Babble: learning better abstractions with e-graphs and anti-unification

david cao et al. (POPL 2023)

Presented by JS. Kwon

IS661 Paper Review

Background: Program Library

```
int Max(int X, int Y) {
  if (X > Y)
      return X;
  else
      return Y;
}

int main() {
  return Max(3, 8);
}
```

User-defined Function 1

Background: Program Library

```
int Max(int X, int Y) {
  if (X > Y)
      return X;
  else
      return Y;
}

int main() {
  return Max(3, 8);
}
```

User-defined Function 1

```
#define Max (X, Y) ((X > Y) ? X : Y)

int main() {
  return Max(3, 8);
}
```

User-defined Function 2

Background: Program Library

```
int Max(int X, int Y) {
  if (X > Y)
       return X;
  else
       return Y;
}

int main() {
  return Max(3, 8);
}
```

```
#define Max (X, Y) ((X > Y) ? X : Y)

int main() {
  return Max(3, 8);
}
```

User-defined Function 1 😡

User-defined Function 2 😡

Background: Program Library

```
int Max(int X, int Y) {
  if (X > Y)
      return X;
  else
      return Y;
}

int main() {
  return Max(3, 8);
}
```

User-defined Function 1 😠

```
#define Max (X, Y) ((X > Y) ? X : Y)

int main() {
  return Max(3, 8);
}
```

User-defined Function 2 😠

```
#include <algorithm.h>

int main() {
  return Max(3, 8);
}
```

Program Library 😁

Background: Library Learning

```
int foo1(int X, int Y) {
  int tmp1 = X + 1;
  int tmp2 = Y + 1;

  return tmp1 + tmp2 + 1;
}
```

Program 1

```
int foo2(int X, int Y) {
  int tmp1 = X + 1;

  return tmp1 + Y + 1;
}
```

Program 2

Background: Library Learning

```
int foo1(int X, int Y) {
  int tmp1 = X + 1;
  int tmp2 = Y + 1;

  return tmp1 + tmp2 + 1;
}
```

Program 1

```
int foo2(int X, int Y) {
  int tmp1 = X + 1;

  return tmp1 + Y + 1;
}
```

Program 2

Library Learning

Library:

```
int add_1(int X) {
  return X + 1;
}
```

Background: Library Learning

```
int foo1(int X, int Y) {
  int tmp1 = X + 1;
  int tmp2 = Y + 1;

  return tmp1 + tmp2 + 1;
}
```

Program 1

```
int foo2(int X, int Y) {
  int tmp1 = X + 1;

  return tmp1 + Y + 1;
}
```

Program 2

Library Learning →

Library:

```
int add_1(int X) {
  return X + 1;
}
```

Refactored Programs:

```
int foo1(int X, int Y) {
  return add_1(add_1(X) add_1(Y);
}
```

Program 1'

```
int foo2(int X, int Y) {
  return add_1(X) + add_1(Y);
}
```

Program 2'

Background: Library Learning

- **Automatically** extract common functionality into libraries

- **Abstract** common functionality to express their intent more **clearly** and **concisely**

Challenges of Library Learning

- **Challenge 1: Precise candidate generation**

```
int foo1(int X, int Y) {
  int tmp1 = X + 1;
  int tmp2 = Y + 1;

  return tmp1 + tmp2 + 2;
}
```

Program 1

```
int foo2(int X, int Y) {
  int tmp1 = X + 1;

  return tmp1 + Y + 2;
}
```

Program 2

Challenges of Library Learning

- **Challenge 1: Precise candidate generation**

```
int foo1(int X, int Y) {
  int tmp1 = X + 1;
  int tmp2 = Y + 1;

  return tmp1 + tmp2 + 2;
}
```

Program 1

```
int foo2(int X, int Y) {
  int tmp1 = X + 1;

  return tmp1 + Y + 2;
}
```

Program 2

Candidate Generation

```
int foo(int X, int Y) {
  return X + Y;
}
```

Candidate 1

```
int foo(int X) {
  return X + 1;
}
```

Candidate 2

```
int foo(int X) {
  return X + 2;
}
```

Candidate 3

Challenges of Library Learning

- **Challenge 1: Precise candidate generation**

```
int foo1(int X, int Y) {
  int tmp1 = X + 1;
  int tmp2 = Y + 1;

  return tmp1 + tmp2 + 2;
}
```

Program 1

```
int foo2(int X, int Y) {
  int tmp1 = X + 1;

  return tmp1 + Y + 2;
}
```

Program 2

Candidate Generation →

```
int foo(int X, int Y) {
  return X + Y;
}
```

Candidate 1

```
int foo(int X) {
  return X + 1;
}
```

Candidate 2

```
int foo(int X) {
  return X + 2;
}
```

Candidate 3

Challenges of Library Learning

- **Challenge 2: Robust to superficial variation**

```
int foo1(int X, int Y) {
  int tmp1 = X + 1;
  int tmp2 = tmp1 + 1;
  return tmp2;
}
```

Program 1

```
int foo1(int X, int Y) {
  int tmp1 = X + 1;
  int tmp2 = 1 + tmp1;
  return tmp2;
}
```

Program 2

Challenges of Library Learning

- **Challenge 2: Robust to superficial variation**

Library:

```
int add_1(int X) {
  return X + 1;
}
```

Library Learning

```
int foo1(int X, int Y) {
  int tmp1 = X + 1;
  int tmp2 = tmp1 + 1;
  return tmp2;
}
```

Program 1

```
int foo1(int X, int Y) {
  int tmp1 = X + 1;
  int tmp2 = 1 + tmp1;
  return tmp2;
}
```

Program 2

Challenges of Library Learning

- **Challenge 2: Robust to superficial variation**

```
int foo1(int X, int Y) {
  int tmp1 = X + 1;
  int tmp2 = tmp1 + 1;
  return tmp2;
}
```

Program 1

```
int foo1(int X, int Y) {
  int tmp1 = X + 1;
  int tmp2 = 1 + tmp1;
  return tmp2;
}
```

Program 2

Library Learning

Library:

```
int add_1(int X) {
  return X + 1;
}
```

Refactored Programs:

```
int foo1(int X, int Y) {
  return add_1(add_1(X));
}
```

Program 1’

Challenges of Library Learning

- **Challenge 2: Robust to superficial variation**

```
int foo1(int X, int Y) {
  int tmp1 = X + 1;
  int tmp2 = tmp1 + 1;
  return tmp2;
}
```

Program 1

```
int foo1(int X, int Y) {
  int tmp1 = X + 1;
  int tmp2 = 1 + tmp1;
  return tmp2;
}
```

Program 2

Library Learning →

Library:

```
int add_1(int X) {
  return X + 1;
}
```

Refactored Programs:

```
int foo1(int X, int Y) {
  return add_1(add_1(X));
}
```

Program 1’

```
int foo2(int X, int Y) {
  return 1 + add_1(X);
}
```

Program 2’

Solution: Library Learning Modulo Theories (LLMT)

- Precise Candidate Generation via **Anti-Unification**
    - Useful abstractions must be used **at least twice**
    - Abstractions should be **"as concrete as possible"**

Solution: Library Learning Modulo Theories (LLMT)

- Precise Candidate Generation via **Anti-Unification**
  - Useful abstractions must be used **at least twice**
  - Abstractions should be **"as concrete as possible"**

- Robustness via **E-Graphs**
  - use domain-specific equational theory**,** find programs that are **semantically equivalent**

Example: How LLMT works?

- **Program Corpus**

```
int foo1(int X, int Y) {
  int tmp1 = X + 1;
  int tmp2 = tmp1 + 1;
  return tmp2;
}
```

```
int foo2(int X, int Y) {
  int tmp1 = X + 1;
  int tmp2 = 1 + tmp1;
  return tmp2;
}
```

Example: How LLMT works?

- **Step1: Equality Saturation**

```
int foo1(int X, int Y) {
  int tmp1 = X + 1;
  int tmp2 = tmp1 + 1;
  return tmp2;
}
```

```
int foo2(int X, int Y) {
  int tmp1 = X + 1;
  int tmp2 = 1 + tmp1;
  return tmp2;
}
```

$$X + Y \equiv Y + X$$

domain-specific equational theory

Example: How LLMT works?

- **Step1: Equality Saturation**

```
int foo1(int X, int Y) {
  int tmp1 = X + 1;
  int tmp2 = tmp1 + 1;
  return tmp2;
}
```

```
int foo2(int X, int Y) {
  int tmp1 = X + 1;
  int tmp2 = 1 + tmp1;
  return tmp2;
}
```

$$X + Y \equiv Y + X$$

domain-specific equational theory

```
int foo3(int X, int Y) {
  int tmp1 = X + 1;
  int tmp2 = tmp1 + 1;
  return tmp2;
}
```

Example: How LLMT works?

- **Step2: Anti-Unification (Candidate Generation)**

```
int foo1(int X, int Y) {
  int tmp1 = X + 1;
  int tmp2 = tmp1 + 1;
  return tmp2;
}
```

```
int foo2(int X, int Y) {
  int tmp1 = X + 1;
  int tmp2 = 1 + tmp1;
  return tmp2;
}
```

```
int foo3(int X, int Y) {
  int tmp1 = X + 1;
  int tmp2 = tmp1 + 1;
  return tmp2;
}
```

$$\lambda x \to x + 1$$

- matched to corpus at least **twice**

$$\lambda x\ y \to x + y$$

Candidates

Example: How LLMT works?

- **Step3: Select Optimal Library**

```
int foo1(int X, int Y) {
  int tmp1 = X + 1;
  int tmp2 = tmp1 + 1;
  return tmp2;
}
```

```
int foo2(int X, int Y) {
  int tmp1 = X + 1;
  int tmp2 = 1 + tmp1;
  return tmp2;
}
```

```
int foo3(int X, int Y) {
  int tmp1 = X + 1;
  int tmp2 = tmp1 + 1;
  return tmp2;
}
```

$\lambda x \rightarrow x + 1$

(Cost: 1, requires one arguments)

(Cost: 2, requires two arguments)

$\lambda x\ y \rightarrow x + y$

Candidates

Example: How LLMT works?

- **Step3: Select Optimal Library**

```
int foo1(int X, int Y) {
  int tmp1 = X + 1;
  int tmp2 = tmp1 + 1;
  return tmp2;
}
```

```
int foo2(int X, int Y) {
  int tmp1 = X + 1;
  int tmp2 = 1 + tmp1;
  return tmp2;
}
```

```
int foo3(int X, int Y) {
  int tmp1 = X + 1;
  int tmp2 = tmp1 + 1;
  return tmp2;
}
```

$\lambda x \rightarrow x + 1$

(Cost: 1, requires one arguments)

(Cost: 2, requires two arguments)

$\lambda x\ y \rightarrow x + y$

Candidates

Library:

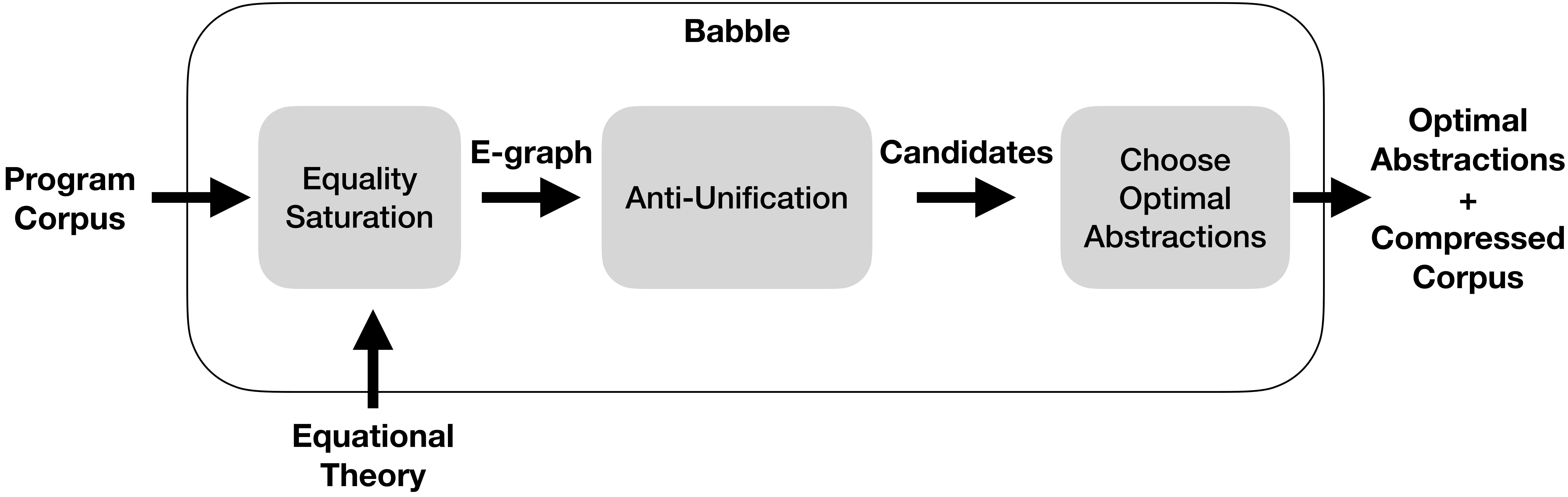
```
int add_1(int X) {
  return X + 1;
}
```

Compressed Corpus:

```
int foo1(int X, int Y) {
  return add_1(add_1(X));
}
```

```
int foo3(int X, int Y) {
  return add_1(add_1(X));
}
```

Babble: Overall Process



Appealing Result

Low Cost

10x faster

Performance

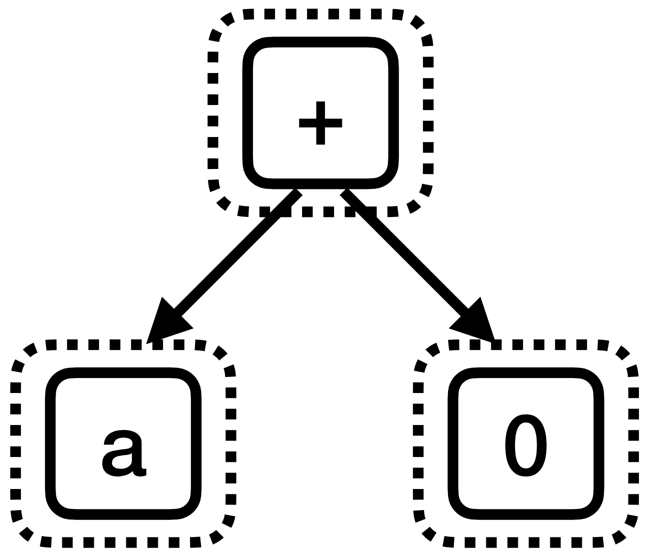
30% better
Compression Ratio

Scalability

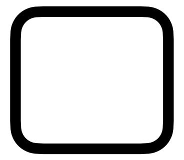
Can support 42,936 AST size input

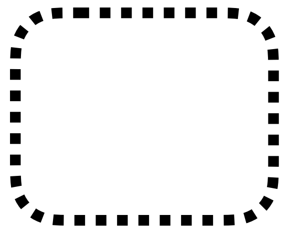
Technical Details: How E-graphs works?

- **E-graphs**



a + 0 expressed in e-graph

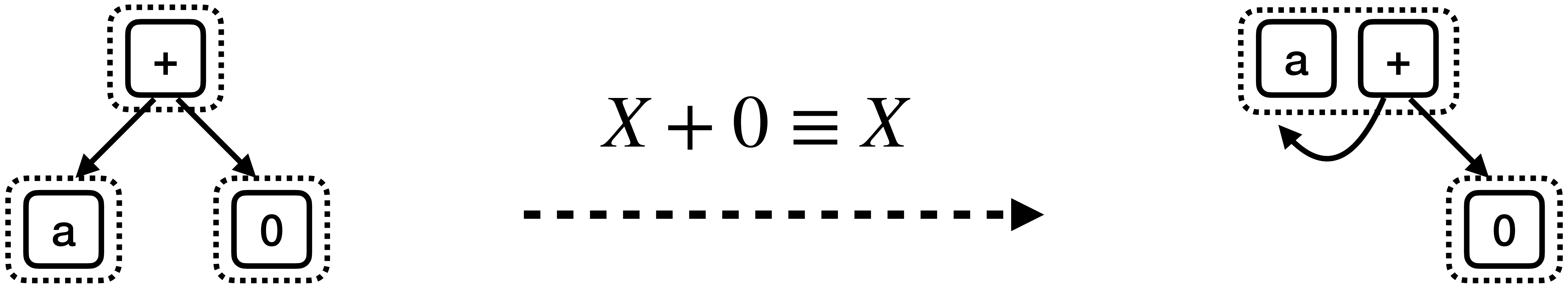
 e-node: Root node of subexpression tree

 e-class: Set of e-nodes, express equivalence relation between e-nodes

 edge: e-node to e-class, express e-class is subtree of e-node

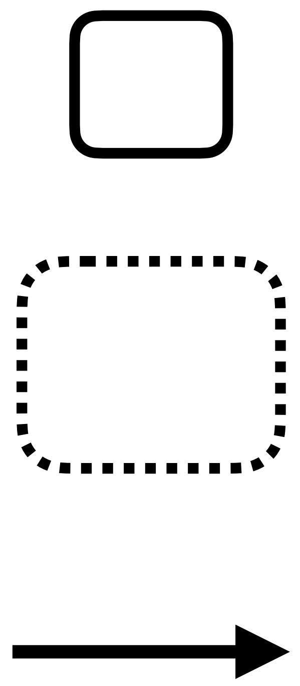
Technical Details: How E-graphs works?

- **E-graphs**



a + 0 expressed in e-graph

…
= a + 0 + 0
= a + 0
a



e-node: Root node of subexpression tree

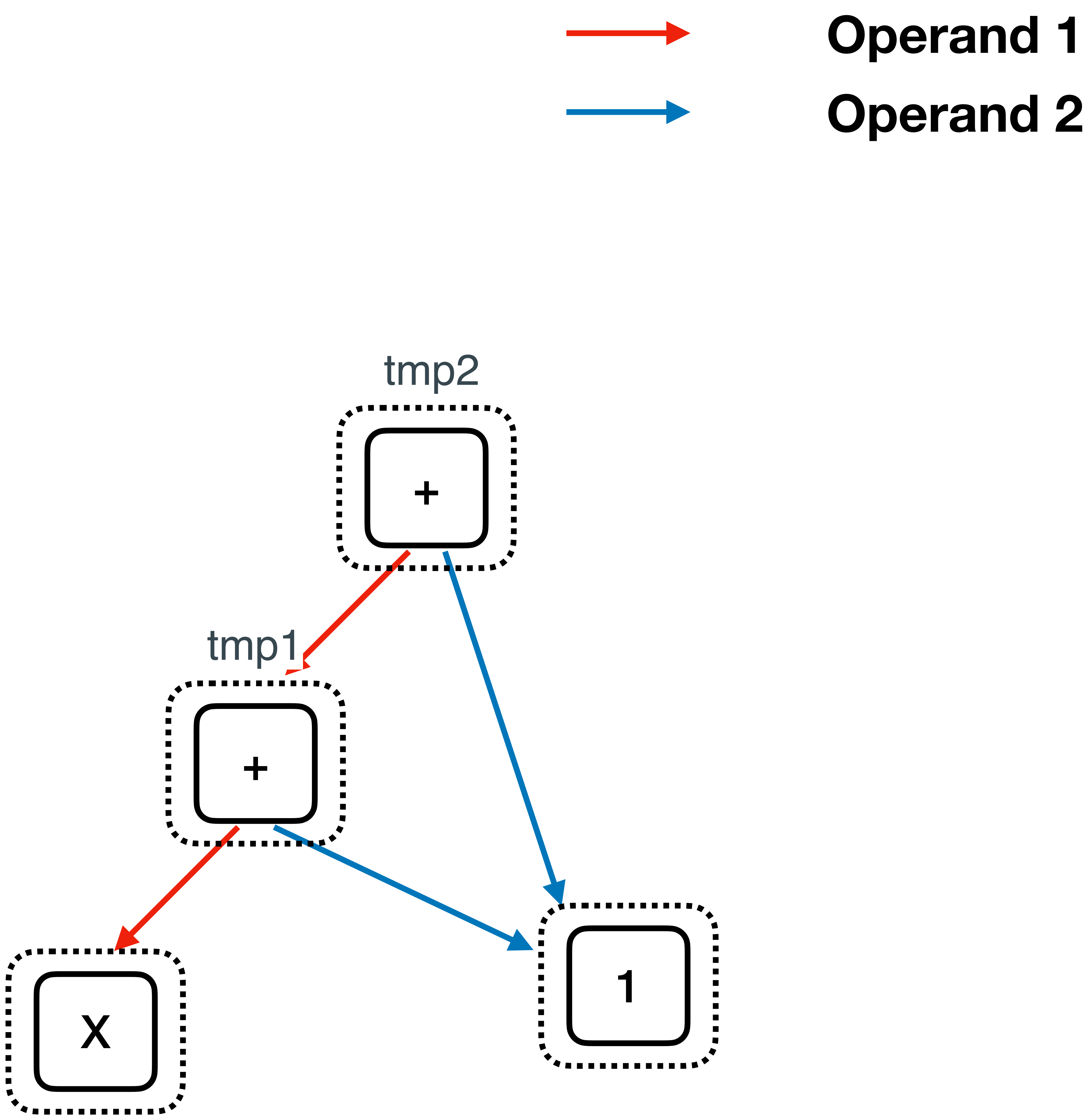
e-class: Set of e-nodes, express equivalence relation between e-nodes

edge: e-node to e-class, express e-class is subtree of e-node

E-graph in Babble

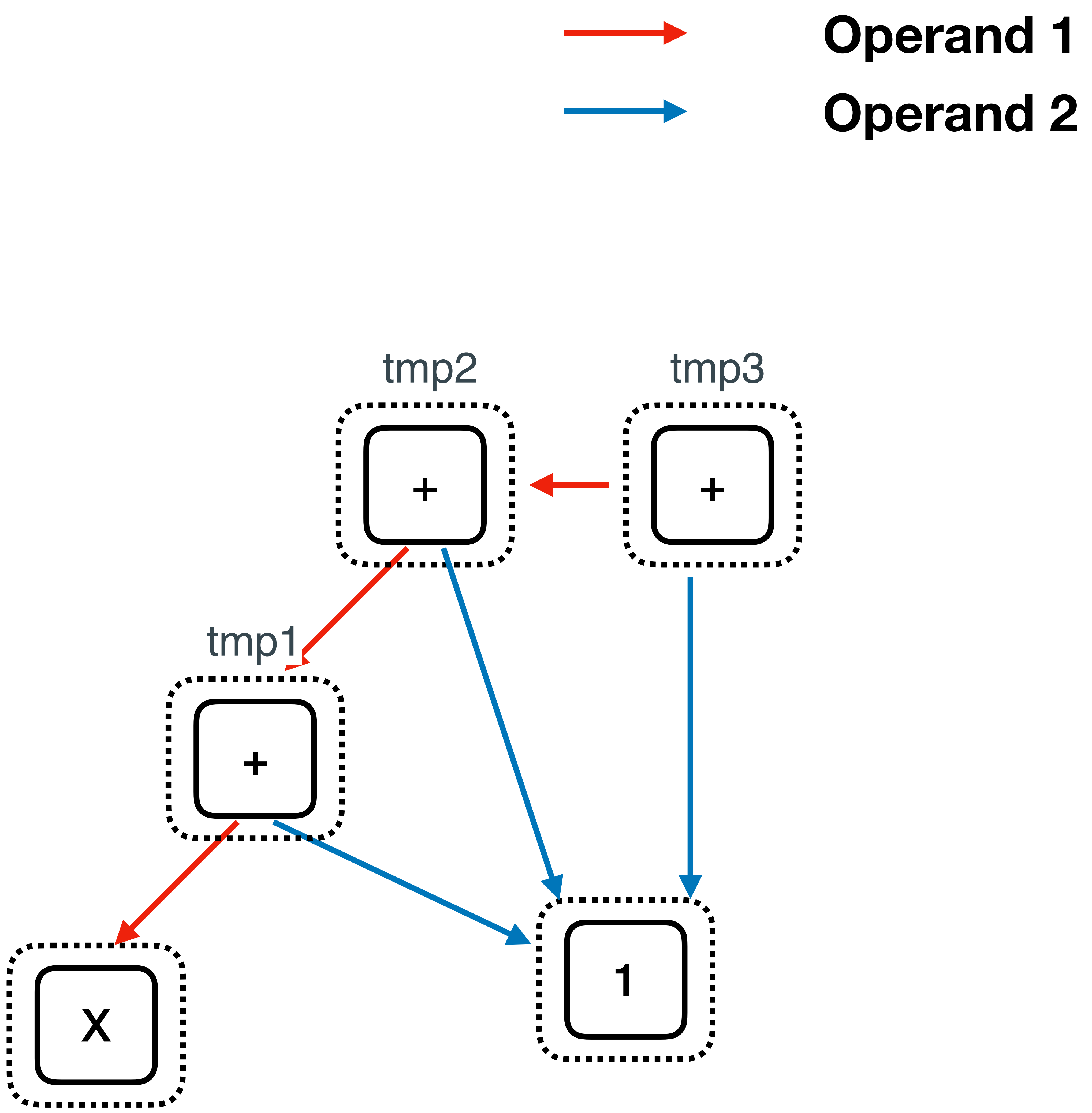
- **Build E-Graph**

```
int foo1(int X, int Y) {
  int tmp1 = X + 1;
  int tmp2 = tmp1 + 1;
  return tmp2;
}
```



E-graph in Babble

- **Build E-Graph**




```
int foo1(int X, int Y) {
  int tmp1 = X + 1;
  int tmp2 = tmp1 + 1;
  return tmp2;
}
```

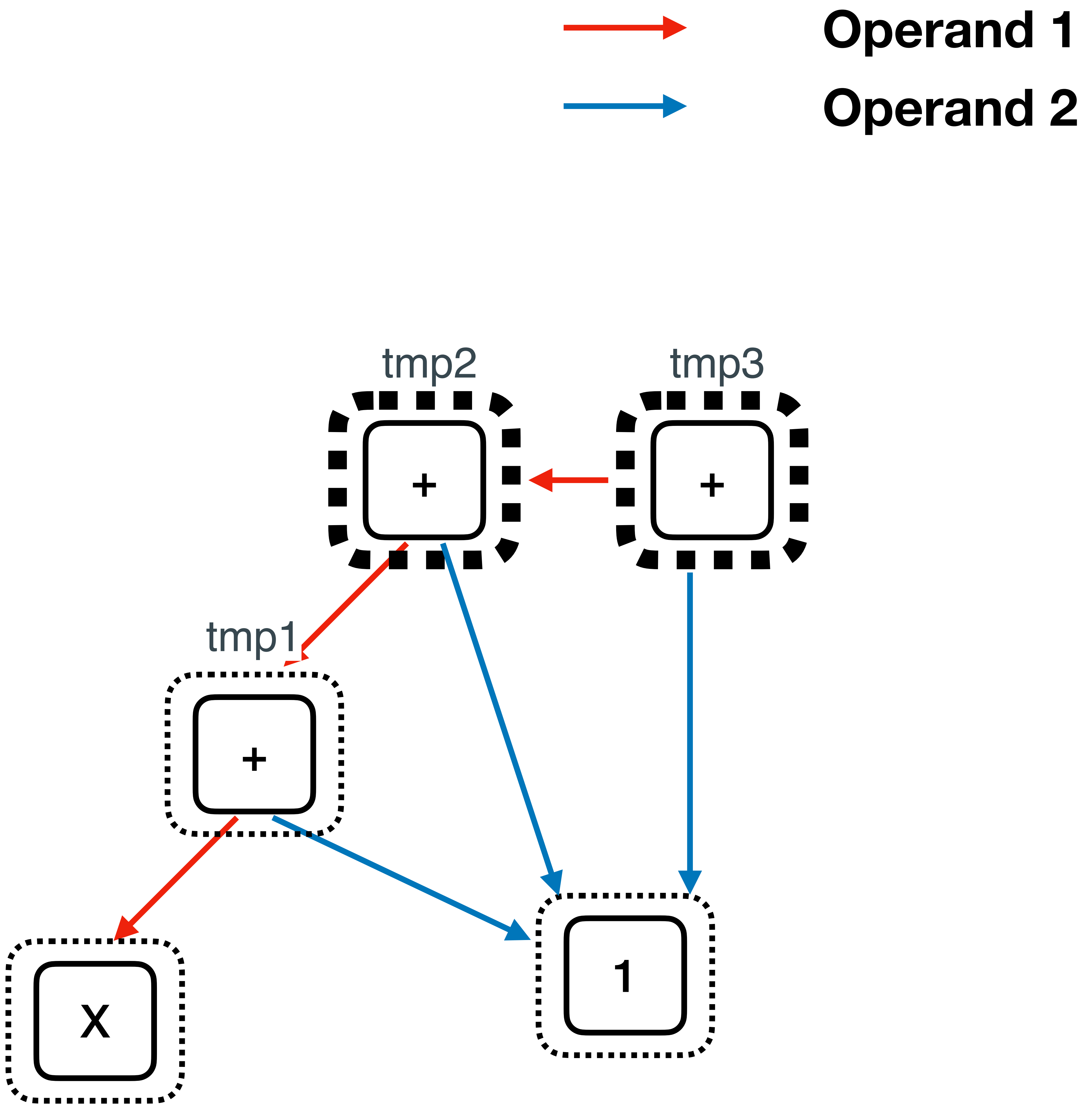
```
int foo2(int X, int Y) {
  int tmp1 = X + 1;
  int tmp2 = tmp1 + 1;
  int tmp3 = tmp2 + 1;
  return tmp3;
}
```

E-graph in Babble

- Generate Candidates
  - Pick two e-classes

```
int foo1(int X, int Y) {
  int tmp1 = X + 1;
  int tmp2 = tmp1 + 1;
  return tmp2;
}
```

```
int foo2(int X, int Y) {
  int tmp1 = X + 1;
  int tmp2 = tmp1 + 1;
  int tmp3 = tmp2 + 1;
  return tmp3;
}
```

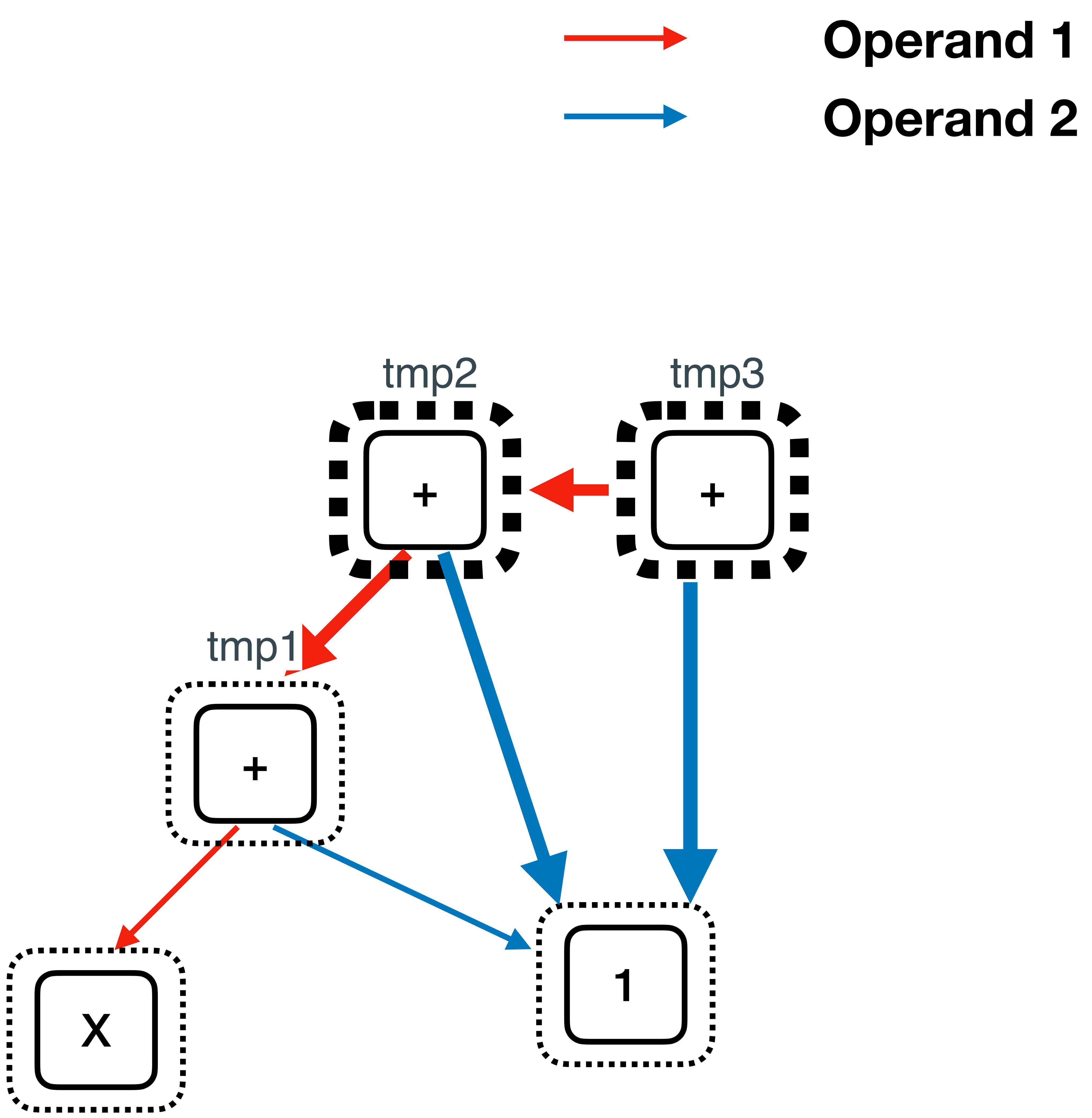


E-graph in Babble

- Generate Candidates
  - Pick two e-classes
  - matching

```
int foo1(int X, int Y) {
  int tmp1 = X + 1;
  int tmp2 = tmp1 + 1;
  return tmp2;
}
```

```
int foo2(int X, int Y) {
  int tmp1 = X + 1;
  int tmp2 = tmp1 + 1;
  int tmp3 = tmp2 + 1;
  return tmp3;
}
```



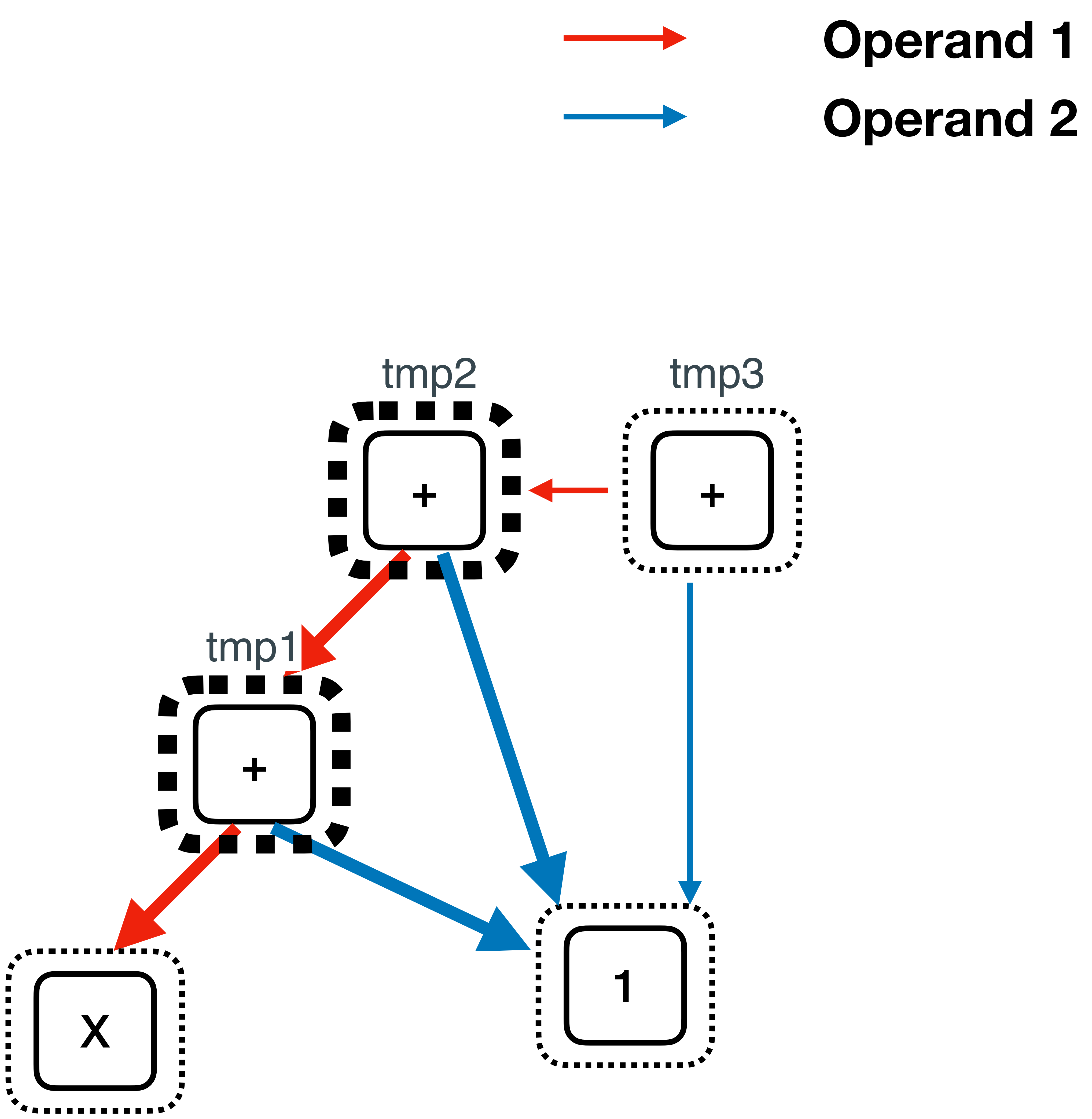
?? + 1 is common

E-graph in Babble

- **Generate Candidates**
  - **Pick two e-classes**
  - **matching**
  - **repeat**

```
int foo1(int X, int Y) {
  int tmp1 = X + 1;
  int tmp2 = tmp1 + 1;
  return tmp2;
}
```

```
int foo2(int X, int Y) {
  int tmp1 = X + 1;
  int tmp2 = tmp1 + 1;
  int tmp3 = tmp2 + 1;
  return tmp3;
}
```



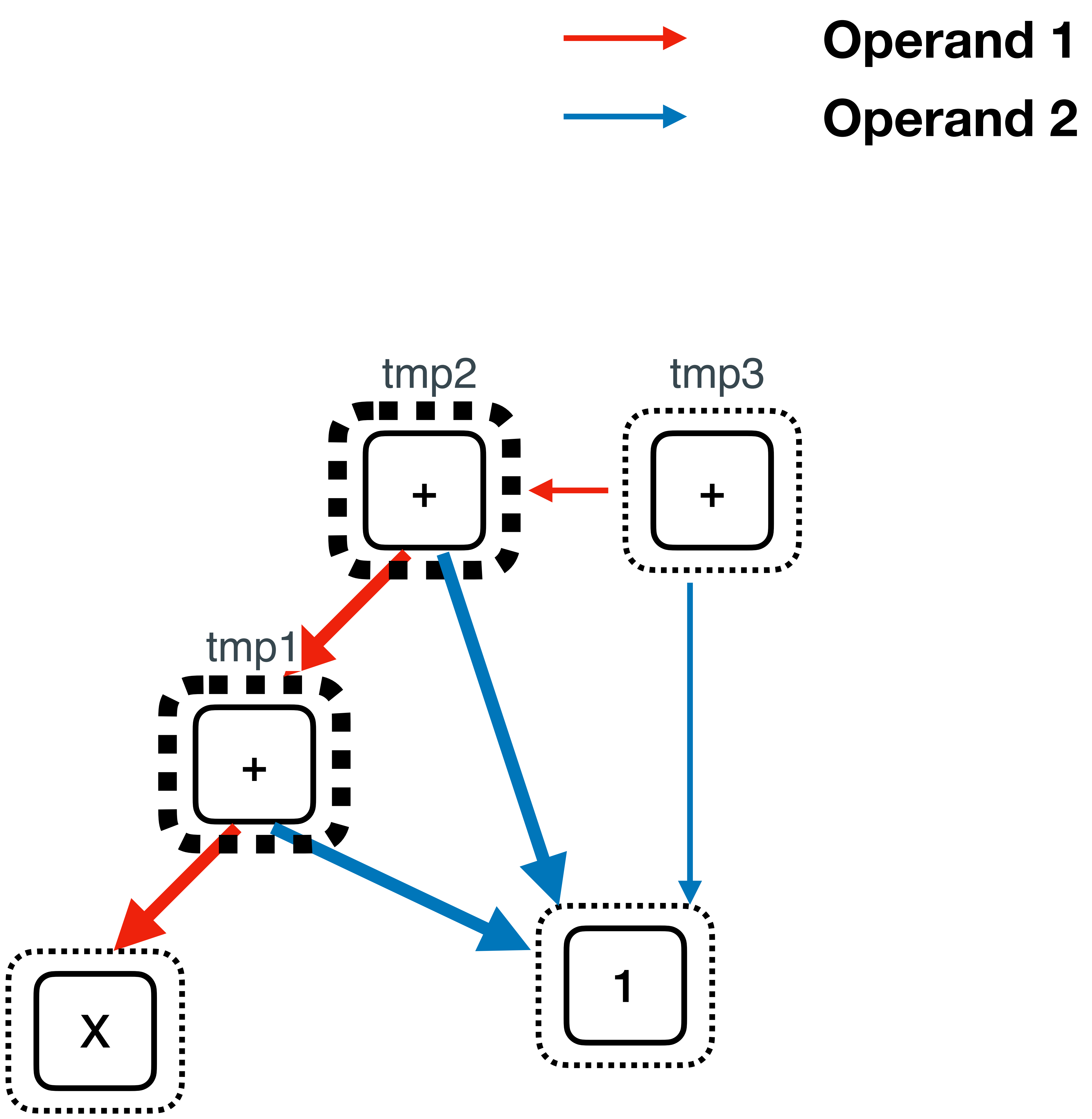
?? + 1 is common

E-graph in Babble

- **Generate Candidates**
  - **Pick two e-classes**
  - **matching**
  - **repeat**

```
int foo1(int X, int Y) {
  int tmp1 = X + 1;
  int tmp2 = tmp1 + 1;
  return tmp2;
}
```

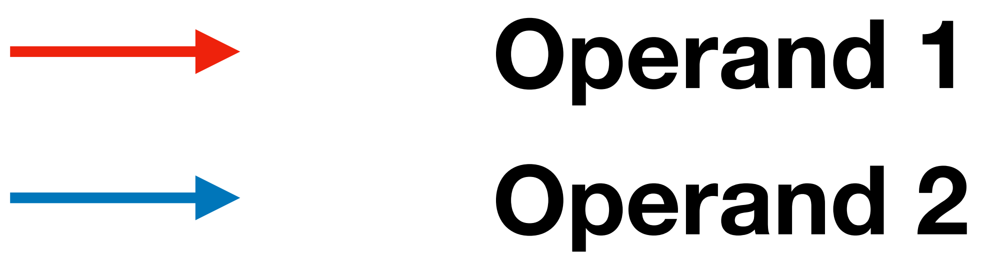
```
int foo2(int X, int Y) {
  int tmp1 = X + 1;
  int tmp2 = tmp1 + 1;
  int tmp3 = tmp2 + 1;
  return tmp3;
}
```



?? + 1 is common

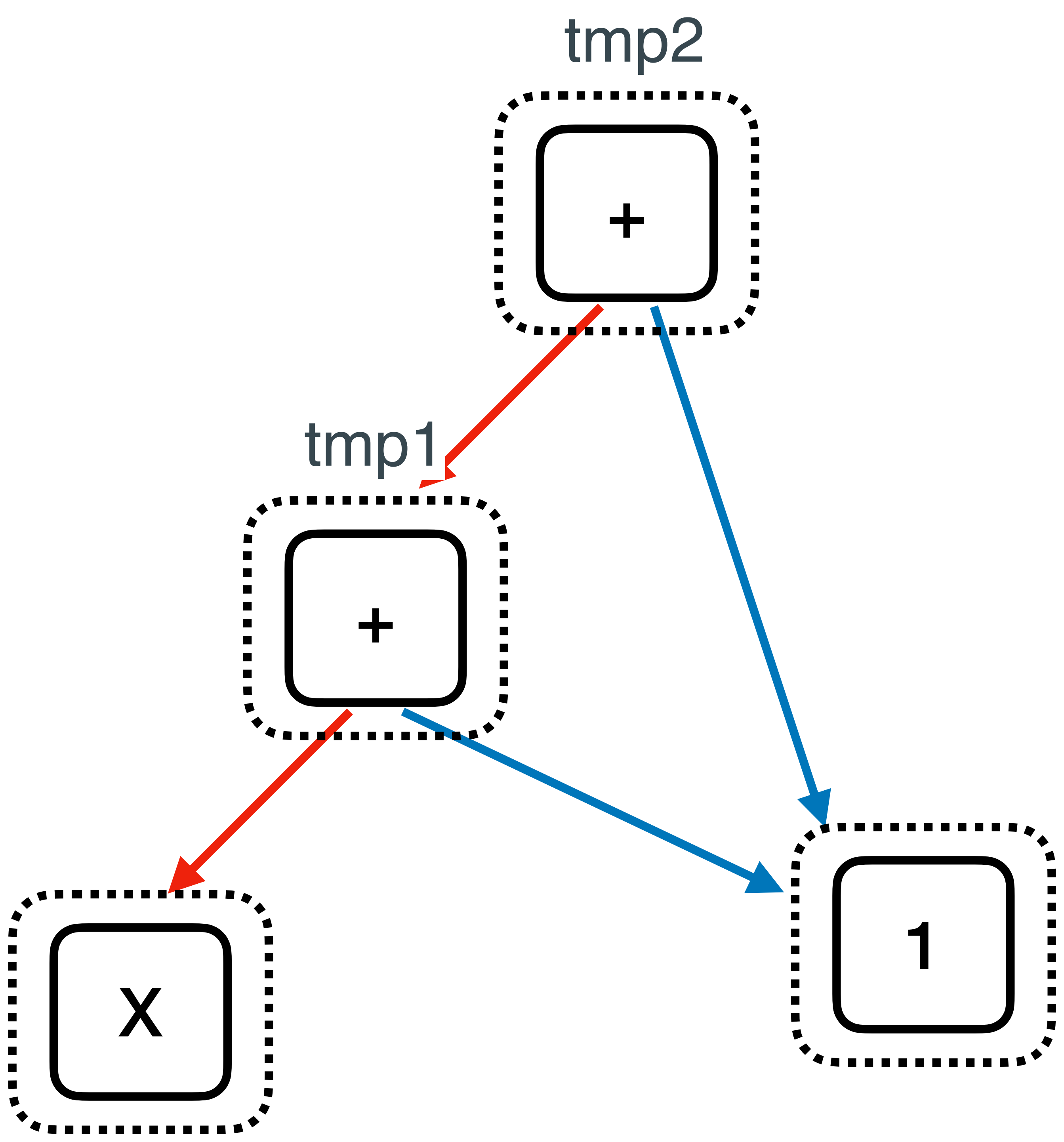
$$\lambda x \rightarrow x + 1$$

E-graph in Babble



- **Build E-Graph**

```
int foo1(int X, int Y) {
  int tmp1 = X + 1;
  int tmp2 = tmp1 + 1;
  return tmp2;
}
```



E-graph in Babble

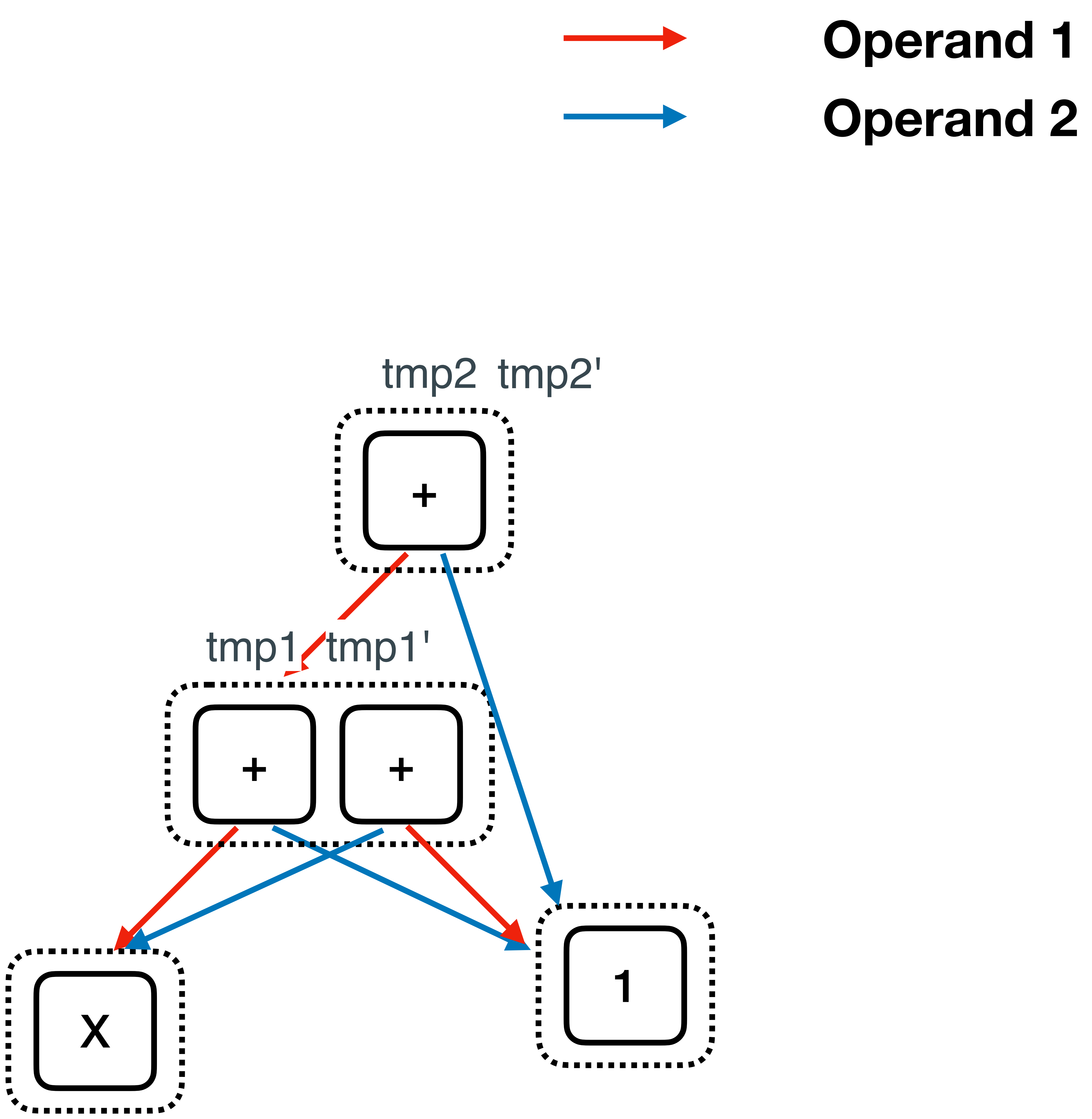
- **Build E-Graph**

$$X + Y \equiv Y + X$$

domain-specific equational theory

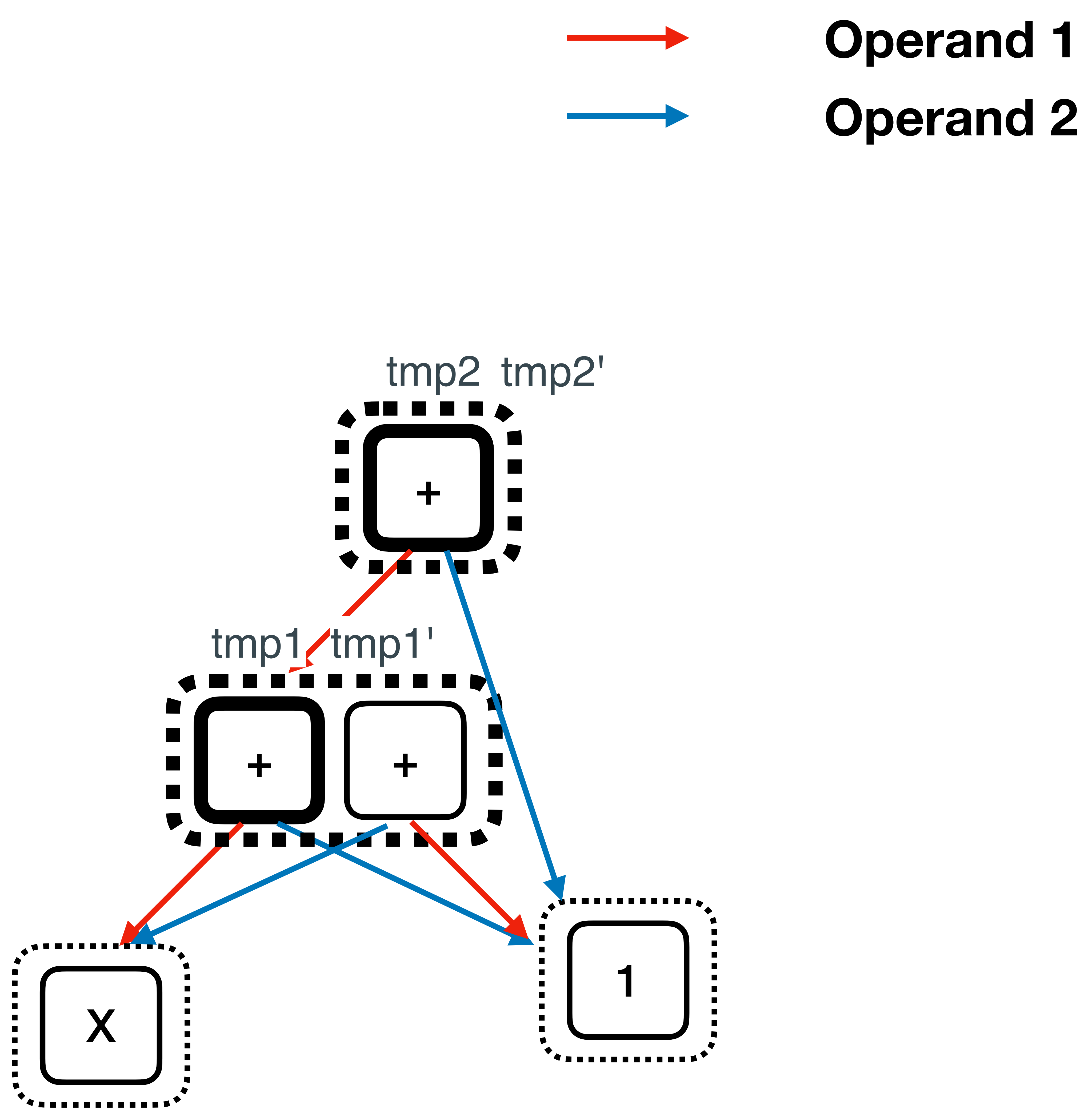
```
int foo1(int X, int Y) {
  int tmp1 = X + 1;
  int tmp2 = tmp1 + 1;
  return tmp2;
}
```

```
int foo2(int X, int Y) {
  int tmp1' = 1 + X;
  int tmp2' = tmp1' + 1;
  return tmp2';
}
```



E-graph in Babble

- **Generate Candidate**
  - **Pick two e-classes**




```
int foo1(int X, int Y) {
  int tmp1 = X + 1;
  int tmp2 = tmp1 + 1;
  return tmp2;
}
```

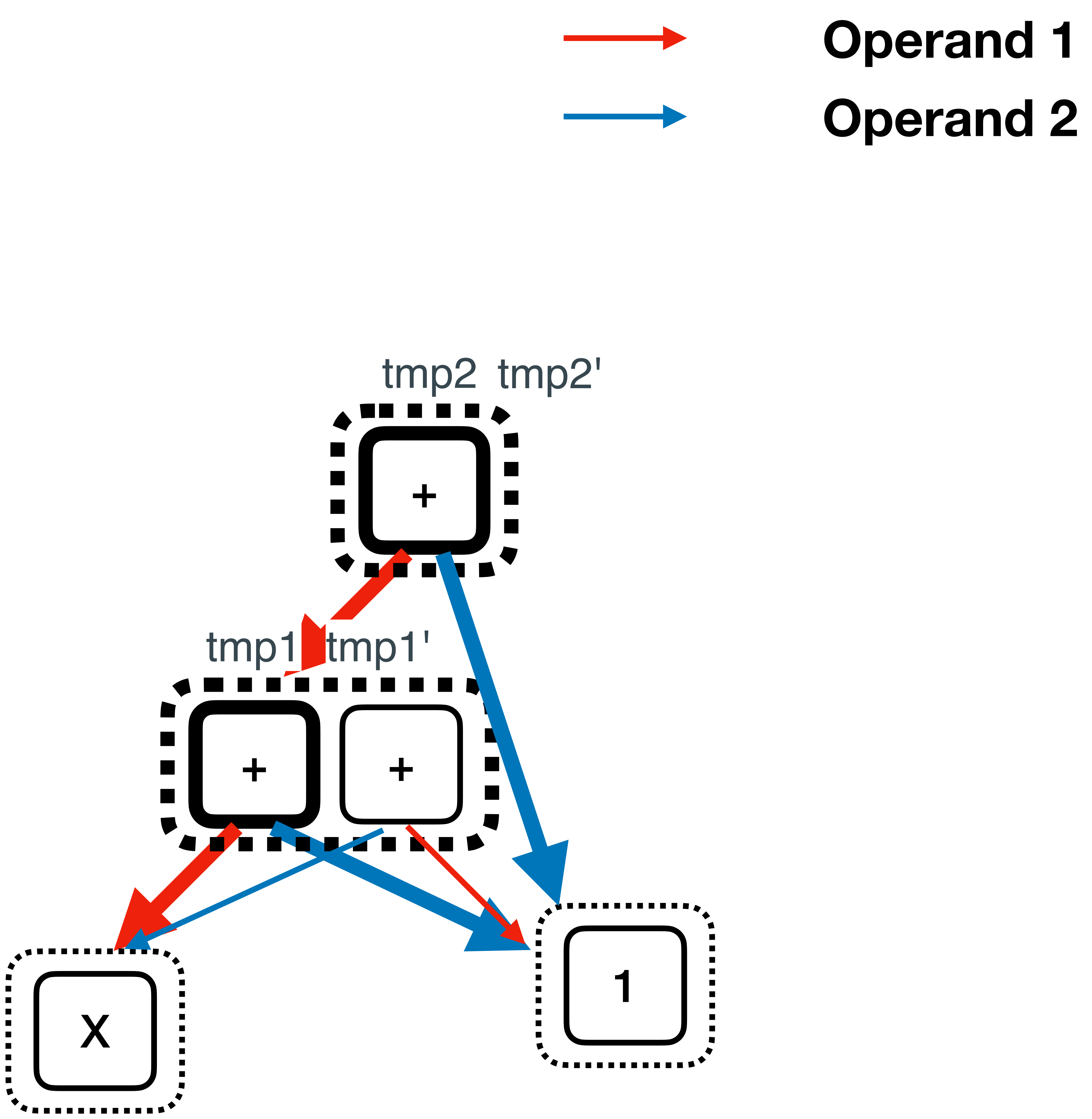
```
int foo2(int X, int Y) {
  int tmp1' = 1 + X;
  int tmp2' = tmp1' + 1;
  return tmp2';
}
```

E-graph in Babble

- **Generate Candidate**
  - **Pick two e-classes**
  - **matching**

```
int foo1(int X, int Y) {
  int tmp1 = X + 1;
  int tmp2 = tmp1 + 1;
  return tmp2;
}
```

```
int foo2(int X, int Y) {
  int tmp1' = 1 + X;
  int tmp2' = tmp1' + 1;
  return tmp2';
}
```

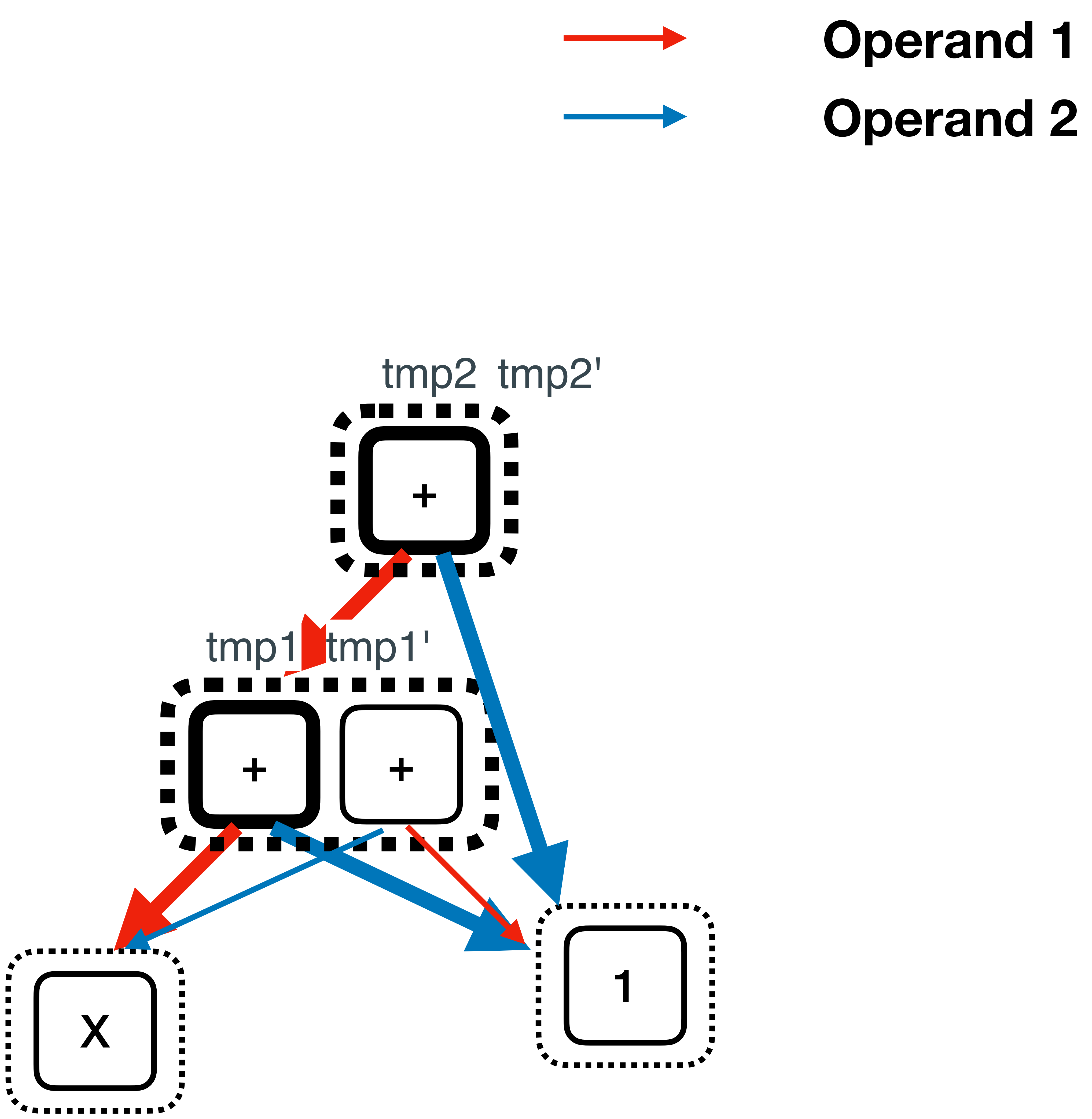


E-graph in Babble

- Generate Candidate
  - Pick two e-classes
  - matching

```
int foo1(int X, int Y) {
  int tmp1 = X + 1;
  int tmp2 = tmp1 + 1;
  return tmp2;
}
```

```
int foo2(int X, int Y) {
  int tmp1' = 1 + X;
  int tmp2' = tmp1' + 1;
  return tmp2';
}
```



?? + 1 is common

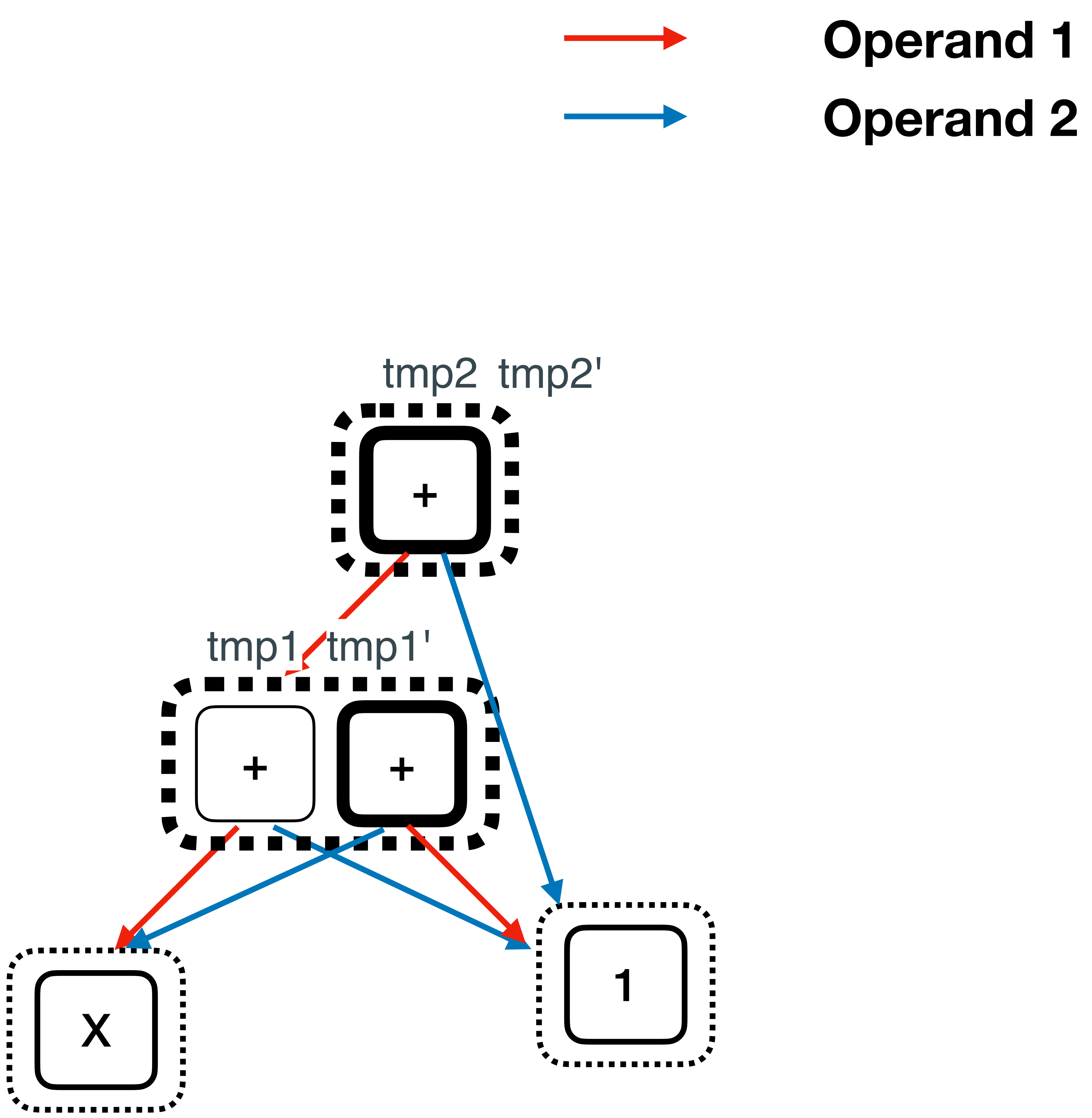
$$\lambda x \rightarrow x + 1$$

E-graph in Babble

- **Generate Candidate**
  - **Pick two e-classes**
  - **matching**
  - **repeat**

```
int foo1(int X, int Y) {
  int tmp1 = X + 1;
  int tmp2 = tmp1 + 1;
  return tmp2;
}
```

```
int foo2(int X, int Y) {
  int tmp1' = 1 + X;
  int tmp2' = tmp1' + 1;
  return tmp2';
}
```

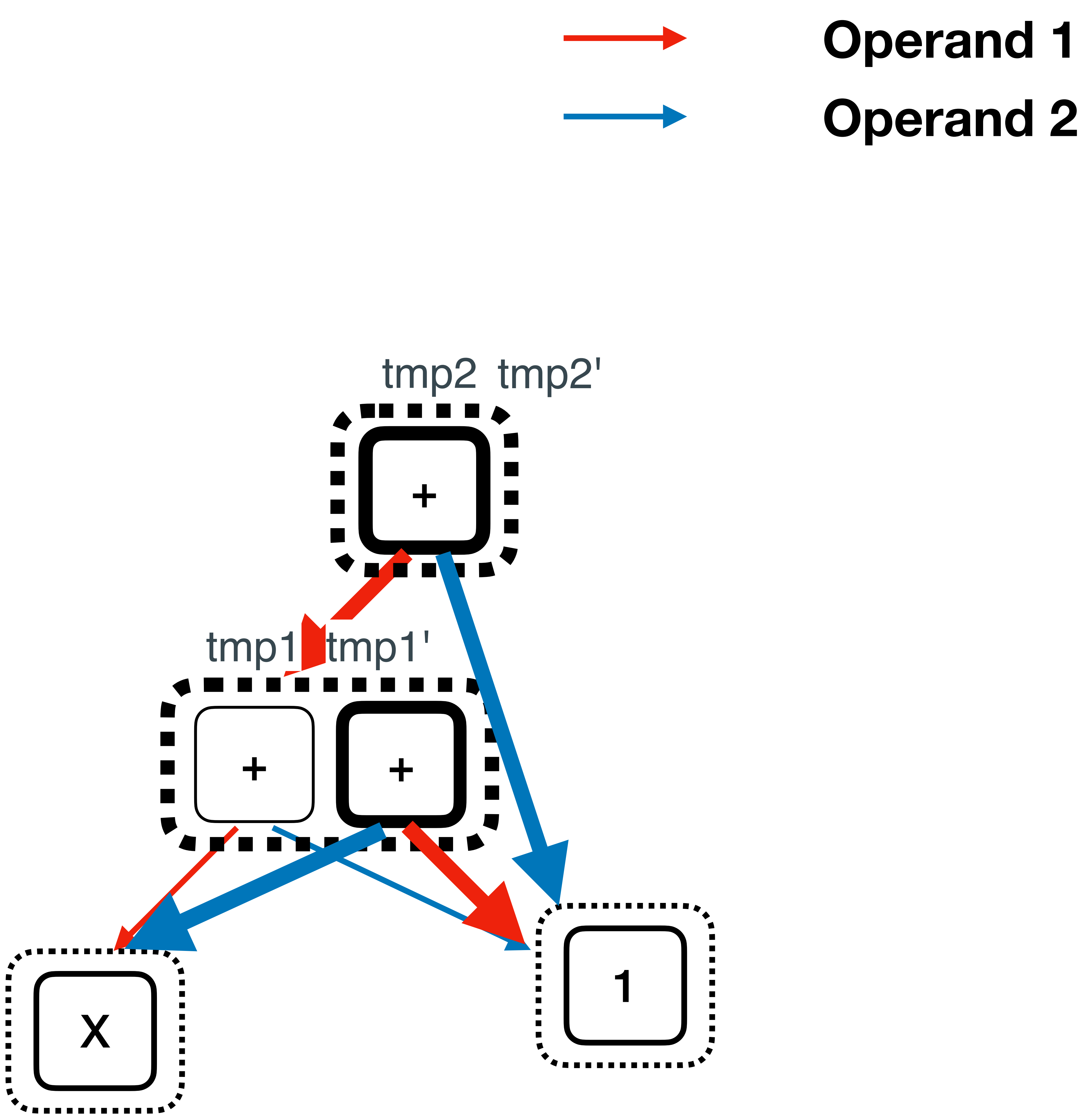


E-graph in Babble

- Generate Candidate
  - Pick two e-classes
  - matching
  - repeat

```
int foo1(int X, int Y) {
  int tmp1 = X + 1;
  int tmp2 = tmp1 + 1;
  return tmp2;
}
```

```
int foo2(int X, int Y) {
  int tmp1' = 1 + X;
  int tmp2' = tmp1' + 1;
  return tmp2';
}
```



?? + ?? is common

$$\lambda x\ y \rightarrow x + y$$

Operand 1

Operand 2

E-graph in Babble

- **Choose Optimal Abstraction**

2 Candidates

```
int foo1(int X, int Y) {
  int tmp1 = X + 1;
  int tmp2 = tmp1 + 1;
  return tmp2;
}
```

```
int foo2(int X, int Y) {
  int tmp1' = 1 + X;
  int tmp2' = tmp1' + 1;
  return tmp2';
}
```

$$\lambda x \to x + 1$$

$$\lambda x\ y \to x + y$$

Operand 1

Operand 2

E-graph in Babble

- **Choose Optimal Abstraction**

2 Candidates

```
int foo1(int X, int Y) {
  int tmp1 = X + 1;
  int tmp2 = tmp1 + 1;
  return tmp2;
}
```

```
int foo2(int X, int Y) {
  int tmp1' = 1 + X;
  int tmp2' = tmp1' + 1;
  return tmp2';
}
```

✓ $\lambda x \to x + 1$

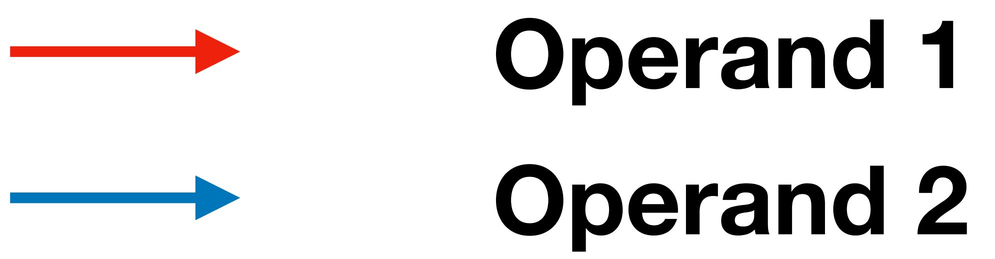
(Cost: 1, requires one arguments)

$\lambda x\ y \to x + y$

(Cost: 2, requires two arguments)

E-graph in Babble

- **Compress**



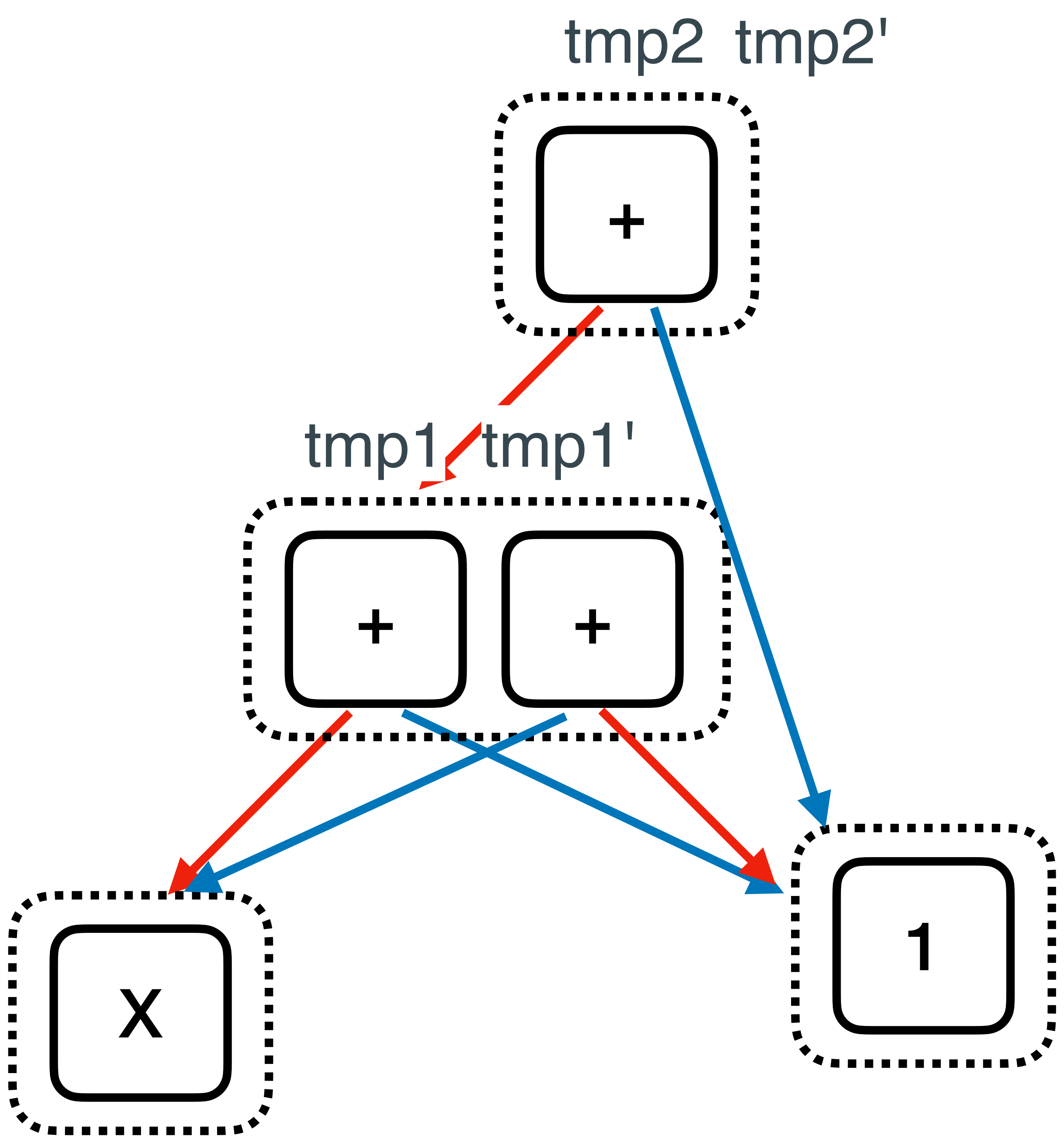
$$\lambda x \to x + 1$$

Library:

```
int add_1(int X) {
  return X + 1;
}
```

```
int foo1(int X, int Y) {
  int tmp1 = X + 1;
  int tmp2 = tmp1 + 1;
  return tmp2;
}
```

```
int foo2(int X, int Y) {
  int tmp1' = 1 + X;
  int tmp2' = tmp1' + 1;
  return tmp2';
}
```



E-graph in Babble

- **Compress**

Operand 1

Operand 2

$$\lambda x \to x + 1$$

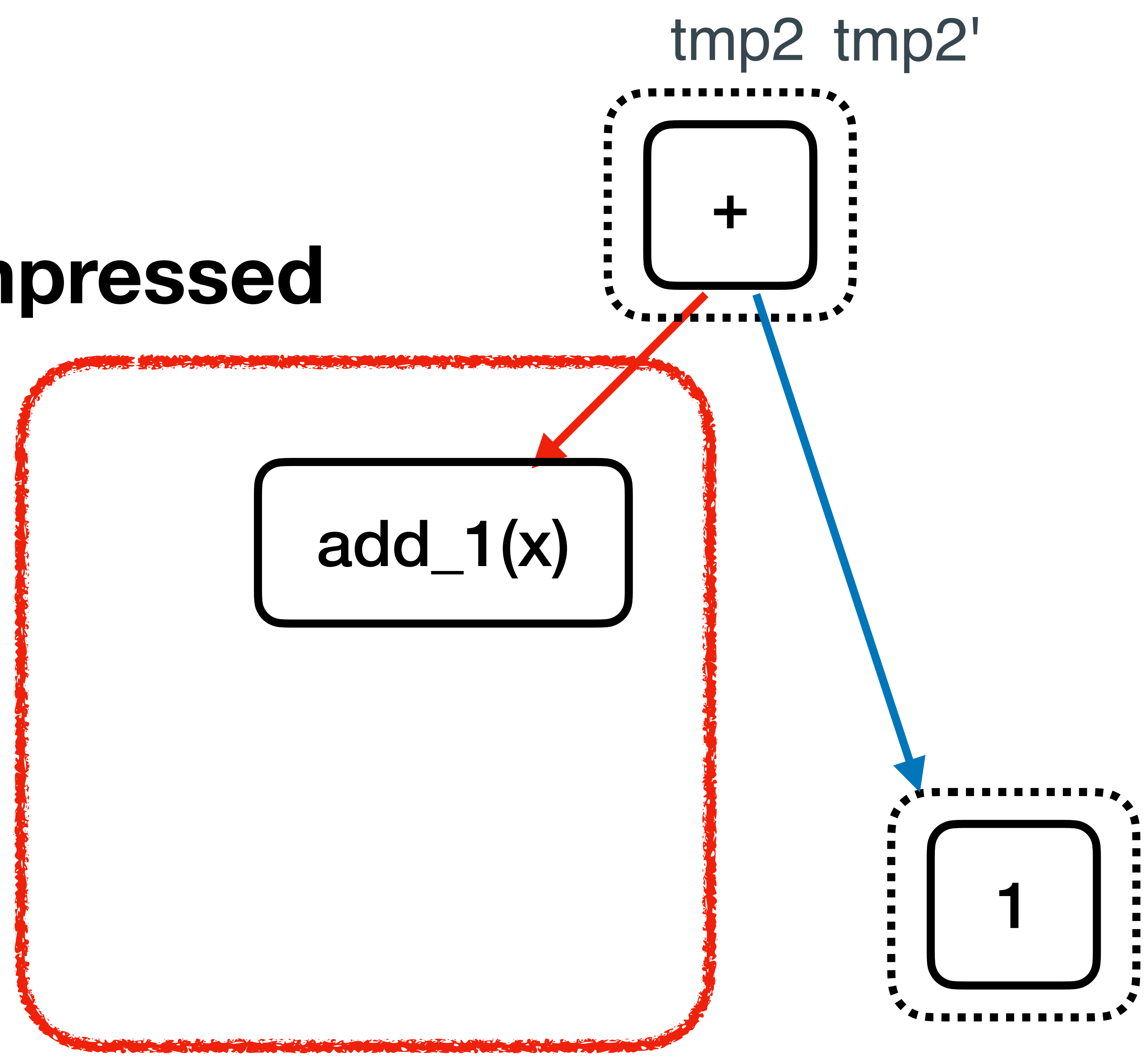
Library:

```
int add_1(int X) {
  return X + 1;
}
```

```
int foo1(int X, int Y) {
  int tmp1 = X + 1;
  int tmp2 = tmp1 + 1;
  return tmp2;
}
```

```
int foo2(int X, int Y) {
  int tmp1' = 1 + X;
  int tmp2' = tmp1' + 1;
  return tmp2';
}
```

Compressed



Operand 1

Operand 2

E-graph in Babble

- **Compress**

$$\lambda x \to x + 1$$

Library:

```
int add_1(int X) {
  return X + 1;
}
```

Compressed

```
add_1(add_1(x))
```

```
int foo1(int X, int Y) {
  int tmp1 = X + 1;
  int tmp2 = tmp1 + 1;
  return tmp2;
}
```

```
int foo2(int X, int Y) {
  int tmp1' = 1 + X;
  int tmp2' = tmp1' + 1;
  return tmp2';
}
```

E-graph in Babble

- **Compress**

Operand 1

Operand 2

$$\lambda x \to x + 1$$

Library:

```
int add_1(int X) {
  return X + 1;
}
```

```
int foo1(int X, int Y) {
  int tmp1 = X + 1;
  int tmp2 = tmp1 + 1;
  return tmp2;
}
```

```
int foo2(int X, int Y) {
  int tmp1' = 1 + X;
  int tmp2' = tmp1' + 1;
  return tmp2';
}
```

Compressed result

Compressed Corpus:

```
int foo1(int X, int Y) {
  return add_1(add_1(X));
}
```

```
int foo2(int X, int Y) {
  return add_1(add_1(X));
}
```

Evaluation: Set-up

- **Baseline: DreamCoder (SOTA)**
- **RQs**
  - **RQ1**: Can Babble compress programs better than SOTA?
  - **RQ2**: Main technique is important to the performance?

Evaluation: Comparison with SOTA

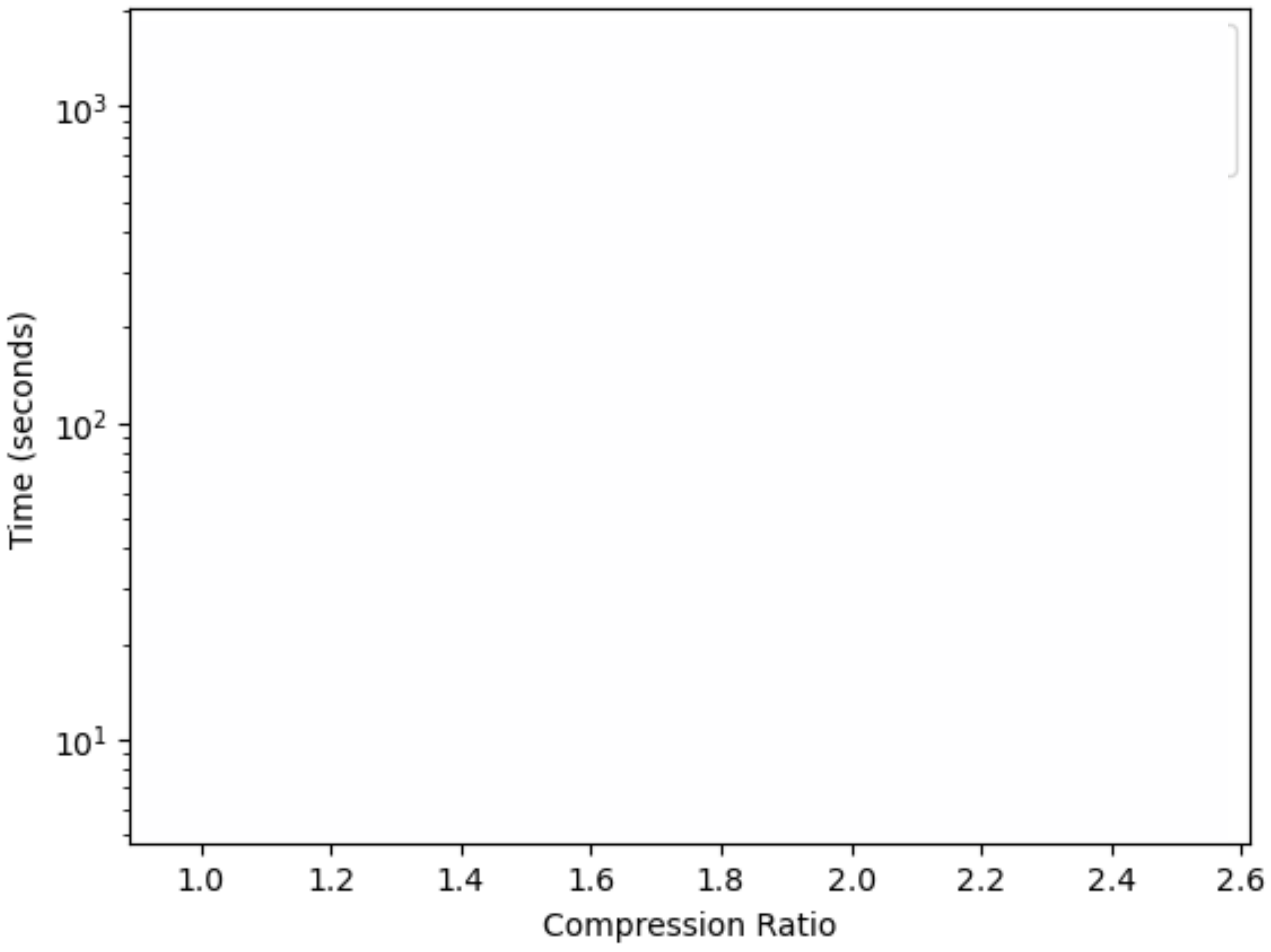
- **Baseline: DreamCoder (SOTA), BabbleSyn (Babble without Equality Thoery)**
- **Benchmarks: List domain**

Sum List
[1 2 3] -> 6
[4 6 8 1] -> 17

Double
[1 2 3] -> [2 4 6]
[4 5 1] -> [8 10 2]

Check Evens
[0 2 3] -> [T T F]
[2 9 6] -> [T F T]
...

does it scale?



does it compress?

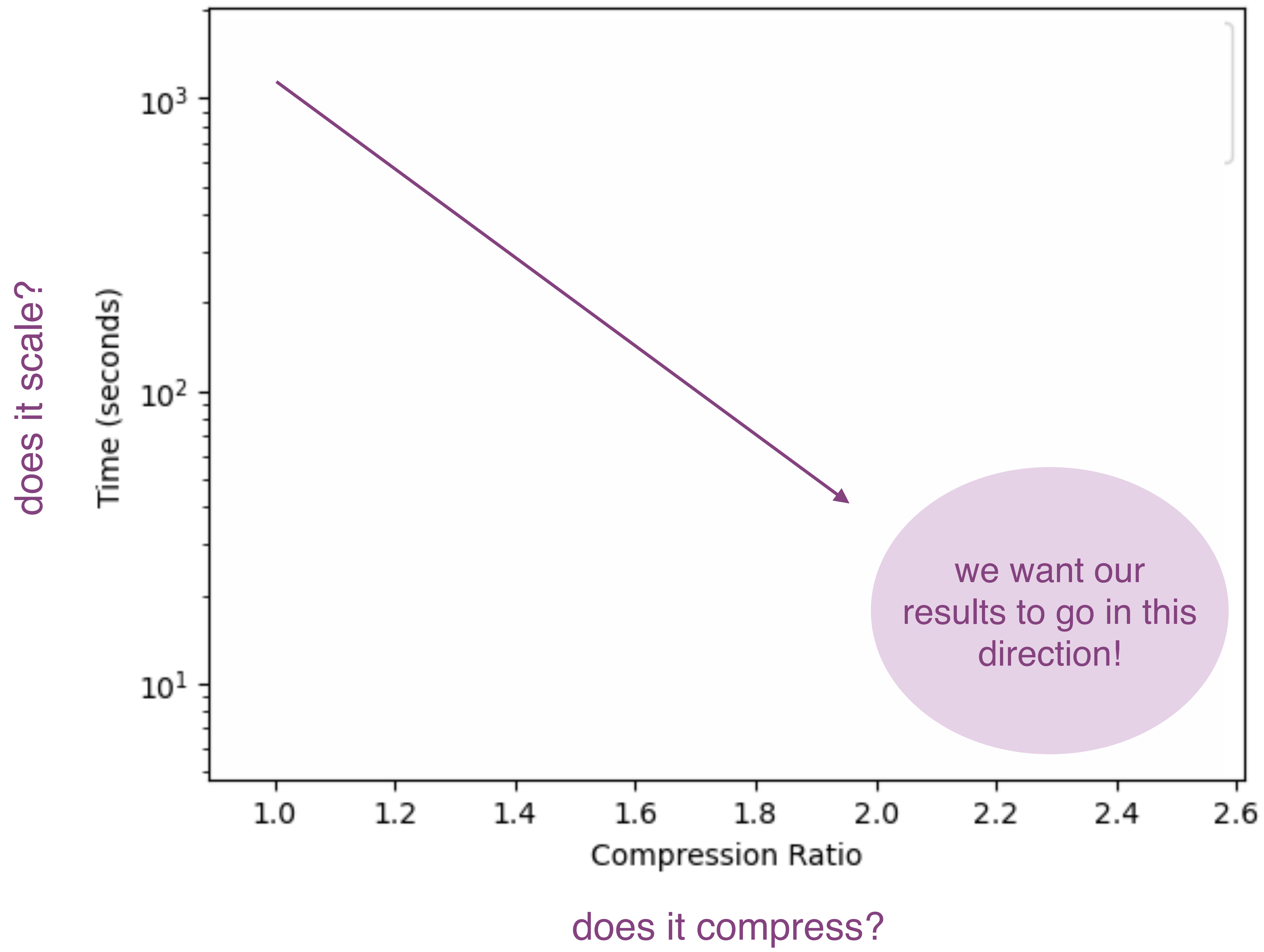
Evaluation: Comparison with SOTA

- **Baseline: DreamCoder (SOTA), BabbleSyn (Babble without Equality Thoery)**
- **Benchmarks: List domain**

Sum List
[1 2 3] -> 6
[4 6 8 1] -> 17

Double
[1 2 3] -> [2 4 6]
[4 5 1] -> [8 10 2]

Check Evens
[0 2 3] -> [T T F]
[2 9 6] -> [T F T]
...



Evaluation: Comparison with SOTA

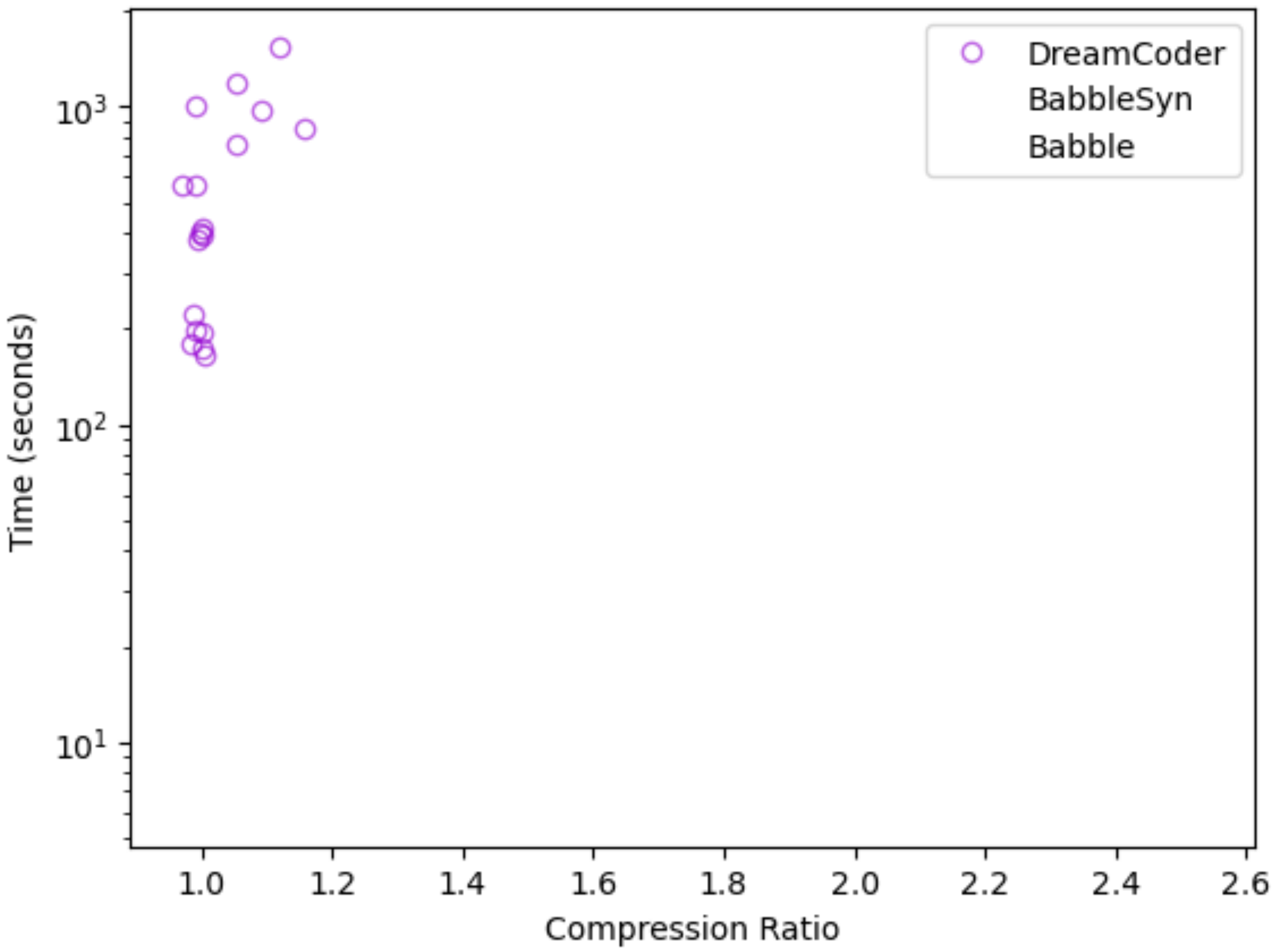
- **Baseline: DreamCoder (SOTA), BabbleSyn (Babble without Equality Thoery)**
- **Benchmarks: List domain**

Sum List
[1 2 3] -> 6
[4 6 8 1] -> 17

Double
[1 2 3] -> [2 4 6]
[4 5 1] -> [8 10 2]

Check Evens
[0 2 3] -> [T T F]
[2 9 6] -> [T F T]
...

does it scale?



does it compress?

Evaluation: Comparison with SOTA

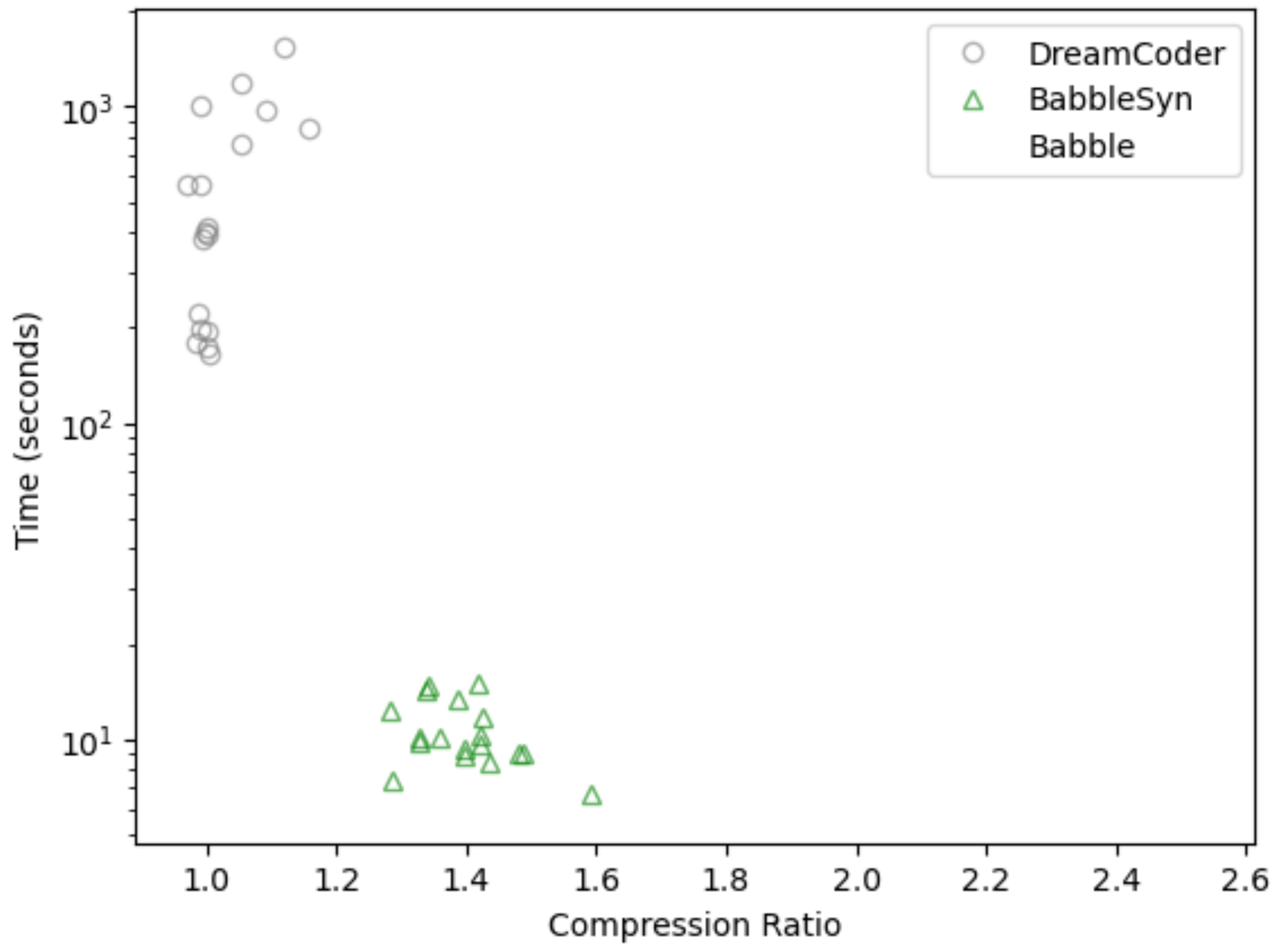
- **Baseline: DreamCoder (SOTA), BabbleSyn (Babble without Equality Thoery)**
- **Benchmarks: List domain**

Sum List
[1 2 3] -> 6
[4 6 8 1] -> 17

Double
[1 2 3] -> [2 4 6]
[4 5 1] -> [8 10 2]

Check Evens
[0 2 3] -> [T T F]
[2 9 6] -> [T F T]
...

does it scale?



does it compress?

Evaluation: Comparison with SOTA

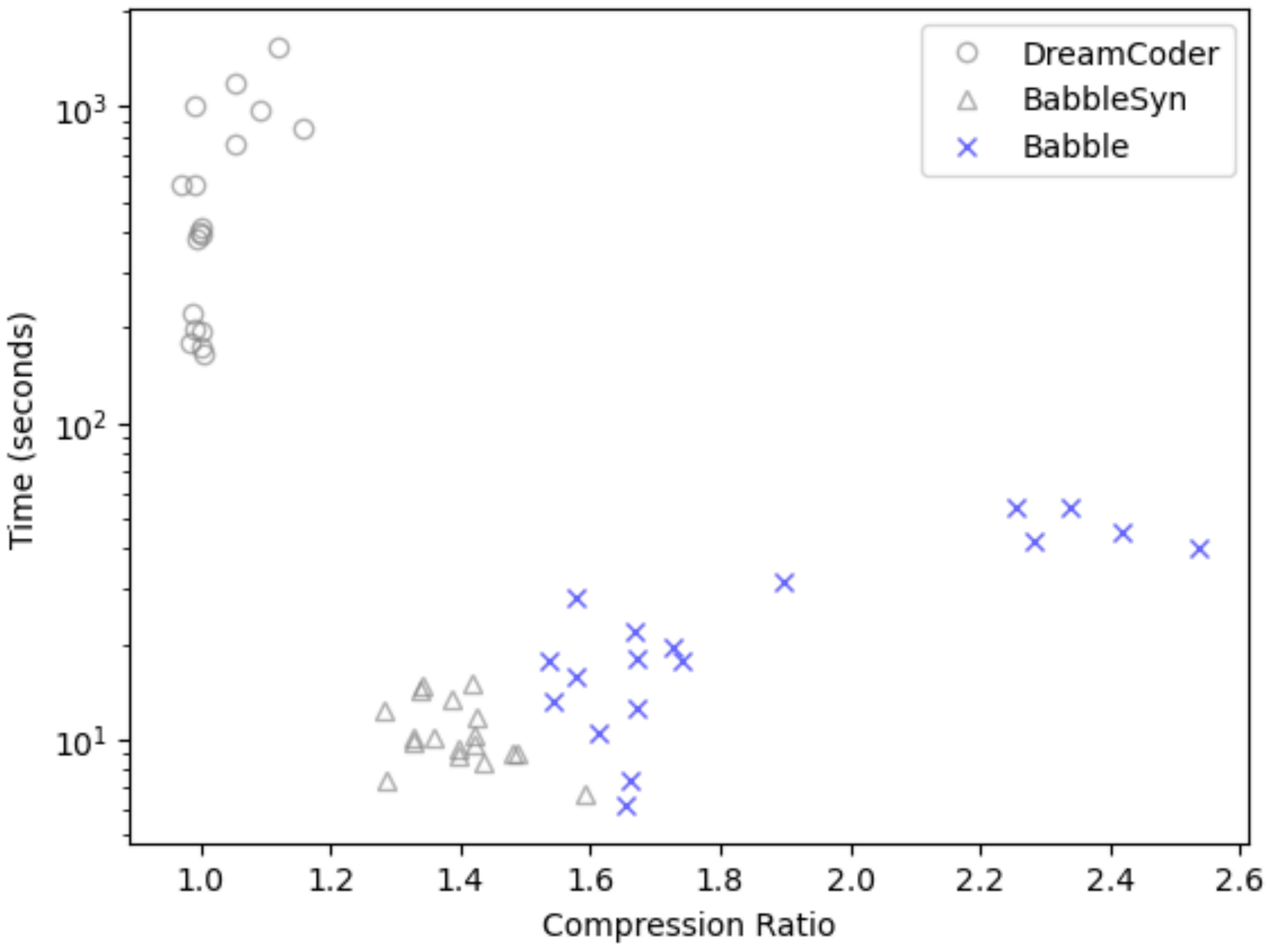
- **Baseline: DreamCoder (SOTA), BabbleSyn (Babble without Equality Thoery)**
- **Benchmarks: List domain**

Sum List
[1 2 3] -> 6
[4 6 8 1] -> 17

Double
[1 2 3] -> [2 4 6]
[4 5 1] -> [8 10 2]

Check Evens
[0 2 3] -> [T T F]
[2 9 6] -> [T F T]
...

does it scale?



does it compress?

Evaluation: Comparison with SOTA

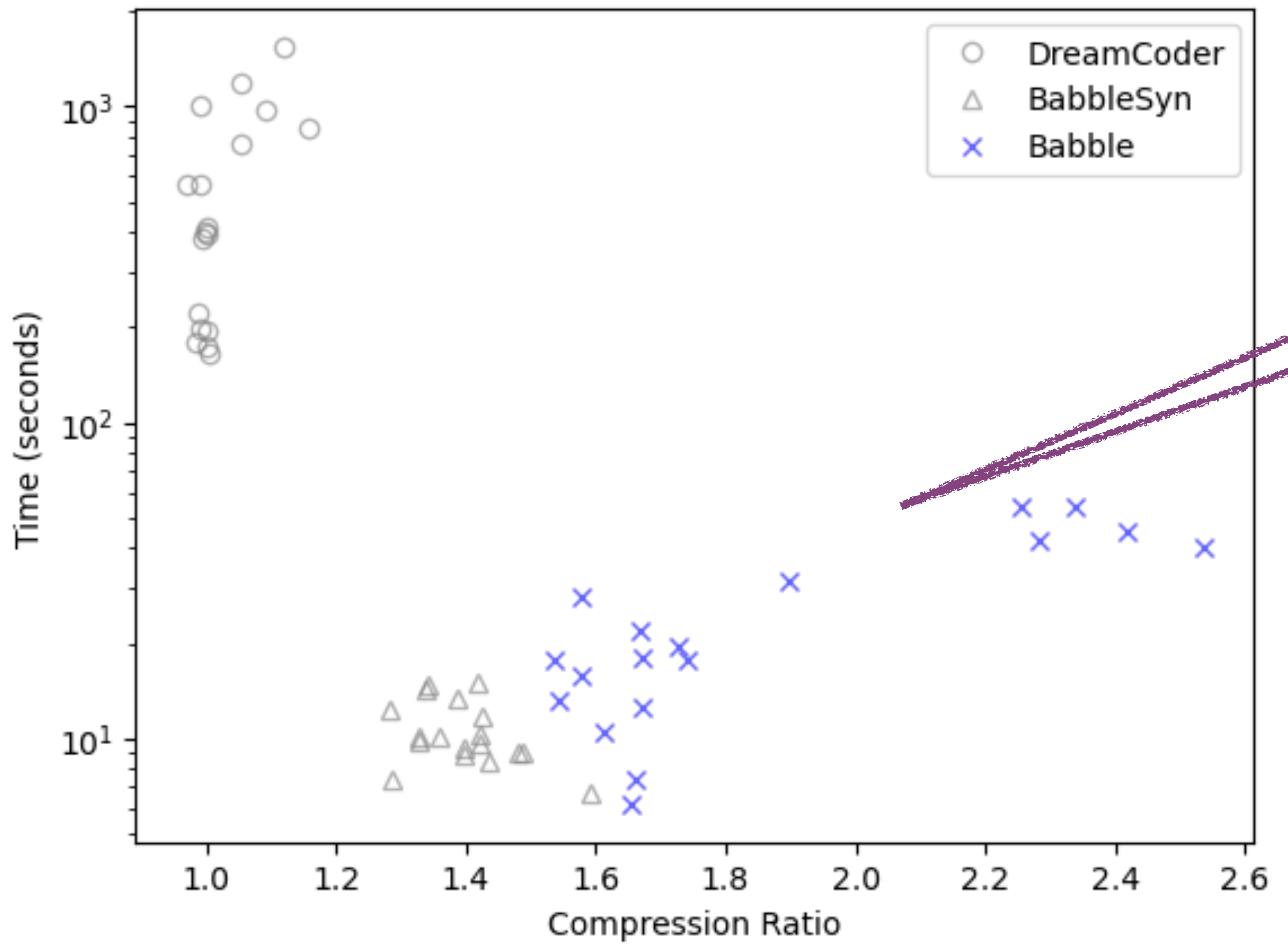
- **Baseline: DreamCoder (SOTA), BabbleSyn (Babble without Equality Thoery)**
- **Benchmarks: List domain**

Sum List
[1 2 3] -> 6
[4 6 8 1] -> 17

Double
[1 2 3] -> [2 4 6]
[4 5 1] -> [8 10 2]

Check Evens
[0 2 3] -> [T T F]
[2 9 6] -> [T F T]
...

does it scale?



faster (100x) &
higher Compression (60%) Ratio!!

does it compress?

Evaluation: Does E-graphs works?

- **Benchmarks: 2D CAD**

Benchmark	Input Size	Without E-Graphs: Compression Rate	Without E-Graphs: Time(s)	With E-Graphs: Compression Rate	With E-Graphs: Time(s)
Nuts & Bolts	19,009	9.23	19	10.90	40
Vehicles	35,427	5.47	79	6.44	78
Gadgets	35,713	5.25	75	7.09	82
Furniture	42,936	4.07	133	4.56	110

Evaluation: Does E-graphs works?

- **Benchmarks: 2D CAD**

		Without E-Graphs		With E-Graphs	
Benchmark	**Input Size**	**Compression Rate**	**Time(s)**	**Compression Rate**	**Time(s)**
Nuts & Bolts	**19,009**	**9.23**	**19**	**10.90**	**40**
Vehicles	**35,427**	**5.47**	**79**	**6.44**	**78**
Gadgets	**35,713**	**5.25**	**75**	**7.09**	**82**
Furniture	**42,936**	**4.07**	**133**	**4.56**	**110**

Higher Compression Ratio!!

Evaluation: Does E-graphs works?

- **Benchmarks: 2D CAD**

		Without E-Graphs		With E-Graphs	
Benchmark	**Input Size**	**Compression Rate**	**Time(s)**	**Compression Rate**	**Time(s)**
Nuts & Bolts	**19,009**	**9.23**	**19**	**10.90**	**40**
Vehicles	**35,427**	**5.47**	**79**	**6.44**	**78**
Gadgets	**35,713**	**5.25**	**75**	**7.09**	**82**
Furniture	**42,936**	**4.07**	**133**	**4.56**	**110**

Can reduce Time!!

Conclusion

- **Library Learning Modulo Theory** (LLMT)
    - learning abstractions (generate library) from a corpus
- **E-Graph Anti-Unification**
    - Efficiently generates candidate abstractions
- **Faster**, **Higher** Compression Ratio (CR)
    - 100x Faster, 60% higher CR than SOTA

Review (My Opinion)

- Strengths
  - E-graph reduce search space and make candidates smarter.
  - Formally defined new strategy for library learning, established a solid system

- Weaknesses
  - It is difficult to define **EQ** relations for higher languages such as C/C++
  - Too naive approach in using E-graph. (in E-class matching)

Appendix

Operand 1

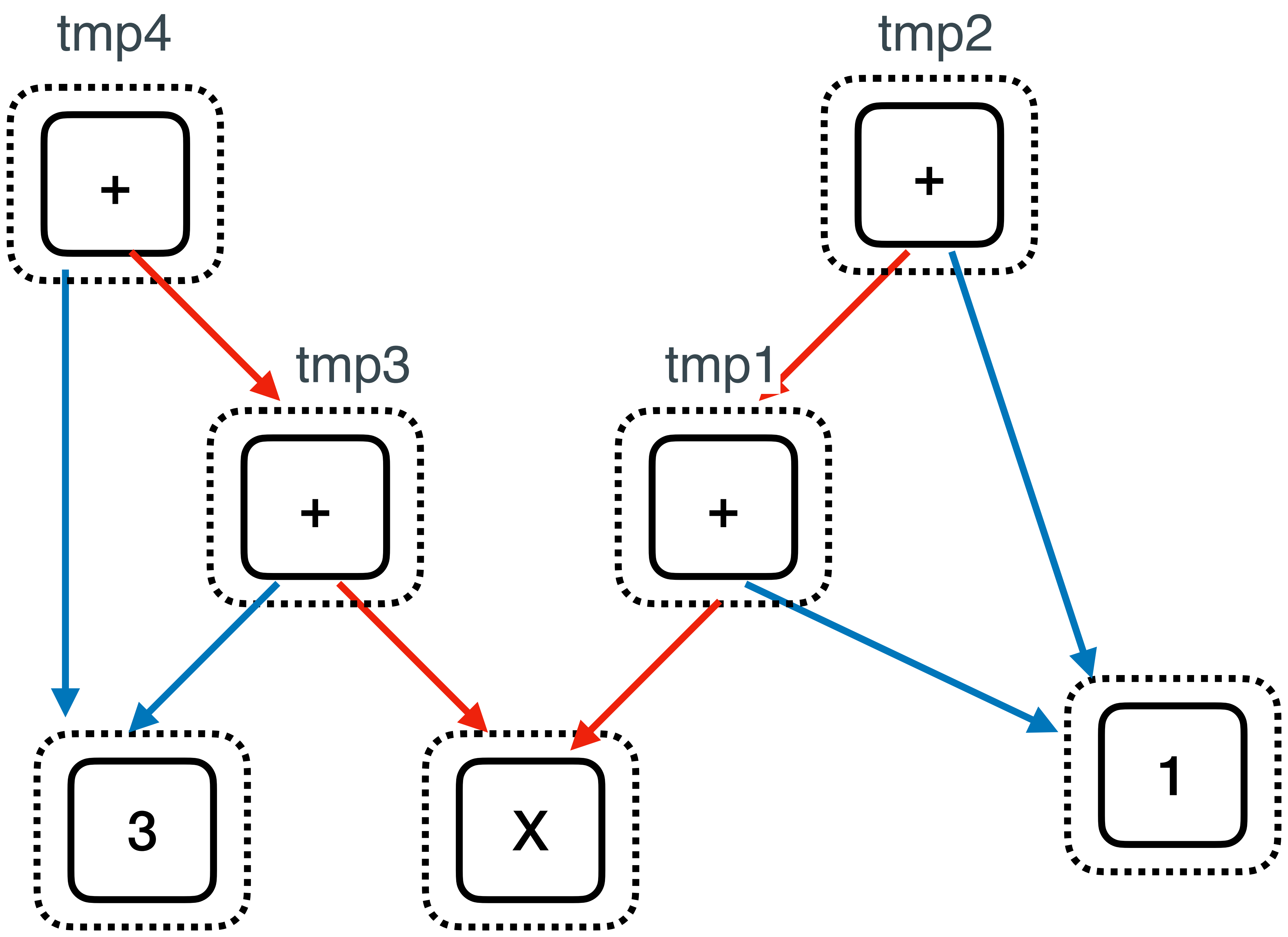
Operand 2

E-graph in Babble

- **Build E-Graph**

```
int foo1(int X, int Y) {
  int tmp1 = X + 1;
  int tmp2 = tmp1 + 1;
  return tmp2;
}
```

```
int foo2(int X, int Y) {
  int tmp3 = X + 3;
  int tmp4 = tmp1 + 3;
  return tmp2;
}
```



Operand 1

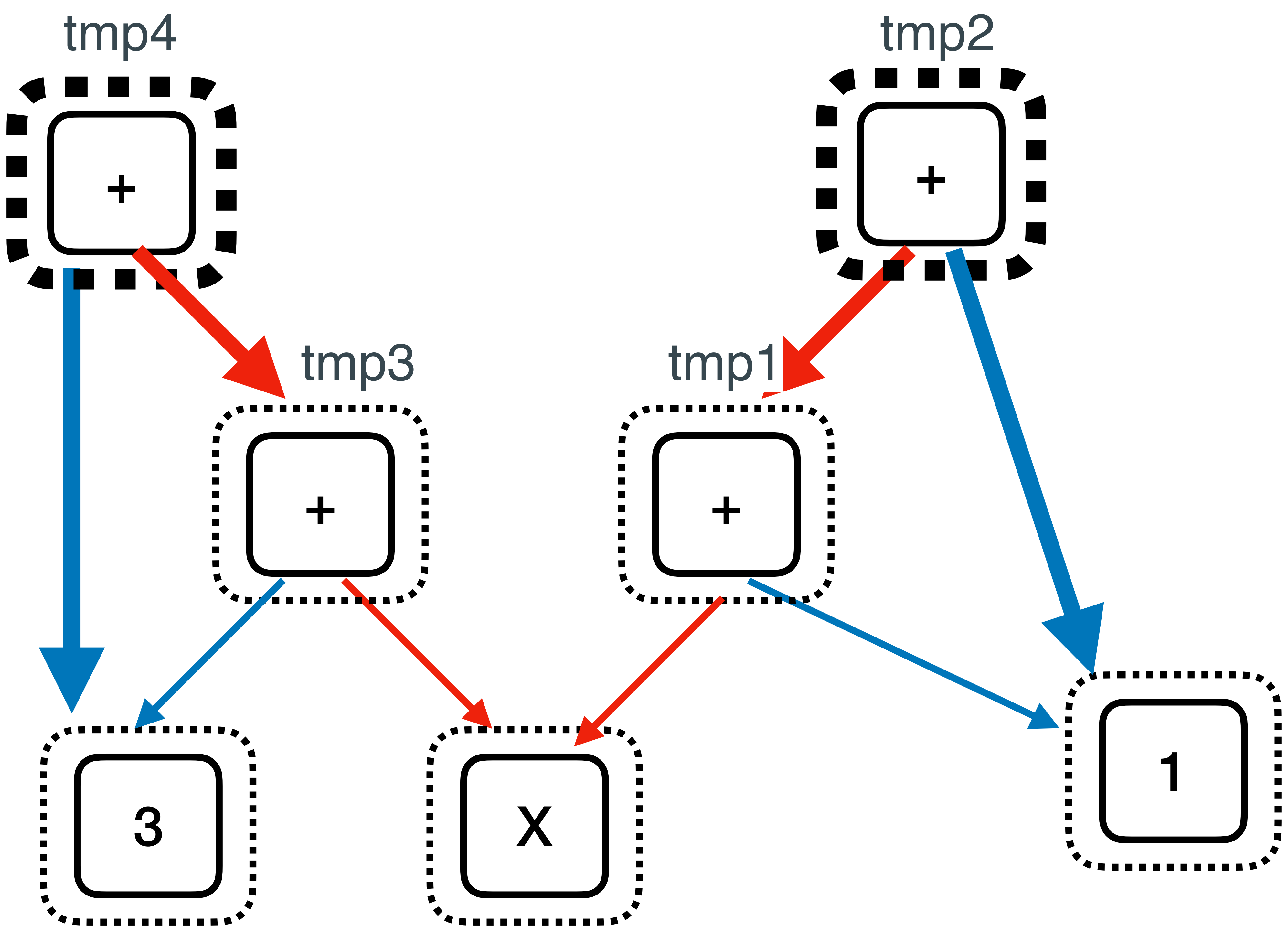
Operand 2

E-graph in Babble

- **Build E-Graph**

```
int foo1(int X, int Y) {
  int tmp1 = X + 1;
  int tmp2 = tmp1 + 1;
  return tmp2;
}
```

```
int foo2(int X, int Y) {
  int tmp3 = X + 3;
  int tmp4 = tmp1 + 3;
  return tmp2;
}
```



?? + ?? is common

$$\lambda x\ y \rightarrow x + y$$