Synthesizing Abstract Transformers for Reduced-Product Domains

Pankaj Kumar Kalita[1], Thomas Reps[2], Subhajit Roy[1]

[1]Indian Institute of Technology Kanpur
[2]University of Wisconsin-Madison

Abstract Interpretation

```
int c = input();
assume(c ⩾ 0);
while(c < 10) {
  c = c + 1;
}
assert(c ⩾ 10);
```

Abstract Interpretation

```
int c = input();
c = [−∞,∞] ⇐
assume(c ⩾ 0);

while(c < 10) {

  c = c + 1;

}

assert(c ⩾ 10);
```

Abstract Interpretation

```
int c = input();
c = [−∞,∞]
assume(c ⩾ 0);
c = [0,∞] ⇐
while(c < 10) {

  c = c + 1;

}

assert(c ⩾ 10);

```

Abstract Interpretation

```
int c = input();
c = [−∞,∞]
assume(c ⩾ 0);
c = [0,∞]
while(c < 10) {
  c = [0, 9] ⇐
  c = c + 1;

}
c = [10, ∞]
assert(c ⩾ 10);

```

Abstract Interpretation

```
int c = input();
c = [−∞,∞]
assume(c ⩾ 0);
c = [0,∞]
while(c < 10) {
   c = [0, 9]
   c = c + 1;

}
c = [10, ∞]
assert(c ⩾ 10);
```

Abstract Transformer

$$[l_1, r_1] +^\sharp [l_2, r_2] = [(l_1 + l_2), (r_1 + r_2)]$$

$$[0,9] +^\sharp [1,1] = [(0+1), (9+1)]$$
$$= [1,10]$$

These abstract transformers need to be created for every concrete operations

Abstract Interpretation

```
int c = input();
c = [−∞,∞]
assume(c ⩾ 0);
c = [0,∞]
while(c < 10) {
  c = [0, 9]
  c = c + 1;
  c = [1, 10] ⇐
}
c = [10, ∞]
assert(c ⩾ 10);

```

Abstract Interpretation

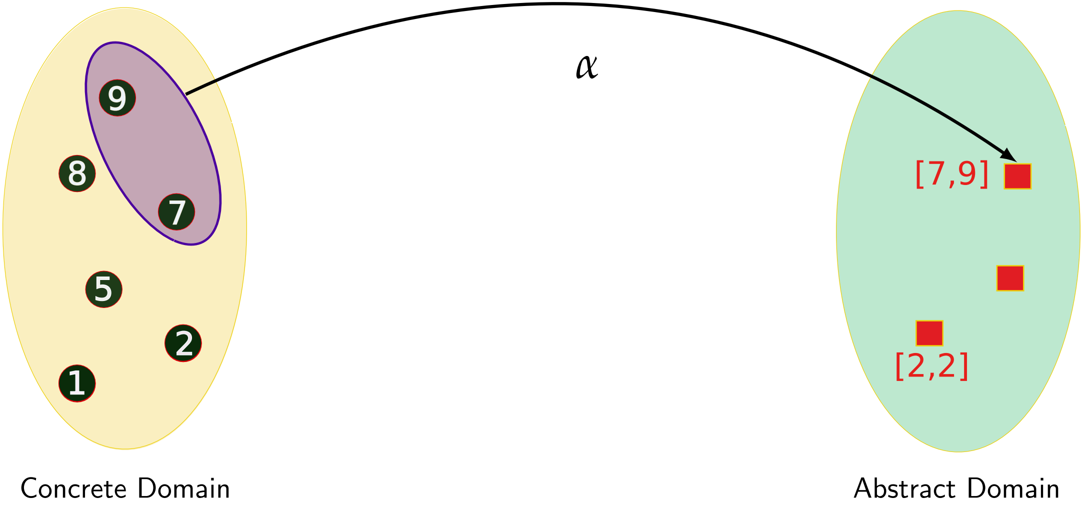
```
int c = input();
c = [−∞,∞]
assume(c ⩾ 0);
c = [0,∞]
while(c < 10) {
  c = [0, 9] ⇐
  c = c + 1;
  c = [1, 10]
}
c = [10, ∞]
assert(c ⩾ 10);
```

Abstract Interpretation

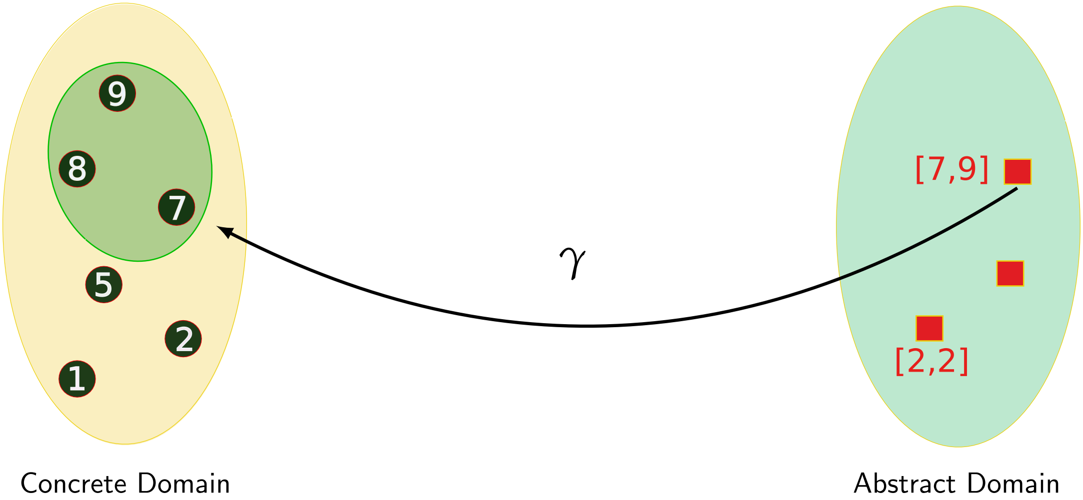
```
int c = input();
c = [-∞,∞]
assume(c ⩾ 0);
c = [0,∞]
while(c < 10) {
  c = [0, 9]
  c = c + 1;
  c = [1, 10]
}
c = [10, ∞] ⇐
assert(c ⩾ 10);

```

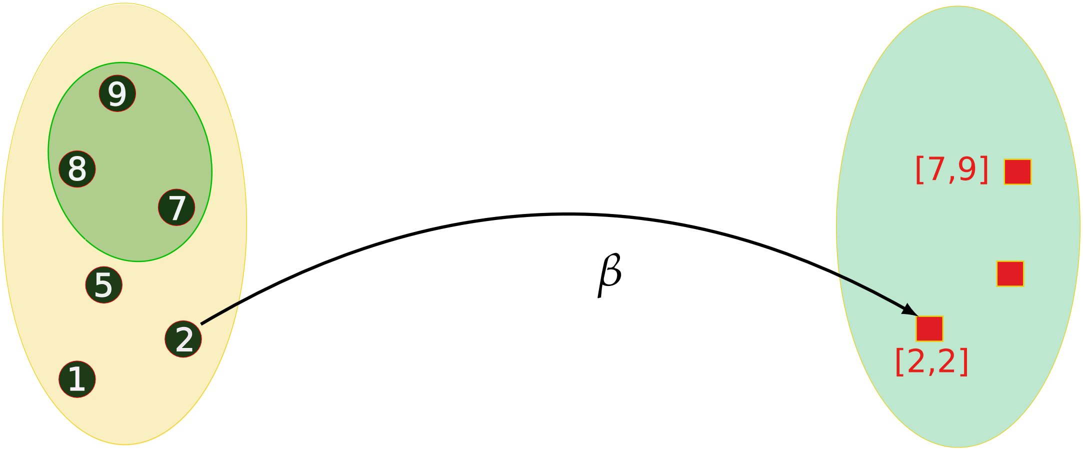
Abstract Interpretation



Abstract Interpretation



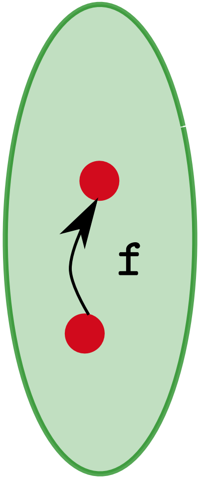
Abstract Interpretation



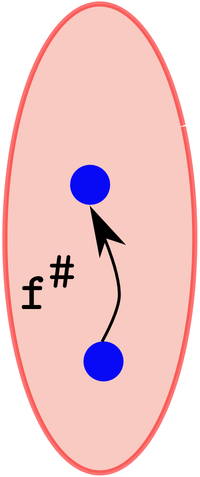
Concrete Domain

Abstract Domain

Abstract Transformer

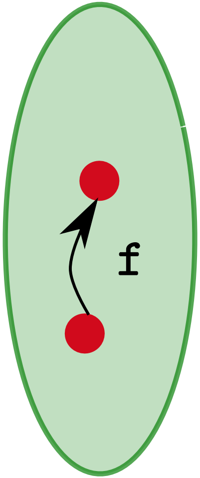


Concrete Domain



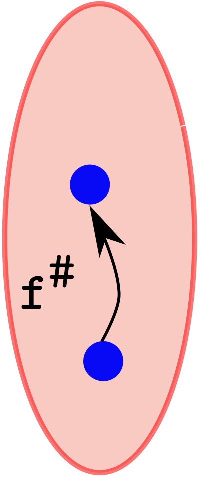
Abstract Domain

Abstract Transformer



Concrete Domain

- Tricky even for trivial operation
- Error-prone



Abstract Domain

Problem statement

Can we automatically *synthesize* an abstract transformer for any operations?

Problem statement

Can we automatically *synthesize* an abstract transformer for any operations?

Given the

- concrete semantics Φ_f of a concrete transformer f,
- description of an abstract domain $(A, \sqsubseteq, \sqcup)$, and its relation to the concrete domain (α, γ, β), and
- a domain-specific language L,

synthesize **a best** abstract transformer $\widehat{f^\sharp}$ of f for A in L.

Problem statement

Can we automatically *synthesize* an abstract transformer for any operations?

Given the

- concrete semantics Φ_f of a concrete transformer f,
- description of an abstract domain $(A, \sqsubseteq, \sqcup)$, and its relation to the concrete domain (α, γ, β), and
- a domain-specific language L,

synthesize **a best** abstract transformer $\widehat{f^\sharp}$ of f for A in L.

$$\widehat{f^\sharp} = \lambda a : \sqcup\{\beta(f(c_i)) \mid c_i \in \gamma(a)\}$$

Problem statement

Can we automatically *synthesize* an abstract transformer for any operations?

Given the

- concrete semantics Φ_f of a concrete transformer f,
- description of an abstract domain $(A, \sqsubseteq, \sqcup)$, and its relation to the concrete domain (α, γ, β), and
- a domain-specific language L,

synthesize **a best** abstract transformer $\widehat{f^\sharp}$ of f for A in L.

We build a tool, अमूर्त (AMURTH[a]), that solves the above problem.

[a]Pankaj Kumar Kalita, Sujt Muduli, Loris D'Antoni,Thomas Reps, Subhajit Roy, **Synthesizing Abstract Transformers**, OOPSLA 2022

Challenges

1. $\widehat{f^{\sharp}}$ may not be computable.

Challenges

1. $\widehat{f^\sharp}$ may not be computable.
2. $\widehat{f^\sharp}$ may not be expressible in L.

Challenges

1. $\widehat{f^\sharp}$ may not be computable.
2. $\widehat{f^\sharp}$ may not be expressible in L.
3. Precision defines a partial ordering on abstract transformers, so $f^\sharp \in L$ may not be unique.

Algorithm Overview

- AMURTH uses counterexample-guided inductive synthesis (CEGIS) strategy.

Algorithm Overview

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

Algorithm Overview

- AMURTH uses counterexample-guided inductive synthesis (CEGIS) strategy.
- Attempts to meet the dual objectives of soundness and precision
- Correspondingly, the algorithm is guided by two kinds of examples,

Algorithm Overview

- AMURTH uses counterexample-guided inductive synthesis (CEGIS) strategy.
- Attempts to meet the dual objectives of soundness and precision
- Correspondingly, the algorithm is guided by two kinds of examples,
  - **Positive Example (E^+)**: $\boxed{\langle a, c' \rangle \text{ such that, } a \in A,\ c' \in C \text{ and } c' \in \gamma(\hat{f}^\sharp(a))}$

Algorithm Overview

- AMURTH uses counterexample-guided inductive synthesis (CEGIS) strategy.
- Attempts to meet the dual objectives of soundness and precision
- Correspondingly, the algorithm is guided by two kinds of examples,
  - **Positive Example (E^+)**: $\langle a, c' \rangle$ such that, $a \in A$, $c' \in C$ and $c' \in \gamma(\hat{f}^\sharp(a))$
    - $\langle [5,9], 10 \rangle$ (`increment` in interval domain)

Algorithm Overview

- AMURTH uses counterexample-guided inductive synthesis (CEGIS) strategy.
- Attempts to meet the dual objectives of soundness and precision
- Correspondingly, the algorithm is guided by two kinds of examples,
  - **Positive Example (E^+)**: $\langle a, c' \rangle$ such that, $a \in A$, $c' \in C$ and $c' \in \gamma(\hat{f}^\sharp(a))$
    - $\langle [5,9], 10 \rangle$ (`increment` in interval domain)
  - **Negative Example (E^-)**:
    $\langle a, c' \rangle$ such that, $a \in A$, $c' \in C$ and $\exists\, \hat{f}^\sharp_L \in \mathcal{L}$ such that, $c' \notin \gamma(\hat{f}^\sharp_L(a))$

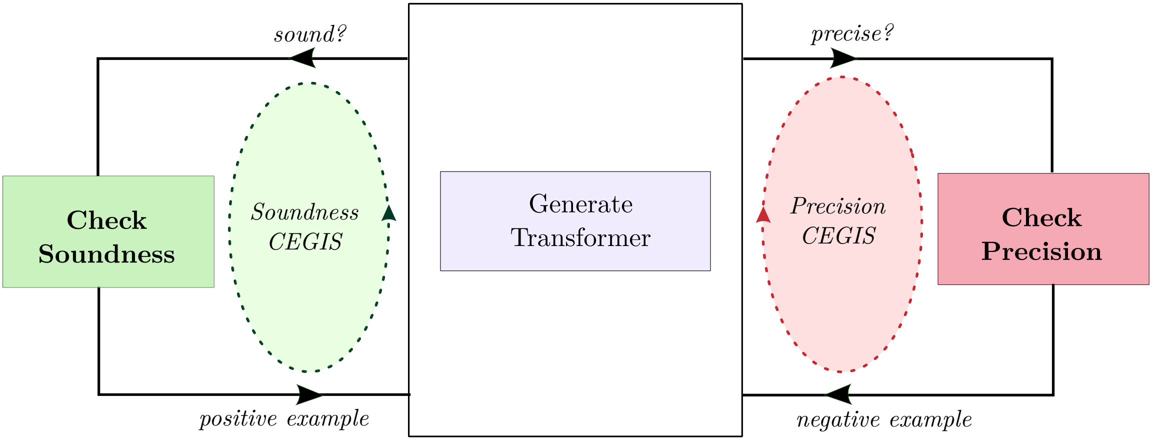
Algorithm Overview

- AMURTH uses counterexample-guided inductive synthesis (CEGIS) strategy.
- Attempts to meet the dual objectives of soundness and precision
- Correspondingly, the algorithm is guided by two kinds of examples,
  - **Positive Example (E^+)**: $\langle a, c' \rangle$ such that, $a \in A$, $c' \in C$ and $c' \in \gamma(\hat{f}^\sharp(a))$
    - $\langle [5,9], 10 \rangle$ (`increment` in interval domain)
  - **Negative Example (E^-)**:
    $\langle a, c' \rangle$ such that, $a \in A$, $c' \in C$ and $\exists\, \hat{f}_L^\sharp \in \mathcal{L}$ such that, $c' \notin \gamma(\hat{f}_L^\sharp(a))$
    - $\langle [5,9], 2 \rangle$ (`increment` in interval domain)

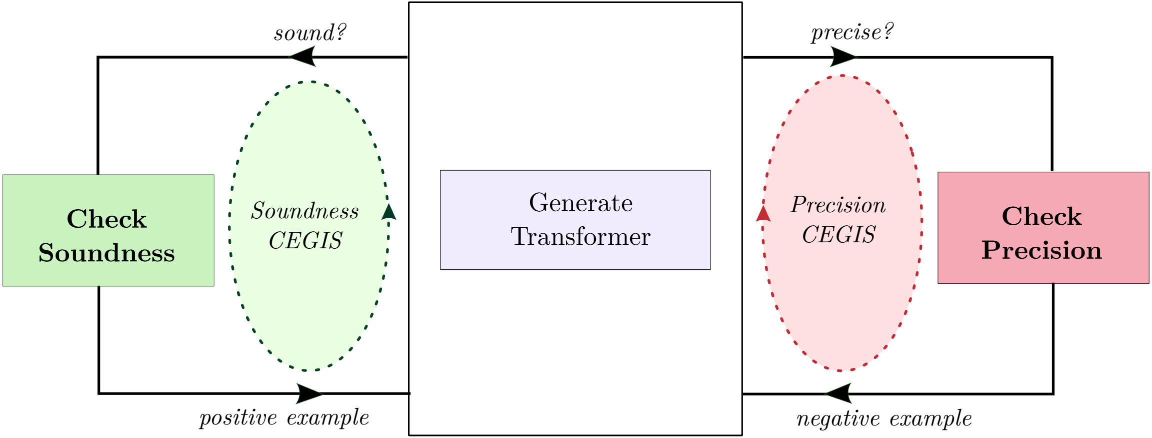
Algorithm Overview

- AMURTH uses counterexample-guided inductive synthesis (CEGIS) strategy.
- Attempts to meet the dual objectives of soundness and precision
- Correspondingly, the algorithm is guided by two kinds of examples,
  - **Positive Example (E^+)**: $\langle a, c' \rangle$ such that, $a \in A$, $c' \in C$ and $c' \in \gamma(\hat{f}^\sharp(a))$
    - $\langle [5,9], 10 \rangle$ (`increment` in interval domain)
  - **Negative Example (E^-)**:
    $\langle a, c' \rangle$ such that, $a \in A$, $c' \in C$ and $\exists\, \hat{f}^\sharp_L \in \mathcal{L}$ such that, $c' \notin \gamma(\hat{f}^\sharp_L(a))$
    - $\langle [5,9], 2 \rangle$ (`increment` in interval domain)
- **Soundness** and **precision** verifiers drive two CEGIS loops.

Algorithm


sound?
precise?
Check
Soundness
Soundness
CEGIS
Generate
Transformer
Precision
CEGIS
Check
Precision
positive example
negative example

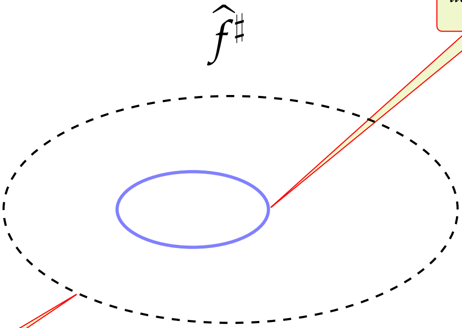
Algorithm



Additional algorithmic components are needed! (see the paper for details)

Amurth in action!

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

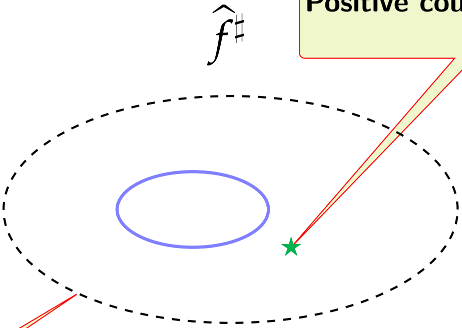


$$\widehat{f}^\sharp = \lambda a : \sqcup\{\beta(f(c_i)) \mid c_i \in \gamma(a)\}$$

Amurth in action!

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

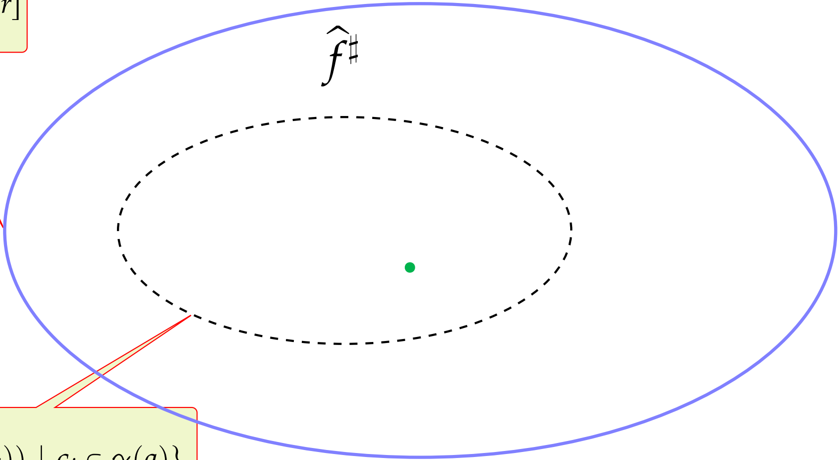
Positive counterexample: $\langle [0,5], 3 \rangle$



$$\widehat{f}^\sharp = \lambda a : \sqcup\{\beta(f(c_i)) \mid c_i \in \gamma(a)\}$$

Amurth in action!

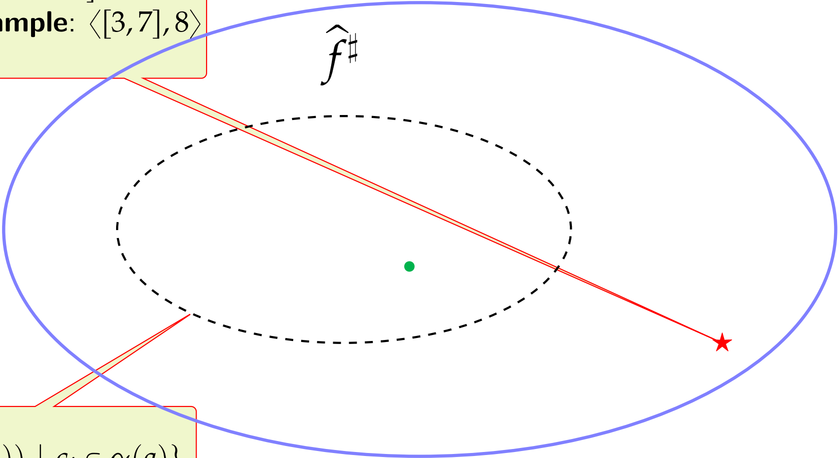
$f^{\sharp}_{abs} \leftarrow \lambda a.[0, a.l + a.r]$



$\widehat{f}^{\sharp} = \lambda a : \sqcup\{\beta(f(c_i)) \mid c_i \in \gamma(a)\}$

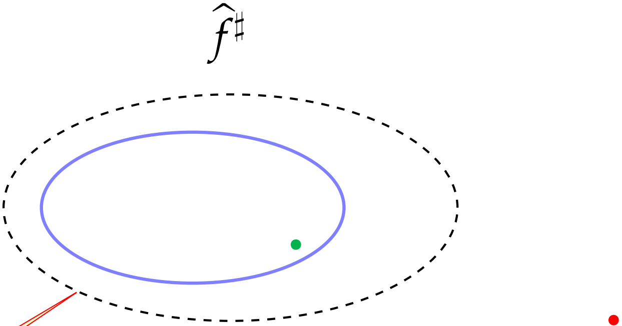
$$f^\sharp_{abs} \leftarrow \lambda a.[0, a.l + a.r]$$

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



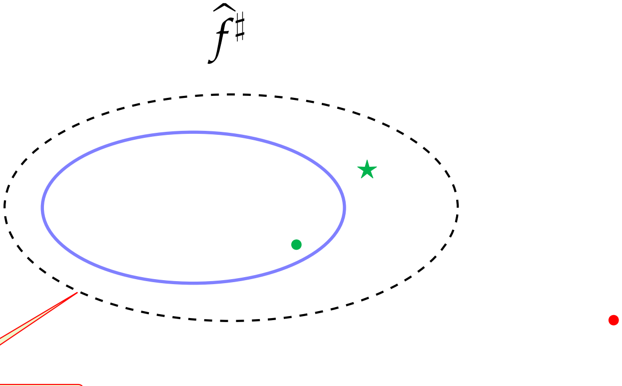
$$\widehat{f}^\sharp = \lambda a : \sqcup \{\beta(f(c_i)) \mid c_i \in \gamma(a)\}$$

Amurth in action!



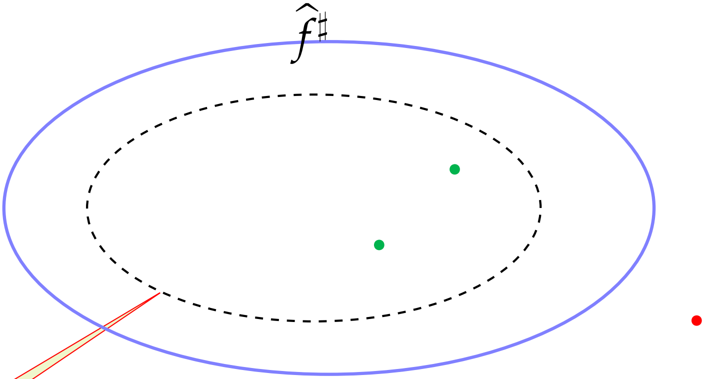
$$\widehat{f}^\sharp = \lambda a : \sqcup\{\beta(f(c_i)) \mid c_i \in \gamma(a)\}$$

Amurth in action!



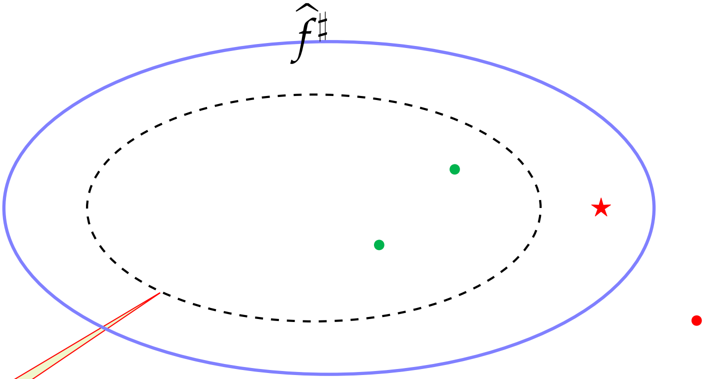
$$\widehat{f}^\sharp = \lambda a : \sqcup\{\beta(f(c_i)) \mid c_i \in \gamma(a)\}$$

Amurth in action!



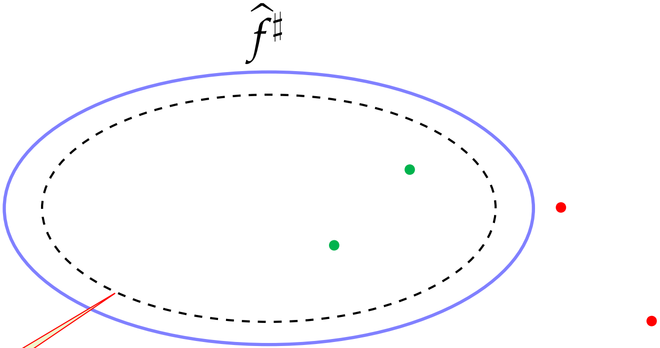
$$\widehat{f}^\sharp = \lambda a : \sqcup\{\beta(f(c_i)) \mid c_i \in \gamma(a)\}$$

Amurth in action!



$$\widehat{f}^{\sharp} = \lambda a : \sqcup\{\beta(f(c_i)) \mid c_i \in \gamma(a)\}$$

Amurth in action!



$$\widehat{f}^{\sharp} = \lambda a : \sqcup \{ \beta(f(c_i)) \mid c_i \in \gamma(a) \}$$

Claims

Theorem 1

If Algorithm terminates, it returns a best $\mathbb{L}$-transformer for the concrete function f.

Theorem 2

If the DSL $\mathbb{L}$ is finite, algorithm always terminates.

This paper

From single domain to reduced product domains.

Odd and Even interval domain

- Odd interval domain:

$$\alpha_O(S) = [isOdd(min(S)) \; ? \; min(S) \; : \; min(S) - 1,$$
$$isOdd(max(S)) \; ? \; max(S) \; : \; max(S) + 1]$$
$$\alpha_O(\{4\}) = [3, 5]$$
$$\gamma_O([l, r]) = \{x \mid l \leqslant x \leqslant r\}$$

- Even interval domain:

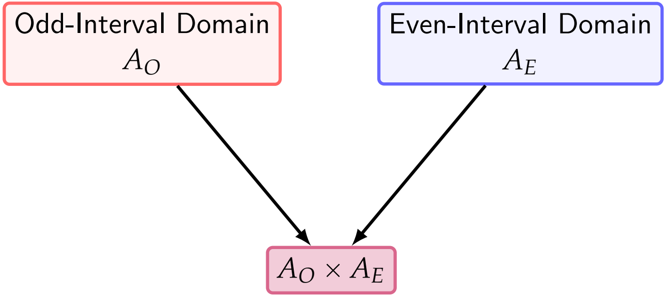
$$\alpha_E(S) = [isEven(min(S)) \; ? \; min(S) \; : \; min(S) - 1,$$
$$isEven(max(S)) \; ? \; max(S) \; : \; max(S) + 1]$$
$$\alpha_E(\{5\}) = [4, 6]$$
$$\gamma_E([l, r]) = \{x \mid l \leqslant x \leqslant r\}$$

Product Domain

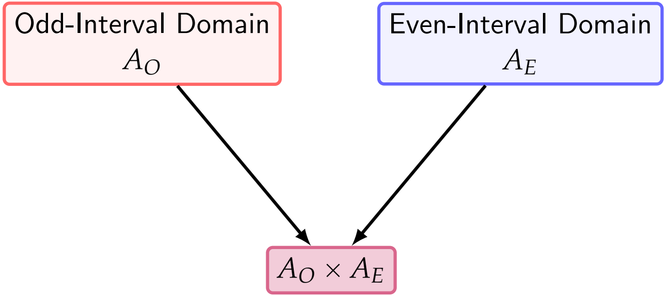
Odd-Interval Domain
A_O

Even-Interval Domain
A_E

Product Domain



Product Domain



$$a_o \in A_O,\ a_e \in A_E.$$

$$\gamma_{O\times E}(\langle a_o, a_e\rangle) = \gamma_O(a_o) \cap \gamma_E(a_e)$$

Transformers for `increment`

$$\texttt{inc}(\texttt{c}) = \texttt{c} + 1$$

Transformers for `increment`

$$\mathtt{inc}(\mathtt{c}) = \mathtt{c} + 1$$

$$\mathtt{inc}^{\sharp D}(\langle \mathtt{o}, \mathtt{e} \rangle) =$$

Transformers for `increment`

$$\texttt{inc}(\texttt{c}) = \texttt{c} + \texttt{1}$$

$$\texttt{inc}^{\sharp D}(\langle \texttt{o}, \texttt{e} \rangle) = \langle [\underbrace{\texttt{o.l , o.r + 2}}_{\text{Odd-Interval}}], [\underbrace{\texttt{e.l , e.r + 2}}_{\text{Even-Interval}}] \rangle$$

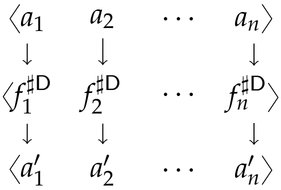
Transformers for `increment`

$$\texttt{inc}(\texttt{c}) = \texttt{c} + 1$$

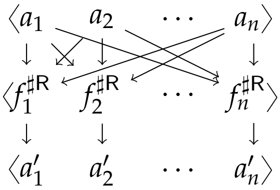
$$\texttt{inc}^{\sharp D}(\langle \texttt{o}, \texttt{e} \rangle) = \langle \overbrace{[\ \texttt{o.l}\ ,\ \texttt{o.r + 2}\ ]}^{\text{Odd-Interval}}, \overbrace{[\ \texttt{e.l}\ ,\ \texttt{e.r + 2}\ ]}^{\text{Even-Interval}} \rangle$$

$$\texttt{inc}^{\sharp D}(\langle [5, 7], [4, 6] \rangle) = \langle [5, 9], [4, 8] \rangle$$

Direct Product vs Reduced Product



Direct-product transformers



Reduced-product transformers

Reduced-Product Transformer

$$\mathtt{inc}(\mathtt{c}) = \mathtt{c} + 1$$

$$\mathtt{inc}^{\sharp D}(\langle \mathtt{o}, \mathtt{e} \rangle) = \langle [\overbrace{\mathtt{o.l}\ ,\ \mathtt{o.r + 2}}^{\text{Odd-Interval}}], [\overbrace{\mathtt{e.l}\ ,\ \mathtt{e.r + 2}}^{\text{Even-Interval}}] \rangle$$

$$\mathtt{inc}^{\sharp D}(\langle [5,7], [4,6] \rangle) = \langle [5,9], [4,8] \rangle$$

Reduced-Product Transformer

$$\texttt{inc}(\texttt{c}) = \texttt{c} + 1$$

$$\texttt{inc}^{\sharp D}(\langle \texttt{o}, \texttt{e} \rangle) = \langle [\overbrace{\texttt{o.l , o.r + 2}}^{\text{Odd-Interval}}], [\overbrace{\texttt{e.l , e.r + 2}}^{\text{Even-Interval}}] \rangle$$

$$\texttt{inc}^{\sharp D}(\langle [5, 7], [4, 6] \rangle) = \langle [5, 9], [4, 8] \rangle$$

$$\texttt{inc}^{\sharp R}(\langle \texttt{o}, \texttt{e} \rangle) = \langle [\overbrace{\texttt{e.l + 1, e.r + 1}}^{\text{Odd Interval}}], [\overbrace{\texttt{o.l + 1, o.r + 1}}^{\text{Even-Interval}}] \rangle$$

Reduced-Product Transformer

$$\texttt{inc}(\texttt{c}) = \texttt{c} + 1$$

$$\texttt{inc}^{\sharp D}(\langle o, e\rangle) = \langle[\overbrace{\texttt{o.l , o.r + 2}}^{\text{Odd-Interval}}], [\overbrace{\texttt{e.l , e.r + 2}}^{\text{Even-Interval}}]\rangle$$

$$\texttt{inc}^{\sharp D}(\langle[5,7],[4,6]\rangle) = \langle[5,9],[4,8]\rangle$$

$$\texttt{inc}^{\sharp R}(\langle o, e\rangle) = \langle[\overbrace{\texttt{e.l + 1, e.r + 1}}^{\text{Odd Interval}}], [\overbrace{\texttt{o.l + 1, o.r + 1}}^{\text{Even-Interval}}]\rangle$$

$$\texttt{inc}^{\sharp R}(\langle[5,7],[4,6]\rangle) = \langle[5,7],[6,8]\rangle$$

Problem statement (Reduced-Product Domain)

Given the

- concrete semantics Φ_f of a concrete transformer f,
- description of an abstract domain $(A, \sqsubseteq, \sqcup)$, and its relation to the concrete domain (α, γ, β), and
- a domain-specific language L,

synthesize **a best** abstract transformer $\widehat{f^\sharp}$ of f for A in L.

Problem statement (Reduced-Product Domain)

Given the

- concrete semantics Φ_f of a concrete transformer f,
- description of abstract domains ($\langle A_1, \ldots, A_n \rangle, \langle \sqsubseteq_1, \ldots, \sqsubseteq_n \rangle, \langle \sqcup_1, \ldots, \sqcup_n \rangle$), and its relation to the concrete domains ($\langle \alpha_1, \ldots, \alpha_n \rangle, \langle \gamma_1, \ldots, \gamma_n \rangle$, $\langle \beta_1, \ldots, \beta_n \rangle$), and
- domain-specific languages $\mathcal{L}_1, \ldots, \mathcal{L}_n$,

synthesize a sound and most precise *reduced* abstract transformer $f^{\sharp\mathsf{R}} : \langle f_1^{\sharp\mathsf{R}}, f_2^{\sharp\mathsf{R}}, \ldots, f_n^{\sharp\mathsf{R}} \rangle$

Problem statement (Reduced-Product Domain)

Given the

- concrete semantics Φ_f of a concrete transformer f,
- description of abstract domains $(\langle A_1, \ldots, A_n \rangle, \langle \sqsubseteq_1, \ldots, \sqsubseteq_n \rangle, \langle \sqcup_1, \ldots, \sqcup_n \rangle)$, and its relation to the concrete domains $(\langle \alpha_1, \ldots, \alpha_n \rangle, \langle \gamma_1, \ldots, \gamma_n \rangle, \langle \beta_1, \ldots, \beta_n \rangle)$, and
- domain-specific languages $\mathcal{L}_1, \ldots, \mathcal{L}_n$,

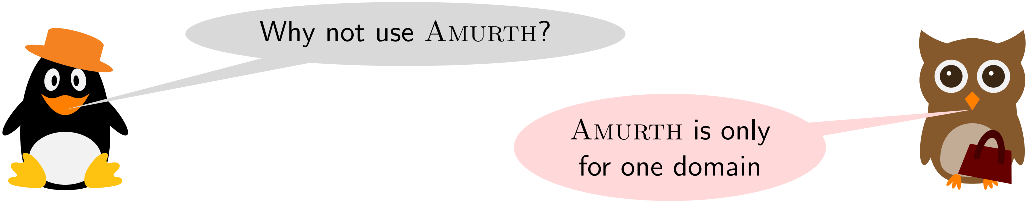
synthesize a sound and most precise *reduced* abstract transformer $f^{\sharp\mathsf{R}} : \langle f_1^{\sharp\mathsf{R}}, f_2^{\sharp\mathsf{R}}, \ldots, f_n^{\sharp\mathsf{R}} \rangle$

We build AMURTH2 to solve the above problem.

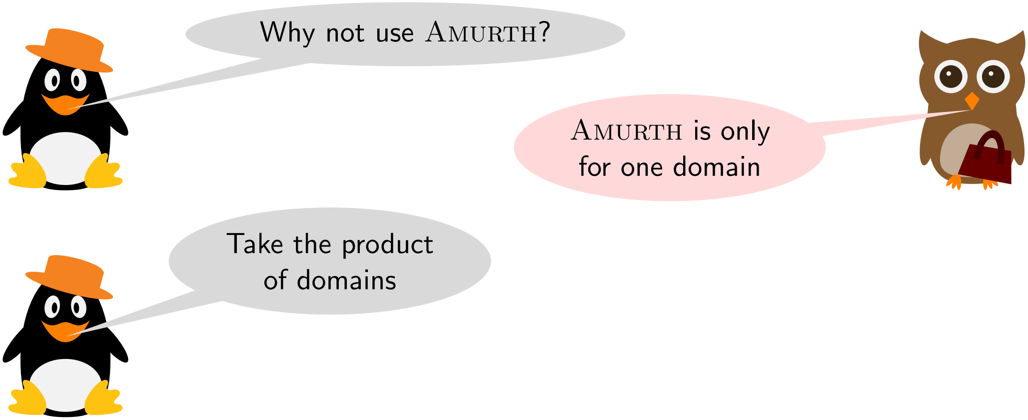
Why Not Amurth?

Why not use AMURTH?

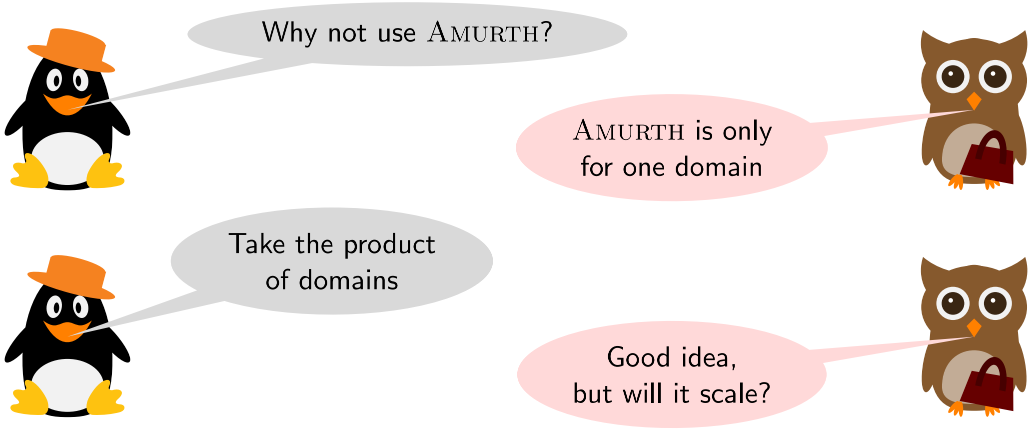
Why Not Amurth?



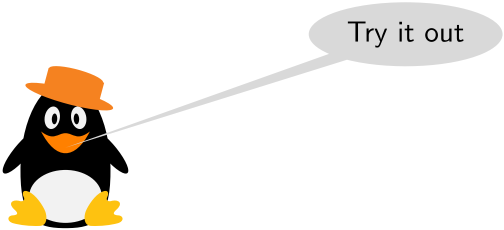
Why Not Amurth?



Why Not Amurth?

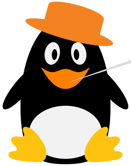


Why Not Amurth?



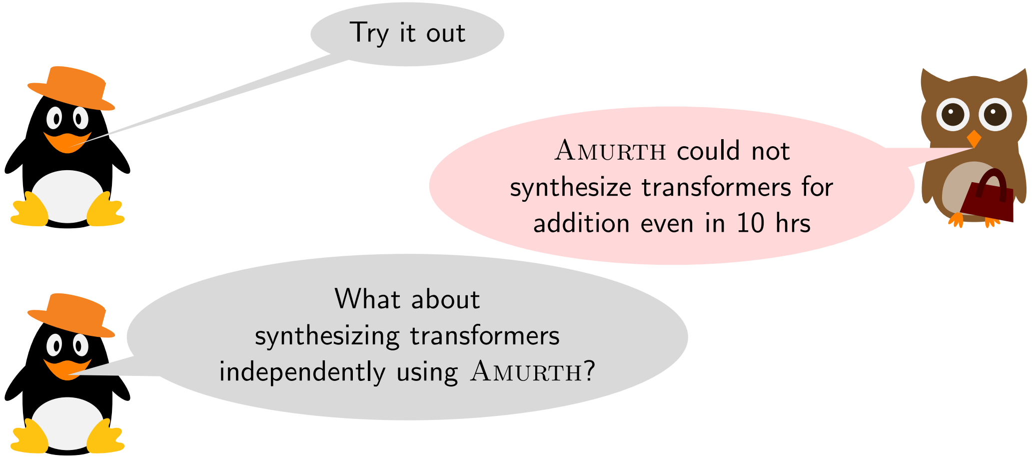
Why Not Amurth?

Try it out

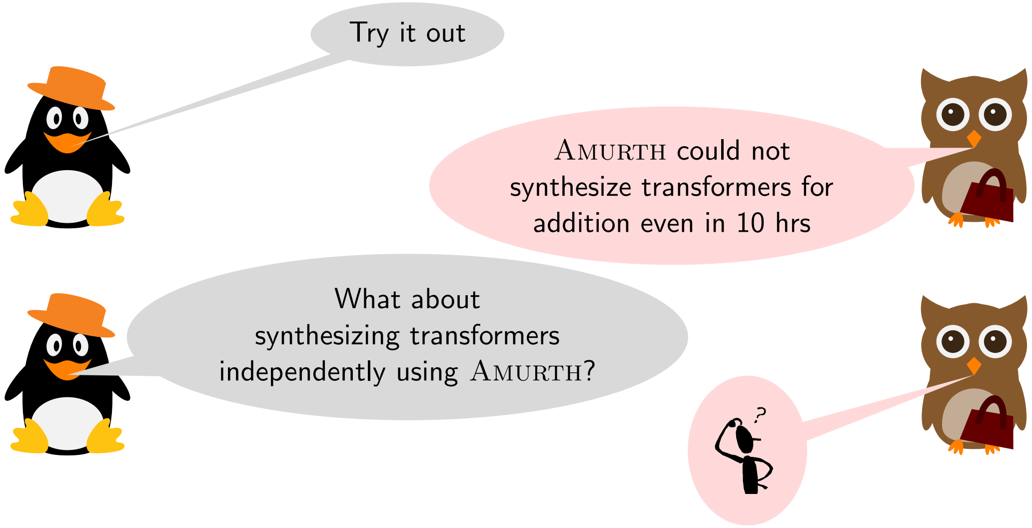


AMURTH could not synthesize transformers for addition even in 10 hrs

Why Not Amurth?



Why Not Amurth?



Why Not Amurth?

$\texttt{inc}^{\sharp}_{\texttt{odd}}(\texttt{o.l}, \texttt{o.r}, \texttt{e.l}, \texttt{e.r});$

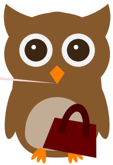
$\texttt{inc}^{\sharp}_{\texttt{even}}(\texttt{o.l}, \texttt{o.r}, \texttt{e.l}, \texttt{e.r});$

Why Not Amurth?

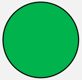
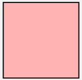
$$\texttt{inc}^{\sharp}_{\texttt{odd}}(\texttt{o.l}, \texttt{o.r}, \texttt{e.l}, \texttt{e.r});$$

$$\texttt{inc}^{\sharp}_{\texttt{even}}(\texttt{o.l}, \texttt{o.r}, \texttt{e.l}, \texttt{e.r});$$

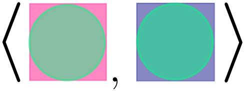
$$\texttt{best}_{\texttt{i}} + \texttt{best}_{\texttt{k}} \neq \texttt{best}$$



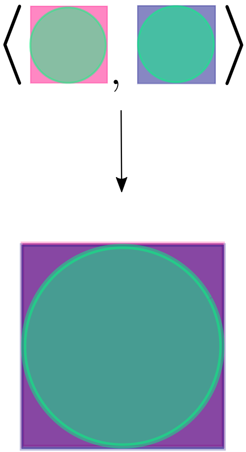
$\textbf{best}_i + \textbf{best}_k \neq \textbf{best}$

Consider, abstracting  using  abstraction.

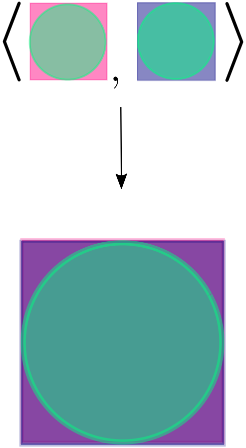
$\mathbf{best}_i + \mathbf{best}_k \neq \mathbf{best}$

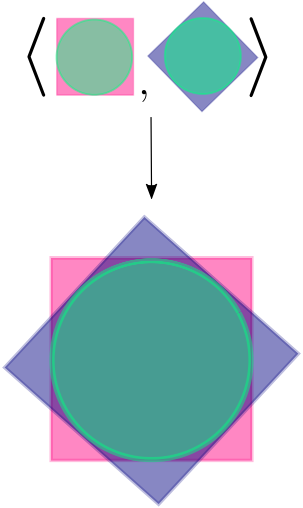


$\mathbf{best}_i + \mathbf{best}_k \neq \mathbf{best}$

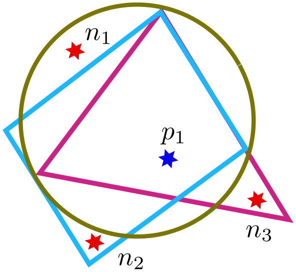


$\text{best}_i + \text{best}_k \neq \text{best}$

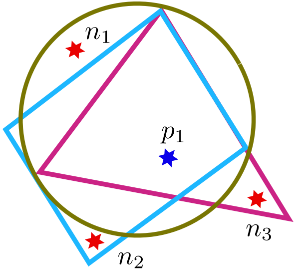




Positive and Negative Example

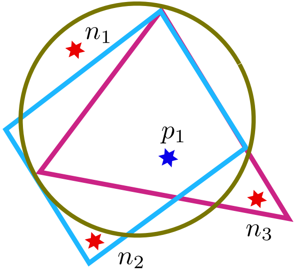


Positive and Negative Example



$\langle\langle a_1, a_2, \ldots, a_n\rangle, c'\rangle$ is a

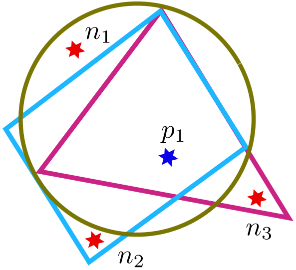
Positive and Negative Example



$\langle\langle a_1, a_2, \ldots, a_n\rangle, c'\rangle$ is a

- *positive example*, if it is contained in all transformers (shared)

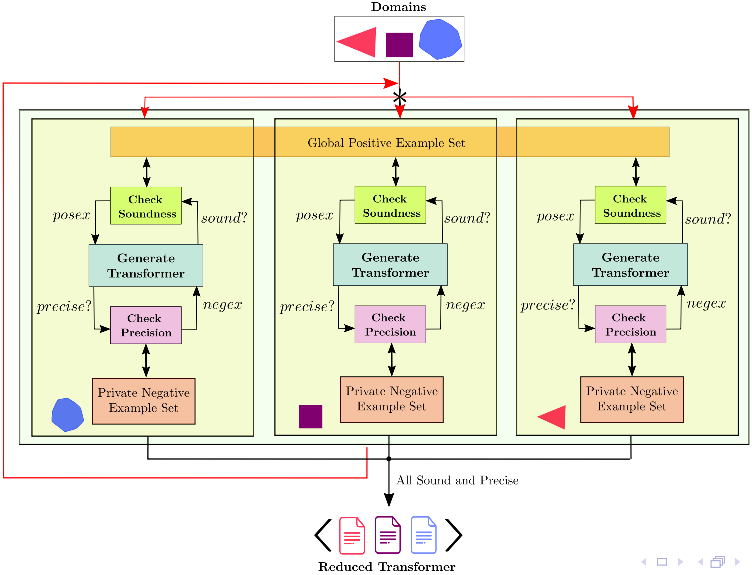
Positive and Negative Example



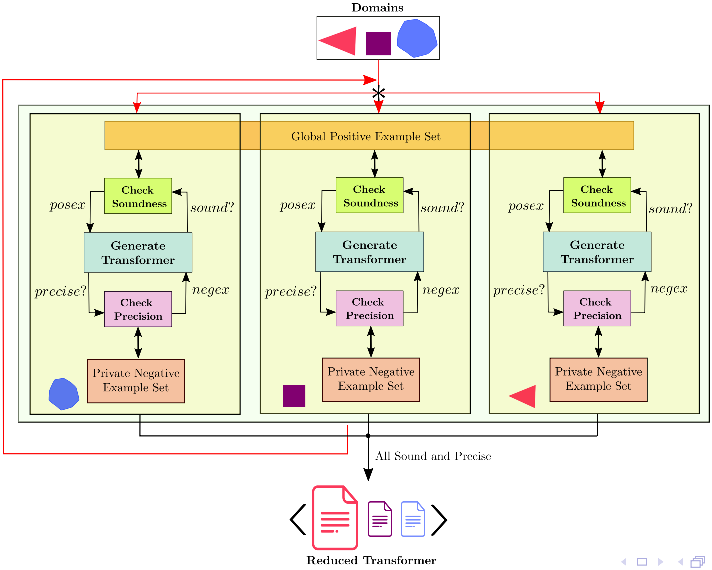
$\langle\langle a_1, a_2, \ldots, a_n\rangle, c'\rangle$ is a

- *positive example*, if it is contained in all transformers (shared)
- *negative example*, if there is any one domain whose transformer excludes it (private)

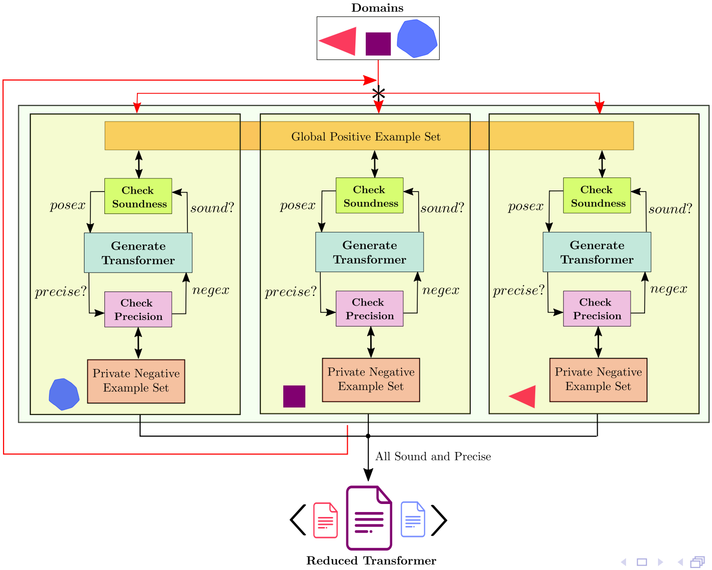
Algorithm


Domains
Global Positive Example Set
Check Soundness
posex
sound?
Generate Transformer
precise?
negex
Check Precision
Private Negative Example Set
Check Soundness
posex
sound?
Generate Transformer
precise?
negex
Check Precision
Private Negative Example Set
Check Soundness
posex
sound?
Generate Transformer
precise?
negex
Check Precision
Private Negative Example Set
All Sound and Precise
Reduced Transformer

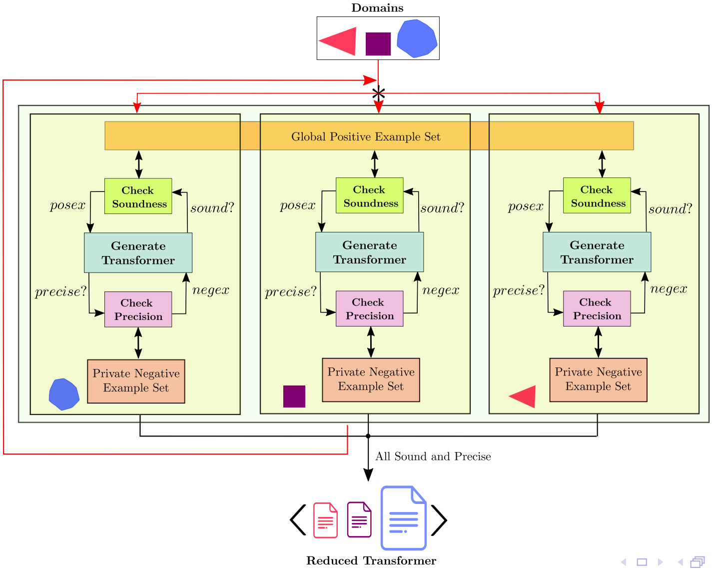
Algorithm


Domains
Global Positive Example Set
Check Soundness
posex
sound?
Generate Transformer
precise?
negex
Check Precision
Private Negative Example Set
Check Soundness
posex
sound?
Generate Transformer
precise?
negex
Check Precision
Private Negative Example Set
Check Soundness
posex
sound?
Generate Transformer
precise?
negex
Check Precision
Private Negative Example Set
All Sound and Precise
Reduced Transformer

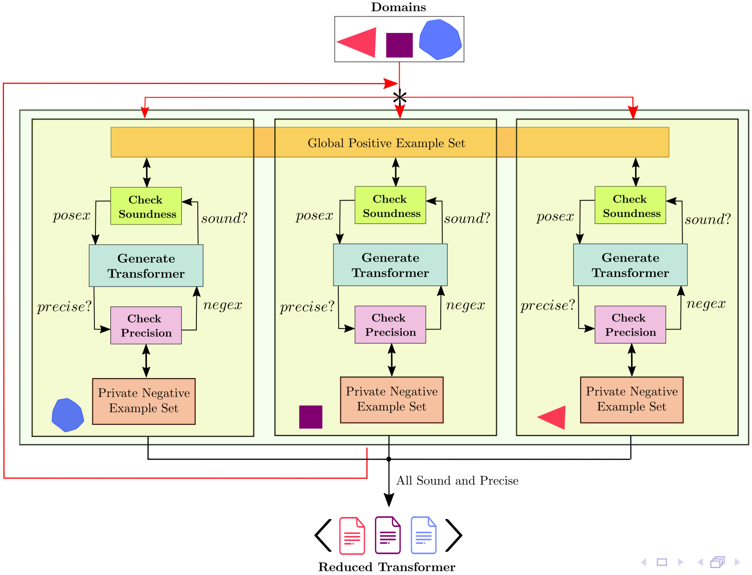
Algorithm


Domains
Global Positive Example Set
Check Soundness
posex
sound?
Generate Transformer
precise?
negex
Check Precision
Private Negative Example Set
All Sound and Precise
Reduced Transformer

Algorithm


Domains
Global Positive Example Set
Check Soundness
posex
sound?
Generate Transformer
precise?
negex
Check Precision
Private Negative Example Set
Check Soundness
posex
sound?
Generate Transformer
precise?
negex
Check Precision
Private Negative Example Set
Check Soundness
posex
sound?
Generate Transformer
precise?
negex
Check Precision
Private Negative Example Set
All Sound and Precise
Reduced Transformer

Algorithm


Domains
Global Positive Example Set
Check Soundness
posex
sound?
Generate Transformer
precise?
negex
Check Precision
Private Negative Example Set
Check Soundness
posex
sound?
Generate Transformer
precise?
negex
Check Precision
Private Negative Example Set
Check Soundness
posex
sound?
Generate Transformer
precise?
negex
Check Precision
Private Negative Example Set
All Sound and Precise
Reduced Transformer

k-Precision

- Synthesize k transformers at a time

k-Precision

- Synthesize k transformers at a time
- We showed 1-precision: iterating over each domain

k-Precision

- Synthesize k transformers at a time
- We showed 1-precision: iterating over each domain
- Conjecture: k-precision can be weaker than $(k+1)$-precision

Claims

Theorem 3

If Algorithm terminates, it returns a sound L_k-transformer for the concrete function f for each domain k.

Theorem 4

Synthesized transformers will be 1-precise.

Theorem 5

Even though each DSLs, $\langle L_1, \ldots, L_n \rangle$ is finite, algorithm might not always terminate. However, in practice, no case of non-termination detected.

Experiment

- Performed on two product domains, JSAI and SAFE (part of SAFE_{str}[1])
- Evaluated on six operations, i.e., `concat`, `contains`, `toLower`, `toUpper`, `charAt`, and `trim`
- Except `contains`, synthesized transformers are more precise than the manually written ones

[1] R. Amadini, A. Jordan, G. Gange, F. Gauthier, P. Schachte, H. Søndergaard, P. J. Stuckey & C. Zhang, "Combining String Abstract Domains for JavaScript Analysis: An Evaluation", in TACAS'17

SAFE Domain

- $\mathcal{SS}_k$ Domain: Set of strings of size k

$$\alpha_{\mathcal{SS}_k}(C) = \begin{cases} C & |C| \leqslant k \\ \top_{\mathcal{SS}_k} & otherwise \end{cases}$$

$$\gamma_{\mathcal{SS}_k}(A) = \begin{cases} A & A \neq \top_{\mathcal{SS}_k} \\ \Sigma^* & otherwise \end{cases}$$

SAFE Domain

- $\mathcal{SS}_k$ Domain: Set of strings of size k

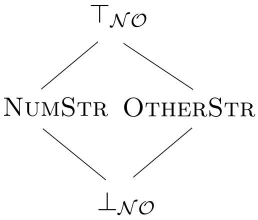
$$\alpha_{\mathcal{SS}_k}(C) = \begin{cases} C & |C| \leqslant k \\ \top_{\mathcal{SS}_k} & otherwise \end{cases}$$

$$\gamma_{\mathcal{SS}_k}(A) = \begin{cases} A & A \neq \top_{\mathcal{SS}_k} \\ \Sigma^* & otherwise \end{cases}$$

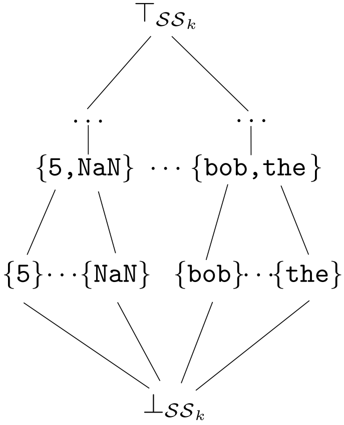
- $\mathcal{NO}$ Domain:

$$\text{Number} : -3,\ 0,\ 2,\ 2.35,\ -0.23,\ \texttt{NaN},\ \ldots$$
$$\text{Others} : \text{Everything else}$$

SAFE Domain



Lattice for $\mathcal{NO}$



Lattice for $\mathcal{SS}_k$

trim in SAFE

```
trim♯D_SAFE(arg₁) {
  out ← arg₁
  if(arg₁.ssk ∉ {⊤_SS_k, ⊥_SS_k}) {
    sset ← ∅
    for(x ← arg₁.ssk)
      sset ← sset ∪ {trim(x)}
    out.ssk ← α_SS_k(sset)
  }
  if(arg₁.no = OtherStr)
    return ⟨out.ssk, ⊤_NO⟩
  else
    return out
}
```

Manually written

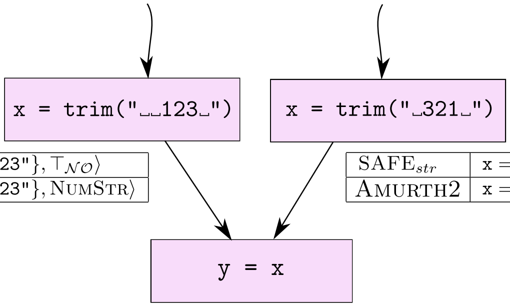
```
trim♯R_SAFE(arg₁) {
  if(arg₁.ssk ∉ {⊤_SS_k, ⊥_SS_k}) {
    sset ← ∅
    for(x ← arg₁.ssk)
      sset ← sset ∪ {trim(x)}
    out.no ← α_NO(sset)
    out.ssk ← α_SS_k(sset)
    return out
  } else {
    if(arg₁.no = OtherStr)
      return ⟨arg₁.ssk, ⊤_NO⟩
    else
      return arg₁
  }
}
```

Synthesized by Amurth2

$\texttt{trim}^{\sharp D}_{\texttt{SAFE}}(``␣␣123␣") = \langle ``123", \top_{\mathcal{NO}} \rangle$

$\texttt{trim}^{\sharp R}_{\texttt{SAFE}}(``␣␣123␣") = \langle ``123", \textsc{NumStr} \rangle$

`trim` in SAFE



SAFE$_{str}$	x $= \langle \{\texttt{"123"}\}, \top_{\mathcal{NO}} \rangle$
AMURTH2	x $= \langle \{\texttt{"123"}\}, \textsc{NumStr} \rangle$

SAFE$_{str}$	x $= \langle \{\texttt{"321"}\}, \top_{\mathcal{NO}} \rangle$
AMURTH2	x $= \langle \{\texttt{"321"}\}, \textsc{NumStr} \rangle$

SAFE$_{str}$	y $= \langle \top_{\mathcal{SS}_k}, \top_{\mathcal{NO}} \rangle$
AMURTH2	y $= \langle \top_{\mathcal{SS}_k}, \textsc{NumStr} \rangle$

Conclusion

- Synthesis of abstract transformer is hard, but synthesis of reduced product transformer is even more challenging

Conclusion

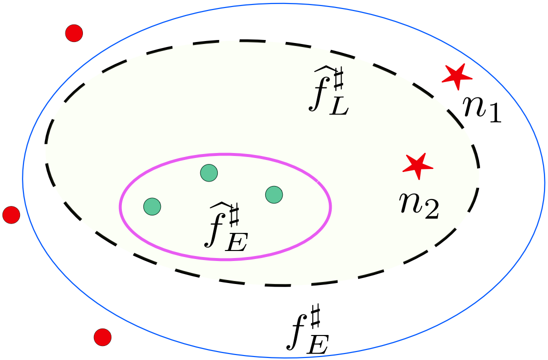
- Synthesis of abstract transformer is hard, but synthesis of reduced product transformer is even more challenging
- Reduced product transformers synthesized by AMURTH2 are more precise than the manually written ones

Acknowledgments

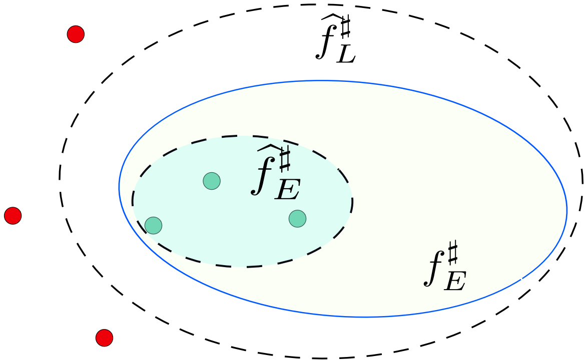
- Wonderful SAS reviewers
- Intel for Intel India Research Fellowship
- Research-I foundation of IIT Kanpur
- SIGPLAN PAC Funding

QUESTIONS

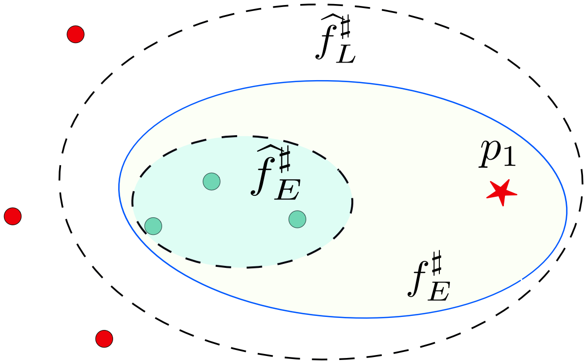
Failed Consistency



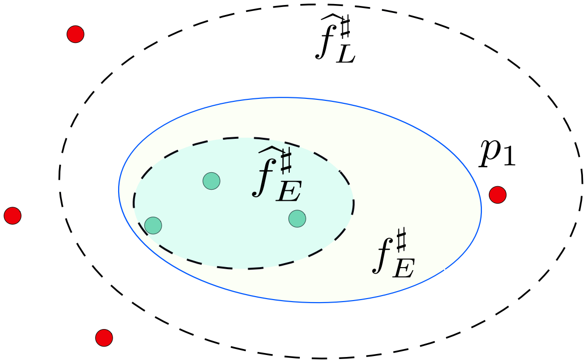
Failed Consistency



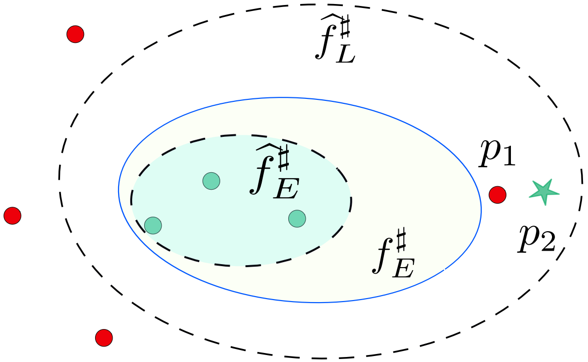
Failed Consistency



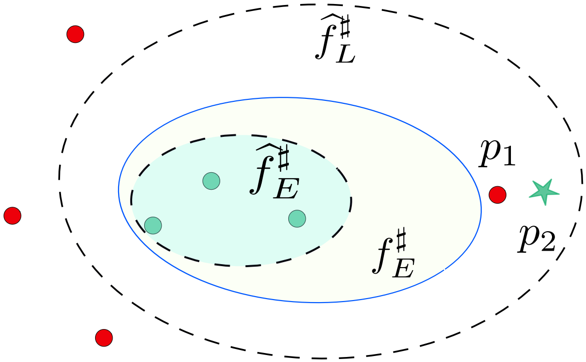
Failed Consistency



Failed Consistency

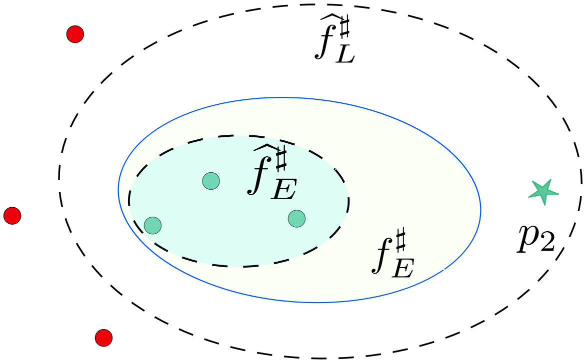


Failed Consistency



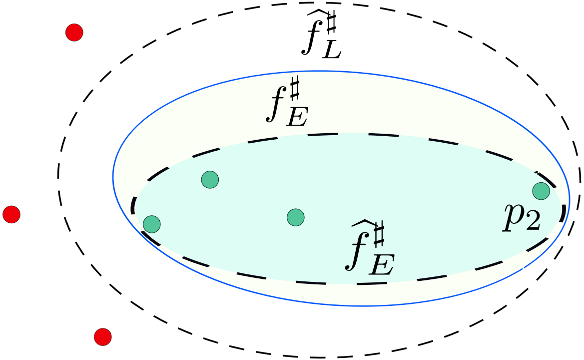
Inconsistent: no $f^{\sharp}_{E} \in L$ that satisfies all positive and negative examples.

Failed Consistency



Occam's razor

Failed Consistency



Occam's razor

Soundness Check

$$\exists\langle a, c'\rangle, \text{ where } a \in A, \text{ and } c' \in C, \text{ such that.}$$
$$\exists c \in C, c \in \gamma(a) \wedge \Phi_f(c, c') \wedge c' \notin \gamma(f_E^\sharp(a)) \tag{1}$$

Let us now define the interface:

$$\textsc{CheckSoundness}(f_E^\sharp, f) = \begin{cases} \textit{False}, \langle a, c'\rangle & \text{if (1) is SAT} \\ \textit{True}, \_ & \text{otherwise} \end{cases} \tag{2}$$

Precision Check

$$\exists h_L^\sharp \in \mathcal{L}, \langle a, c' \rangle. \text{ where } a \in A, \text{ and } c' \in C, \text{ such that.}$$
$$sat^+(h_L^\sharp, E^+) \wedge sat^-(h_L^\sharp, E^- \cup \{\langle a, c' \rangle\}) \wedge \neg sat^-(f_E^\sharp, \{\langle a, c' \rangle\}) \quad (3)$$

We can now define the CheckPrecision interface:

$$\textsc{CheckPrecision}(f_E^\sharp,\ E^+,\ E^-) = \begin{cases} \mathit{False}, \langle a, c' \rangle & \text{if (3) is SAT} \\ \mathit{True}, \_ & \text{otherwise} \end{cases} \quad (4)$$

Soundness Check (Reduced)

$$\exists c \in \mathcal{C}.\ (\bigwedge_{i=1}^{n} c \in \gamma_i(a_i)) \wedge c' = f(c) \wedge (c' \notin \gamma_k(f_k^{\sharp\mathsf{R}}(a_1, \ldots, a_n)) \quad (5)$$

$$\textsc{CheckSoundness}(f_k^{\sharp\mathsf{R}}, f) = \begin{cases} \mathit{False}, \langle\langle a_1, \ldots, a_n\rangle, c'\rangle & \text{if Eqn 5 is SAT} \\ \mathit{True}, \_ & \text{otherwise} \end{cases}$$

Precision Check (Reduced)

$$\exists h_i^{\sharp R}, \langle\langle a_1, \ldots, a_n\rangle, c'\rangle, \text{s.t. } satI^{+}(h_i^{\sharp R}, E^{+}) \ \wedge \\ satI^{-}(h_i^{\sharp R}, E_i^{-} \cup \{\langle\langle a_1, \ldots, a_n\rangle, c'\rangle\}) \ \wedge \\ \neg sat^{-}(\langle f_1^{\sharp R}, \ldots, f_n^{\sharp R}\rangle, \{\langle\langle a_1, \ldots, a_n\rangle, c'\rangle\}) \tag{6}$$

$$\textsc{CheckPrecision}(\langle f_1^{\sharp R} \ldots f_n^{\sharp R}\rangle, f, i, E^{+}, E_i^{-}) = \begin{cases} \mathit{False}, \langle\langle a_1, \ldots, a_n\rangle, c'\rangle & \text{if Eqn 6 is SAT} \\ \mathit{True}, \_ & \text{otherwise} \end{cases}$$