-Int ($\mathcal{A}_{\texttt{uintv}}$) Signed-Int ($\mathcal{A}_{\texttt{uintv}}$) Wrapped ($\mathcal{W}$)	$\texttt{add}^\sharp$, $\texttt{sub}^\sharp$, $\texttt{mul}^\sharp$, $\texttt{and}^\sharp$, $\texttt{or}^\sharp$, $\texttt{xor}^\sharp$, $\texttt{shl}^\sharp$, $\texttt{ashr}^\sharp$, $\texttt{lshr}^\sharp$

Results

f	$\mathcal{CS}$	$\mathcal{SS}_k$	$\mathcal{CI}$	$\mathcal{PS}$	$\mathcal{SH}$
charAt	18.29	3.94	24.91	5.94	3.76
concat	99.05	9.57	1,983.83	8.92	609.30
contains	132.06	78.42	1,804.69	9.13	10.39
toLower	11.26	11.74	381.65	6.91	8.44
toUpper	9.77	12.18	735.13	5.85	3.73
trim	4.31	16.35	641.53	8.52	8.29

Time taken to synthesize the transformers (in secs)

Results



Similar performance as manually written transformers in terms of analysis time, imprecision index, fixpoint iteration, program states.

Results & Conclusion

- The transformers generated by AMURTH were as effective as the manually written ones.
- When transformers generated by AMURTH were compared to the existing ones, the authors found *4 soundness bugs* in the present transformers.
- This shows the current manual techniques can be error-prone, imprecise and sound.
- Using a tool like AMURTH can let you generate abstract transformers which are provably sound and precise.

Existing Soundness Bugs

```
contains♯_CI(a₁ : CI)(a₂ : CI) : AbsBool =
 ite(isBot(a₁.l, a₁.u)∨isBot(a₂.l, a₂.u),
   boolBot,
[−] ite(isTop(a₁.l, a₁.u) ∨ isTop(a₂.l, a₂.u), // Bug
[−]  boolTop,                                  // Bug
     ite(¬isSubset(a₂.l, a₁.u),
       boolFalse,
   [-] ite(size(a₂.u) ≤ 1 ∧ isSubset(a₂.u, a₁.l), // Bug
   [+] ite(isEmpty(a₂),                           // Fix
           boolTrue,
      [−]  boolTop))))                            // Bug
      [+]  boolTop)))                             // Fix
```

(a) Abstract transformers for contains.

```
trim♯_CI(a : CI) : CI =
 ite(isBot(a.l,a.u),
   Bot,
   ite(isTop(a.l,a.u),
     Top,
     ite(size(a.u)≤1∧containsSpace(a.u),
       [∅,∅],
   [-] a                          // Bug
   [+] [removeSpace(a.l), a.u]    // Fix
       )))
```

(b) Abstract transformers for trim.

Fig. 6. Bugs found and fixed in the CI domain for contains and trim. The lines in blue show how the synthesized transformers differ from the incorrect ones in $\mathrm{SAFE}_{\mathrm{str}}$ (denoted by the lines in red).

Existing Soundness Bugs

```
trim#_PS(a : PS) : PS =
 ite(isBot(a.p,a.s),
   BOT,
   ite(isTop(a.p,a.s),
    TOP,
[-] [trimStart(a.p), trimEnd(a.s)] // Bug
[+] [trim(a.p), trim(a.s)]         // Fix
))
```

Fig. 7. Abstract transformers for trim in the $\mathcal{PS}$ domain.

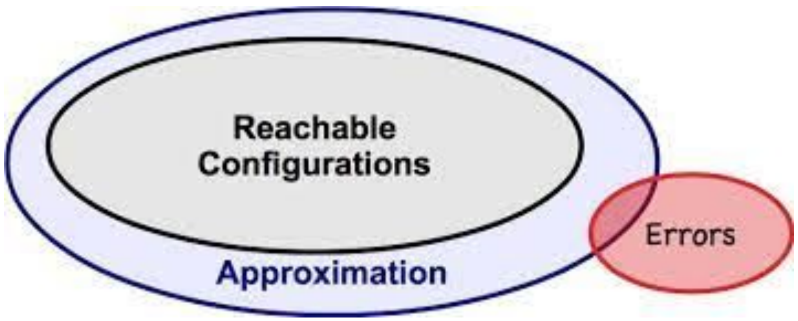
```
concat#(a : Long)(b : Long) : Long =
  r ← reverse(b); c ← 0; i ← 0
  WHILE i < b
    r ←rotateLeft(r, 1)
    IF (a & r) ≠ 0 THEN
[-]   c ← c | (1 << i)   //SAFEstr
[+]   c ← c ^ (1 << i)   //AMURTH
    i ← i + 1
  RETURN c
```

Fig. 8. Abstract transformers for concat in the $\mathcal{SH}$ domain.

Thanks

Backup Slides

Over-approximation Caveat

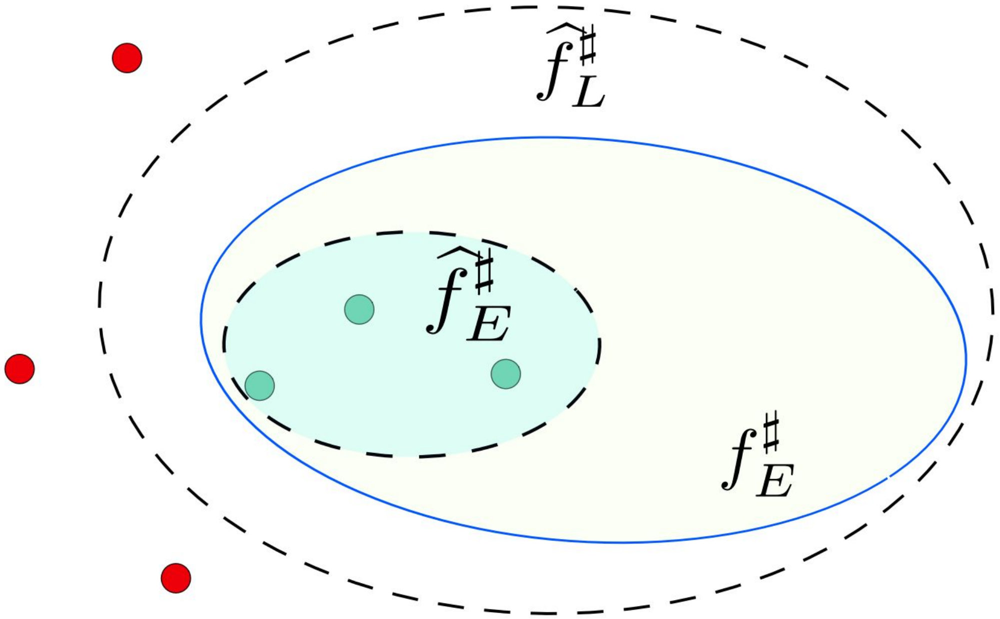


Here, though the approximation we generated has some intersection with error state, we cannot (should not) conclude that we have errors as we over-approximate

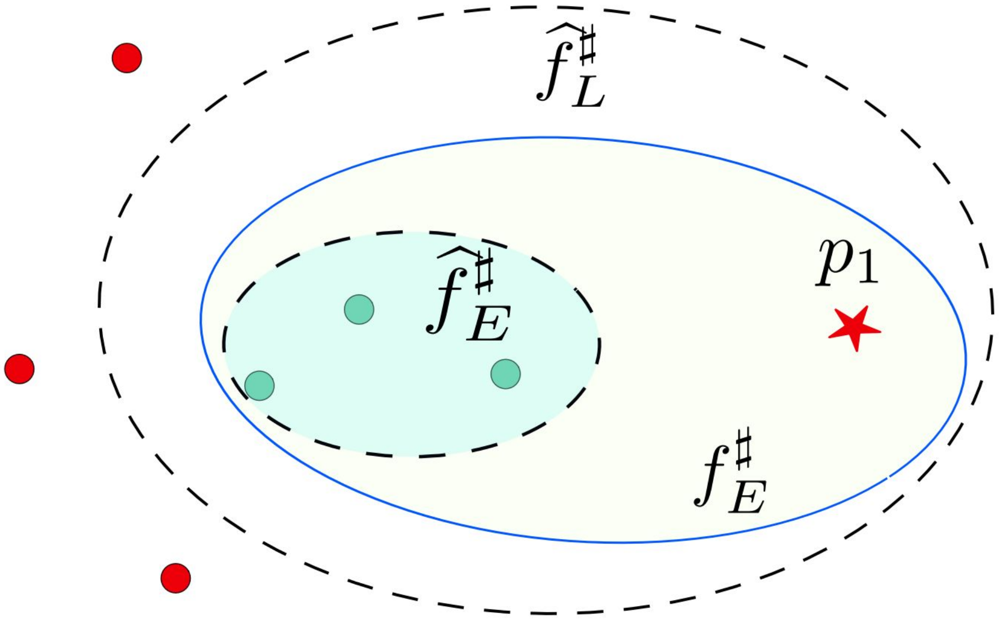
Approximating Precision

- The current set of examples E is used to approximate $f_{\mathrm{L}}^{\sharp}$
- A most-precise L-transformer that satisfies E (denoted by $f_{E}^{\sharp}$) is optimistically assumed to be $f_{\mathrm{L}}^{\sharp}$

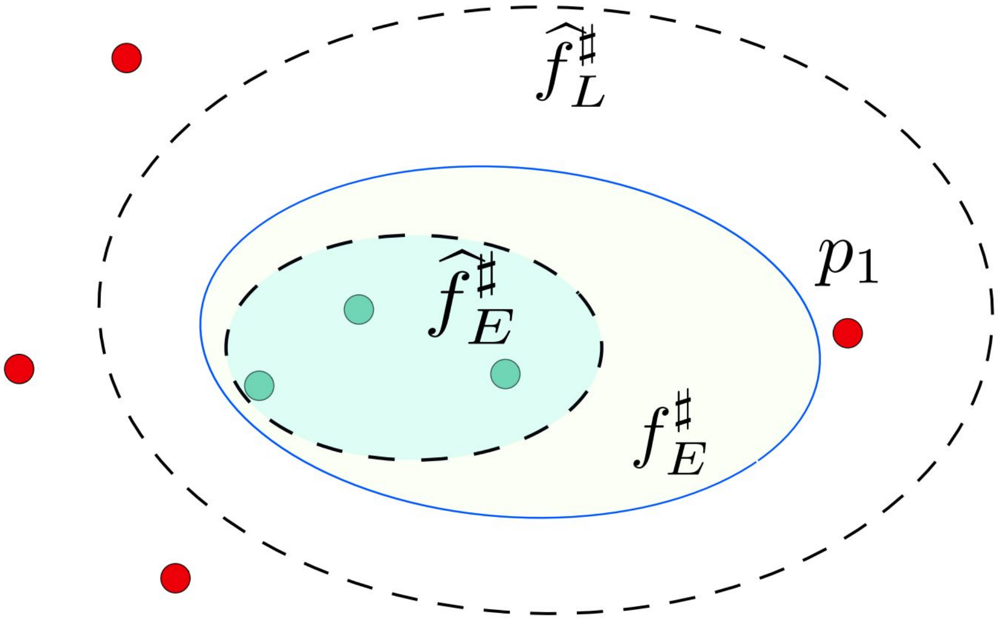
Failed Consistency



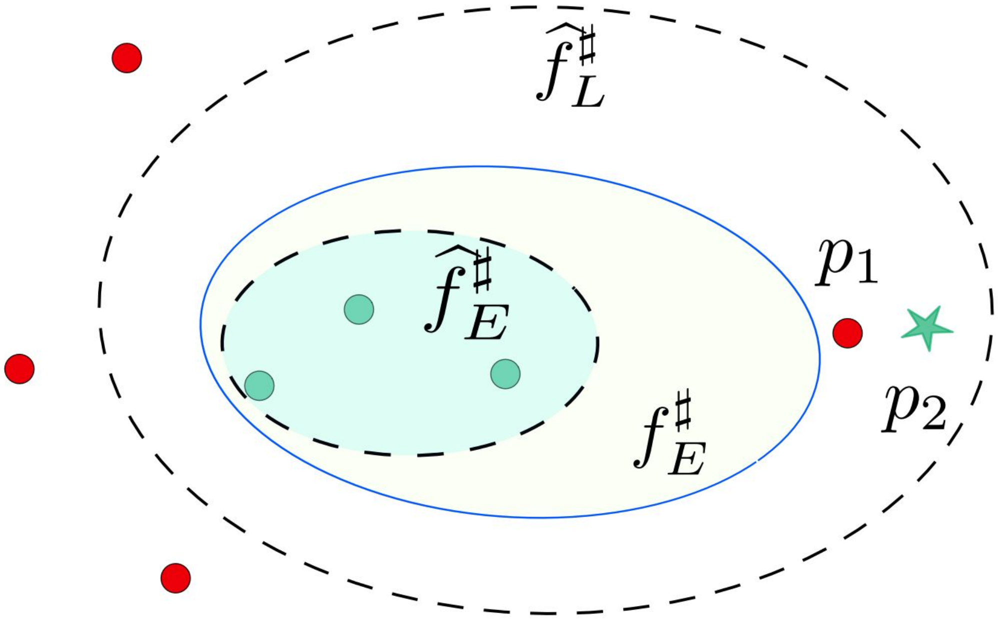
Failed Consistency



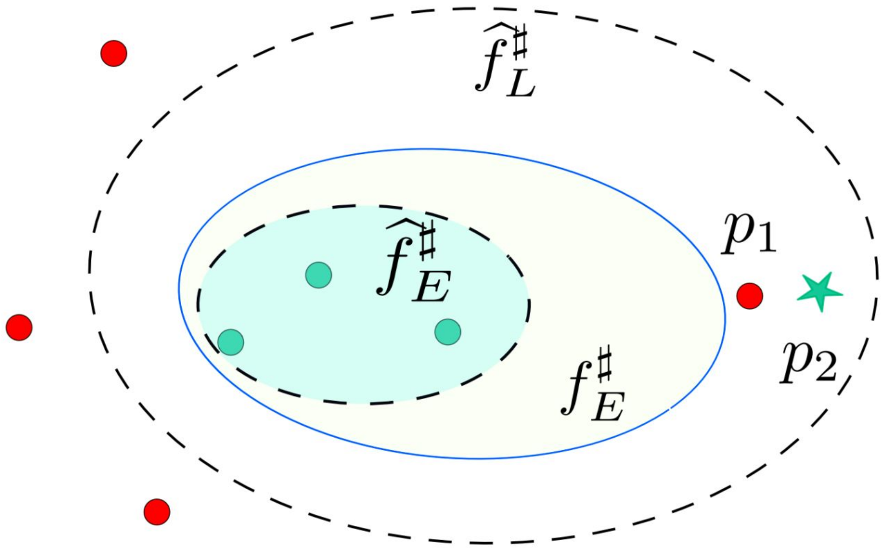
Failed Consistency



Failed Consistency

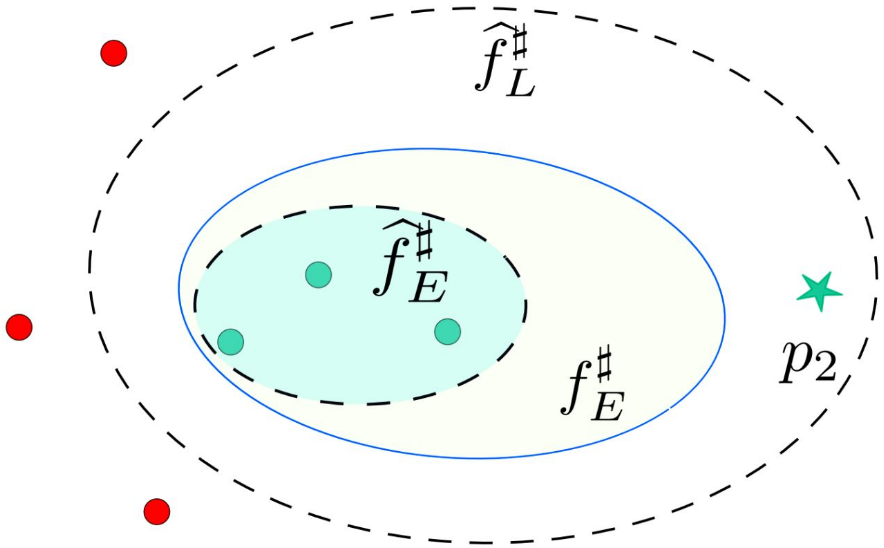


Failed Consistency



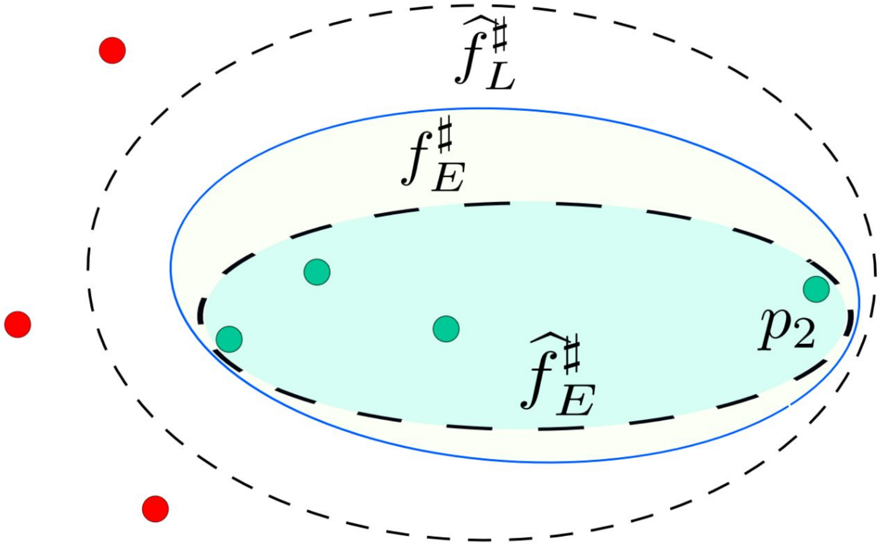
Inconsistent: no $f^{\sharp}_E \in L$ that satisfies all positive and negative examples.

Failed Consistency



$\boxed{\text{Occam's razor}}$

Failed Consistency



Occam's razor

Complete Algorithm