babble

learning better abstractions *with e-graphs and anti-unification*

david cao†, rose kunkel†, chandrakana nandi, max willsey, zachary tatlock,
nadia polikarpova

uc san diego university of washington certora, inc.

† equal contribution

input 1

```
[scale 5 circle,
 move -2 -1.5 circle,
 move 2 -1.5 circle,
 move 0 2 (x-scale 3 circle)]
```

input 2

```
[scale 5 circle,
 move -2 -1.5 (rotate 90 line),
 move 2 -1.5 (rotate 90 line),
 move 0 2 (x-scale 3 line)]
```

input 1

```
[scale 5 circle,
 move -2 -1.5 circle,
 move 2 -1.5 circle,
 move 0 2 (x-scale 3 circle)]
```

input 2

```
[scale 5 circle,
 move -2 -1.5 (rotate 90 line),
 move 2 -1.5 (rotate 90 line),
 move 0 2 (x-scale 3 line)]
```

input 1

```
[scale 5 circle,
  move -2 -1.5 circle,
  move 2 -1.5 circle,
  move 0 2 (x-scale 3 circle)]
```

input 2

```
[scale 5 circle,
  move -2 -1.5 (rotate 90 line),
  move 2 -1.5 (rotate 90 line),
  move 0 2 (x-scale 3 line)]
```

input 1

```
[scale 5 circle,
 move -2 -1.5 circle,
 move 2 -1.5 circle,
 move 0 2 (x-scale 3 circle)]
```

input 2

```
[scale 5 circle,
 move -2 -1.5 (rotate 90 line),
 move 2 -1.5 (rotate 90 line),
 move 0 2 (x-scale 3 line)]
```

common pattern: face, where mouth is stretched out eyes

input 1

```
[scale 5 circle,
 move -2 -1.5 circle,
 move 2 -1.5 circle,
 move 0 2 (x-scale 3 circle)]
```

input 2

```
[scale 5 circle,
 move -2 -1.5 (rotate 90 line),
 move 2 -1.5 (rotate 90 line),
 move 0 2 (x-scale 3 line)]
```

input 1

```
[scale 5 circle,
 move -2 -1.5 circle,
 move 2 -1.5 circle,
 move 0 2 (x-scale 3 circle)]
```

input 2

```
[scale 5 circle,
 move -2 -1.5 (rotate 90 line),
 move 2 -1.5 (rotate 90 line),
 move 0 2 (x-scale 3 line)]
```

abstraction

```
let face = λshape →
 [scale 5 circle,
  move -2 -1.5 (rotate 90 shape),
  move 2 -1.5 (rotate 90 shape),
  move 0 2 (x-scale 3 shape)]
```

```
face circle
```

```
face line
```

```
let square = ...
 in face square
```

etc.

input 1

```
[scale 5 circle,
  move -2 -1.5 circle,
  move 2 -1.5 circle,
  move 0 2 (x-scale 3 circle)]
```

input 2

```
[scale 5 circle,
  move -2 -1.5 (rotate 90 line),
  move 2 -1.5 (rotate 90 line),
  move 0 2 (x-scale 3 line)]
```

abstraction


```
let face = λshape →
 [scale 5 circle,
  move -2 -1.5 (rotate 90 shape),
  move 2 -1.5 (rotate 90 shape),
  move 0 2 (x-scale 3 shape)]
```

```
face circle
```

```
face line
```

```
let square = ...
 in face square
```

etc.

input 1

```
[scale 5 circle,
  move -2 -1.5 circle,
  move 2 -1.5 circle,
  move 0 2 (x-scale 3 circle)]
```

input 2

```
[scale 5 circle,
  move -2 -1.5 (rotate 90 line),
  move 2 -1.5 (rotate 90 line),
  move 0 2 (x-scale 3 line)]
```

abstraction

humans are really good at this!

```
let face = λshape →
 [scale 5 circle,
  move -2 -1.5 (rotate 90 shape),
  move 2 -1.5 (rotate 90 shape),
  move 0 2 (x-scale 3 shape)]
```

```
face circle
```

```
face line
```

```
let square = ...
 in face square
```

etc.

input 1

```
[scale 5 circle,
 move -2 -1.5 circle,
 move 2 -1.5 circle,
 move 0 2 (x-scale 3 circle)]
```

input 2

```
[scale 5 circle,
 move -2 -1.5 (rotate 90 line),
 move 2 -1.5 (rotate 90 line),
 move 0 2 (x-scale 3 line)]
```

abstraction

humans are really good at this!

how can we get computers to do this?

```
let face = λshape →
 [scale 5 circle,
  move -2 -1.5 (rotate 90 shape),
  move 2 -1.5 (rotate 90 shape),
  move 0 2 (x-scale 3 shape)]
```

```
face circle
```

```
face line
```

```
let square = ...
 in face square
```

etc.

library learning

```
[scale 5 circle,
 move -2 -1.5 circle,
 move 2 -1.5 circle,
 move 0 2 (x-scale 3 circle)]
```

```
[scale 5 circle,
 move -2 -1.5 (rotate 90 line),
 move 2 -1.5 (rotate 90 line),
 move 0 2 (x-scale 3 line)]
```

algorithm to learn
"best" abstractions

```
let face = λshape →
 [scale 5 circle,
  move -2 -1.5 (rotate 90 shape),
  move 2 -1.5 (rotate 90 shape),
  move 0 2 (x-scale 3 shape)]
```

```
face circle
```

```
face line
```

library learning

```
[scale 5 circle,
 move -2 -1.5 circle,
 move 2 -1.5 circle,
 move 0 2 (x-scale 3 circle)]
```

```
[scale 5 circle,
 move -2 -1.5 (rotate 90 line),
 move 2 -1.5 (rotate 90 line),
 move 0 2 (x-scale 3 line)]
```

algorithm to learn
"best" abstractions

i.e. best compression:
minimize abstraction +
program size

```
let face = λshape →
 [scale 5 circle,
  move -2 -1.5 (rotate 90 shape),
  move 2 -1.5 (rotate 90 shape),
  move 0 2 (x-scale 3 shape)]
```

```
face circle
```

```
face line
```

library learning

```
[scale 5 circle,
 move -2 -1.5 circle,
 move 2 -1.5 circle,
 move 0 2 (x-scale 3 circle)]
```

```
[scale 5 circle,
 move -2 -1.5 (rotate 90 line),
 move 2 -1.5 (rotate 90 line),
 move 0 2 (x-scale 3 line)]
```

babble

algorithm to learn "best" abstractions

i.e. best compression: minimize abstraction + program size

```
let face = λshape →
 [scale 5 circle,
  move -2 -1.5 (rotate 90 shape),
  move 2 -1.5 (rotate 90 shape),
  move 0 2 (x-scale 3 shape)]
```

```
face circle
```

```
face line
```

who cares about library learning?

```
[scale 5 circle,
 move -2 -1.5 circle,
 move 2 -1.5 circle,
 move 0 2 (x-scale 3 circle)]
```



```
let face = λshape →
 [scale 5 circle,
  move -2 -1.5 (rotate 90 shape),
  move 2 -1.5 (rotate 90 shape),
  move 0 2 (x-scale 3 shape)]
```


fpga design

learn best library of operations to optimize hardware for

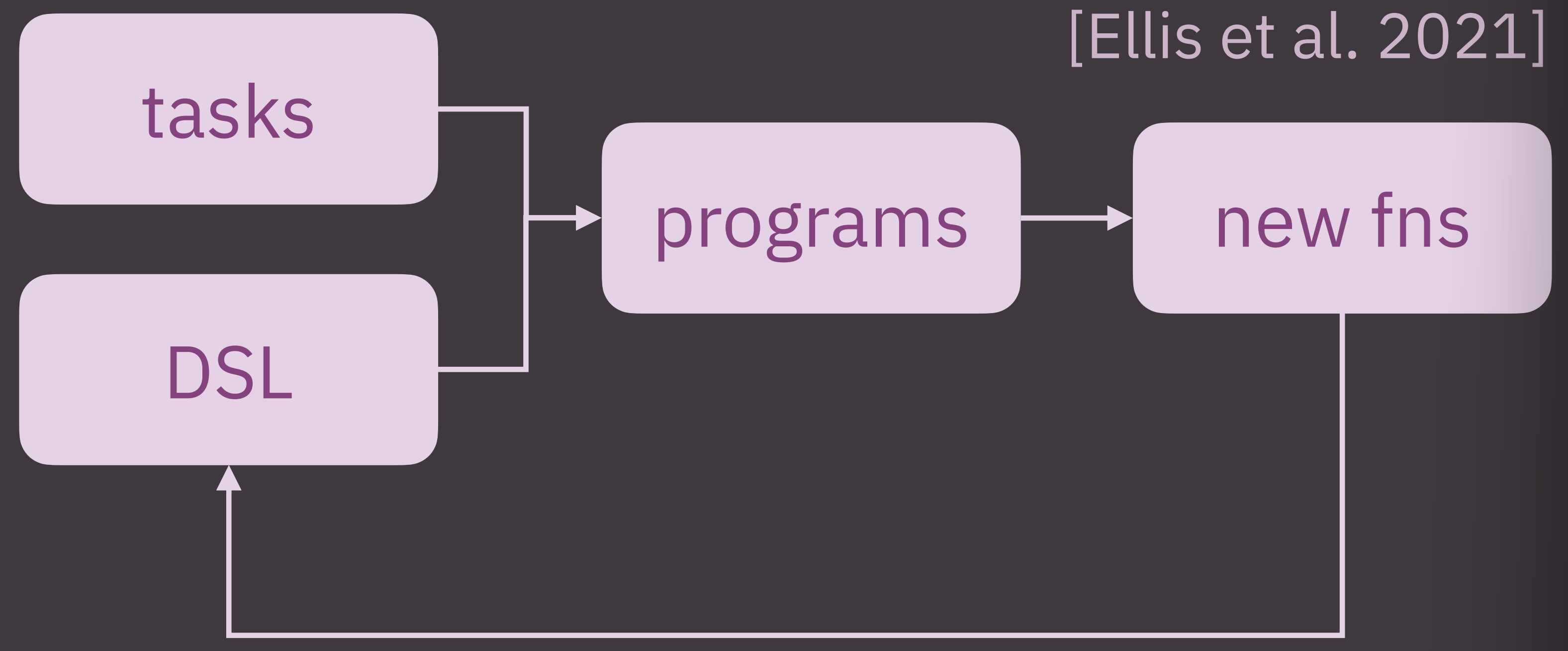
pick 2:

map | filter | foldl

improved program synthesis

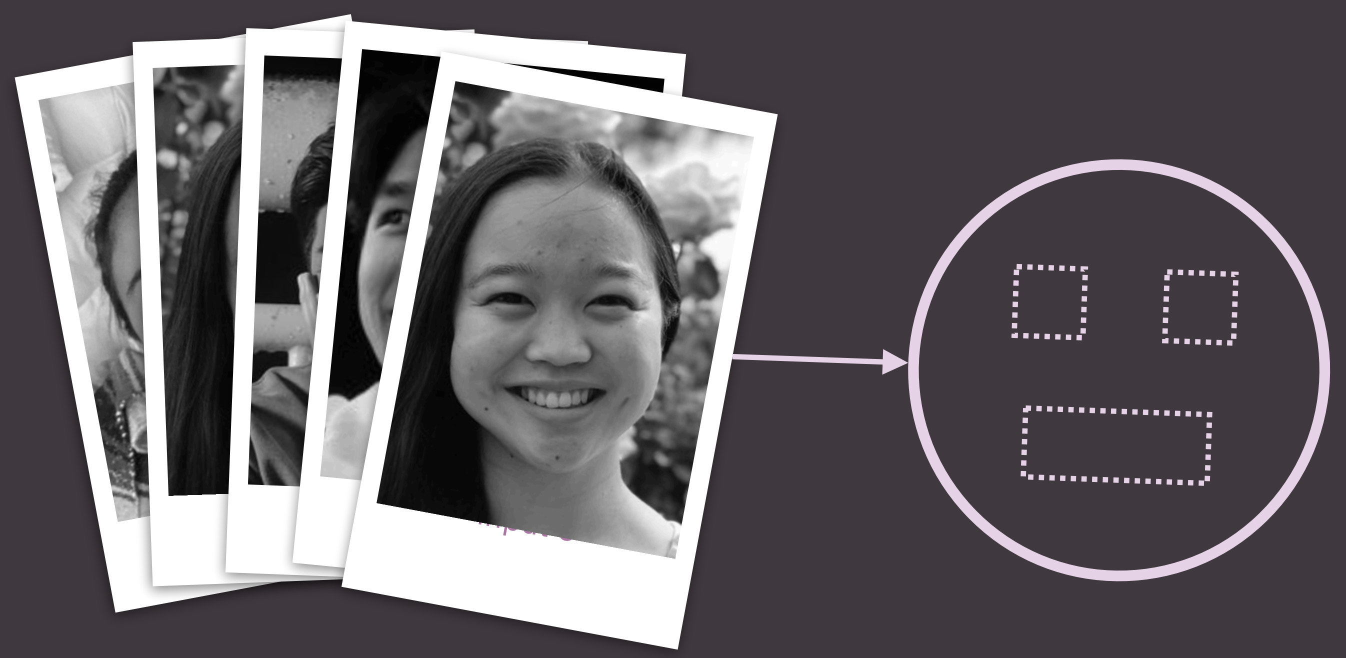
use past synthesis solutions to learn improved DSLs

[Ellis et al. 2021]



modeling human perception

make algorithm to examine how humans recognize visual structure



[Wang et al. 2021]

what's the challenge?

how does babble work?

how well does it work?

what's the challenge?

how does babble work?

how well does it work?

```
[scale 5 circle,
 move -2 -1.5 circle,
 move 2 -1.5 circle,
 move 0 2 (x-scale 3 circle)]
```



```
[scale 5 circle,
 move -2 -1.5 (rotate 90 line),
 move 2 -1.5 (rotate 90 line),
```

babble

algorithm to learn "best" abstractions

i.e. best compression: minimize abstraction +

```
let face = λshape →
 [scale 5 circle,
  move -2 -1.5 (rotate 90 shape),
  move 2 -1.5 (rotate 90 shape),
  move 0 2 (x-scale 3 shape)]
```



```
face circle
```

what's the challenge?

```
[scale 5 circle,
 move -2 -1.5 circle,
 move 2 -1.5 circle,
 move 0 2 (x-scale 3 circle)]
```



```
[scale 5 circle,
 move -2 -1.5 (rotate 90 line),
 move 2 -1.5 (rotate 90 line),
 move 0 2 (x-scale 3 line)]
```


a non-exhaustive list of requirements:

what's the challenge?

```
[scale 5 circle,
 move -2 -1.5 circle,
 move 2 -1.5 circle,
 move 0 2 (x-scale 3 circle)]
```



```
[scale 5 circle,
 move -2 -1.5 (rotate 90 line),
 move 2 -1.5 (rotate 90 line),
 move 0 2 (x-scale 3 line)]
```


a non-exhaustive list of requirements:

be scalable

run reasonably quickly on large corpuses with complex programs

what's the challenge?


```
[scale 5 circle,
 move -2 -1.5 circle,
 move 2 -1.5 circle,
 move 0 2 (x-scale 3 circle)]
```



```
[scale 5 circle,
 move -2 -1.5 (rotate 90 line),
 move 2 -1.5 (rotate 90 line),
 move 0 2 (x-scale 3 line)]
```


a non-exhaustive list of requirements:

be scalable

run reasonably quickly on large corpuses with complex programs

learn abstractions in subterms

not just abstractions which can be applied at the top level

what's the challenge?


```
[scale 5 circle,
 move -2 -1.5 circle,
 move 2 -1.5 circle,
 move 0 2 (x-scale 3 circle)]
```



```
[scale 5 circle,
 move -2 -1.5 (rotate 90 line),
 move 2 -1.5 (rotate 90 line),
 move 0 2 (x-scale 3 line)]
```


a non-exhaustive list of requirements:

be scalable

run reasonably quickly on large corpuses with complex programs

learn abstractions in subterms

not just abstractions which can be applied at the top level

handles nested abstractions

allowing for common structure across abstractions themselves to be shared

what's the challenge?

```
[scale 5 circle,
 move -2 -1.5 circle,
 move 2 -1.5 circle,
 move 0 2 (x-scale 3 circle)]
```



```
[scale 5 circle,
 move -2 -1.5 (rotate 90 line),
 move 2 -1.5 (rotate 90 line),
 move 0 2 (x-scale 3 line)]
```


what babble tackles:

incorporating semantic equivalence

what's the challenge?


```
[scale 5 circle,
 move -2 -1.5 circle,
 move 2 -1.5 circle,
 move 0 2 (x-scale 3 circle)]
```

```
[scale 5 circle,
 move -2 -1.5 (rotate 90 line),
 move 2 -1.5 (rotate 90 line),
 move 0 2 (x-scale 3 line)]
```

what's the challenge?
syntactic alignment

input 1

```
[scale 5 circle,
  move -2 -1.5 circle,
  move 2 -1.5 circle,
  move 0 2 (x-scale 3 circle)]
```

input 2

```
[scale 5 circle,
  move -2 -1.5 (rotate 90 line),
  move 2 -1.5 (rotate 90 line),
  move 0 2 (x-scale 3 line)]
```

what's the challenge?

syntactic alignment

input 1

```
[scale 5 circle,
  move -2 -1.5 circle,
  move 2 -1.5 circle,
  move 0 2 (x-scale 3 circle)]
```

input 2

```
[scale 5 circle,
  move -2 -1.5 (rotate 90 line),
  move 2 -1.5 (rotate 90 line),
  move 0 2 (x-scale 3 line)]
```

arg 1: `circle` `(rotate 90 line)`

what's the challenge?
syntactic alignment

input 1

```
[scale 5 circle,
  move -2 -1.5 circle,
  move 2 -1.5 circle,
  move 0 2 (x-scale 3 circle)]
```

input 2

```
[scale 5 circle,
  move -2 -1.5 (rotate 90 line),
  move 2 -1.5 (rotate 90 line),
  move 0 2 (x-scale 3 line)]
```

arg 1: `circle` `(rotate 90 line)`

arg 2: `circle` `line`

what's the challenge?

syntactic alignment

```
[scale 5 circle,
 move -2 -1.5 circle,
 move 2 -1.5 circle,
 move 0 2 (x-scale 3 circle)]
```

```
[scale 5 circle,
 move -2 -1.5 (rotate 90 line),
 move 2 -1.5 (rotate 90 line),
 move 0 2 (x-scale 3 line)]
```

```
let face-ish = λeye mouth →
 [scale 5 circle,
  move -2 -1.5 eye,
  move 2 -1.5 eye,
  move 0 2 (x-scale 3 mouth)]
```

```
face-ish circle circle
```

```
face-ish (rotate 90 line) line
```

what's the challenge?
syntactic alignment

```
[scale 5 circle,
 move -2 -1.5 circle,
 move 2 -1.5 circle,
 move 0 2 (x-scale 3 circle)]
```



```
[scale 5 circle,
 move -2 -1.5 (rotate 90 line),
 move 2 -1.5 (rotate 90 line),
 move 0 2 (x-scale 3 line)]
```


purely syntactic

```
let face-ish = λeye mouth →
 [scale 5 circle,
  move -2 -1.5 eye,
  move 2 -1.5 eye,
  move 0 2 (x-scale 3 mouth)]
```

```
face-ish circle circle
```

```
face-ish (rotate 90 line) line
```

ideal

```
let face = λshape →
 [scale 5 circle,
  move -2 -1.5 (rotate 90 shape),
  move 2 -1.5 (rotate 90 shape),
  move 0 2 (x-scale 3 shape)]
```

```
face circle
```

```
face line
```

what's the challenge?
syntactic alignment

```
[scale 5 circle,
 move -2 -1.5 circle,
 move 2 -1.5 circle,
 move 0 2 (x-scale 3 circle)]
```



```
[scale 5 circle,
 move -2 -1.5 (rotate 90 line),
 move 2 -1.5 (rotate 90 line),
 move 0 2 (x-scale 3 line)]
```


purely syntactic

```
let face-ish = λeye mouth →
 [scale 5 circle,
  move -2 -1.5 eye,
  move 2 -1.5 eye,
  move 0 2 (x-scale 3 mouth)]
```

```
face-ish circle circle
```

```
face-ish (rotate 90 line) line
```

```
face-ish circle circle
```

```
face-ish (rotate 90 line) line
```

```
face-ish circle circle
```

ideal

```
let face = λshape →
 [scale 5 circle,
  move -2 -1.5 (rotate 90 shape),
  move 2 -1.5 (rotate 90 shape),
  move 0 2 (x-scale 3 shape)]
```

```
face circle
```

```
face line
```

```
face circle
```

```
face line
```

```
face circle
```

what's the challenge?
syntactic alignment

```
[scale 5 circle,
 move -2 -1.5 circle,
 move 2 -1.5 circle,
 move 0 2 (x-scale 3 circle)]
```

```
[scale 5 circle,
 move -2 -1.5 (rotate 90 line),
 move 2 -1.5 (rotate 90 line),
 move 0 2 (x-scale 3 line)]
```

ideal

```
let face = λshape →
 [scale 5 circle,
  move -2 -1.5 (rotate 90 shape),
  move 2 -1.5 (rotate 90 shape),
  move 0 2 (x-scale 3 shape)]
```

```
face circle
```

```
face line
```

what's the challenge?
syntactic alignment

```
[scale 5 circle,
 move -2 -1.5 circle,
 move 2 -1.5 circle,
 move 0 2 (x-scale 3 circle)]
```

```
[scale 5 circle,
 move -2 -1.5 (rotate 90 line),
 move 2 -1.5 (rotate 90 line),
 move 0 2 (x-scale 3 line)]
```

ideal

```
let face = λshape →
 [scale 5 circle,
  move -2 -1.5 (rotate 90 shape),
  move 2 -1.5 (rotate 90 shape),
  move 0 2 (x-scale 3 shape)]
```

```
face circle
```

```
face line
```

what's the challenge?
syntactic alignment

```
[scale 5 circle,
 move -2 -1.5 circle,
 move 2 -1.5 circle,
 move 0 2 (x-scale 3 circle)]
```

≡

```
[scale 5 circle,
 move -2 -1.5 (rotate 90 circle),
 move 2 -1.5 (rotate 90 circle),
 move 0 2 (x-scale 3 circle)]
```

```
[scale 5 circle,
 move -2 -1.5 (rotate 90 line),
 move 2 -1.5 (rotate 90 line),
 move 0 2 (x-scale 3 line)]
```

ideal

```
let face = λshape →
 [scale 5 circle,
  move -2 -1.5 (rotate 90 shape),
  move 2 -1.5 (rotate 90 shape),
  move 0 2 (x-scale 3 shape)]
```

```
face circle
```

```
face line
```

semantically equivalent version of input shares common structure!

what's the challenge?

a library learning approach which discovers **more precise structure** by considering **semantically equivalent versions** of our programs.

what's the challenge?

how does babble work?

how well does it work?

```
[scale 5 circle,
 move -2 -1.5 circle,
 move 2 -1.5 circle,
 move 0 2 (x-scale 3 circle)]
```


≡

```
[scale 5 circle,
 move -2 -1.5 (rotate 90 circle),
 move 2 -1.5 (rotate 90 circle),
 move 0 2 (x-scale 3 circle)]
```



```
[scale 5 circle,
 move -2 -1.5 (rotate 90 line),
 move 2 -1.5 (rotate 90 line)
```

ideal

```
let face = λshape →
 [scale 5 circle,
  move -2 -1.5 (rotate 90 shape),
  move 2 -1.5 (rotate 90 shape),
  move 0 2 (x-scale 3 shape)]
```

```
face circle
```

```
face line
```

what's the challenge?

how does babble work?

how well does it work?

```
[scale 5 circle,
 move -2 -1.5 circle,
 move 2 -1.5 circle,
 move 0 2 (x-scale 3 circle)]
```

≡

```
[scale 5 circle,
 move -2 -1.5 (rotate 90 circle),
 move 2 -1.5 (rotate 90 circle),
 move 0 2 (x-scale 3 circle)]
```

```
[scale 5 circle,
 move -2 -1.5 (rotate 90 line),
 move 2 -1.5 (rotate 90 line)
```

ideal

```
let face = λshape →
 [scale 5 circle,
  move -2 -1.5 (rotate 90 shape),
  move 2 -1.5 (rotate 90 shape),
  move 0 2 (x-scale 3 shape)]
```

```
face circle
```

```
face line
```

how does babble work?
intuition

```
[scale 5 circle,
 move -2 -1.5 circle,
 move 2 -1.5 circle,
 move 0 2 (x-scale 3 circle)]
```

```
[scale 5 circle,
 move -2 -1.5 (rotate 90 line),
 move 2 -1.5 (rotate 90 line),
 move 0 2 (x-scale 3 line)]
```

≡

input 1, version 2

```
[scale 5 circle,
 move -2 -1.5 (rotate 90 circle),
 move 2 -1.5 (rotate 90 circle),
 move 0 2 (x-scale 3 circle)]
```

≡

input 1, version 3

```
[scale 5 circle,
 move -2 -1.5 (rotate 90 circle),
 move 2 -1.5 (rotate 90 circle),
 move 0 2 (x-scale 3 circle)]
```

idea:
consider every equivalent version of our inputs

how does babble work?
changing the problem

```
[scale 5 circle,
 move -2 -1.5 circle,
 move 2 -1.5 circle,
 move 0 2 (x-scale 3 circle)]
```

```
[scale 5 circle,
 move -2 -1.5 (rotate 90 line),
 move 2 -1.5 (rotate 90 line),
 move 0 2 (x-scale 3 line)]
```

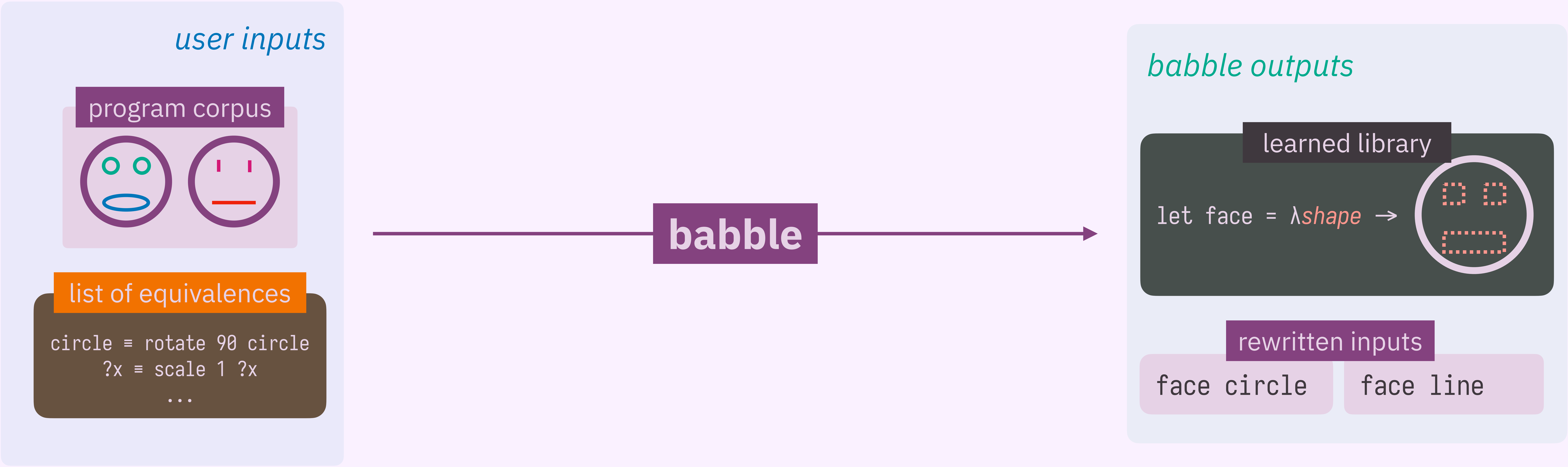
babble →

```
let face = λshape →
 [scale 5 circle,
  move -2 -1.5 (rotate 90 shape),
  move 2 -1.5 (rotate 90 shape),
  move 0 2 (x-scale 3 shape)]
```

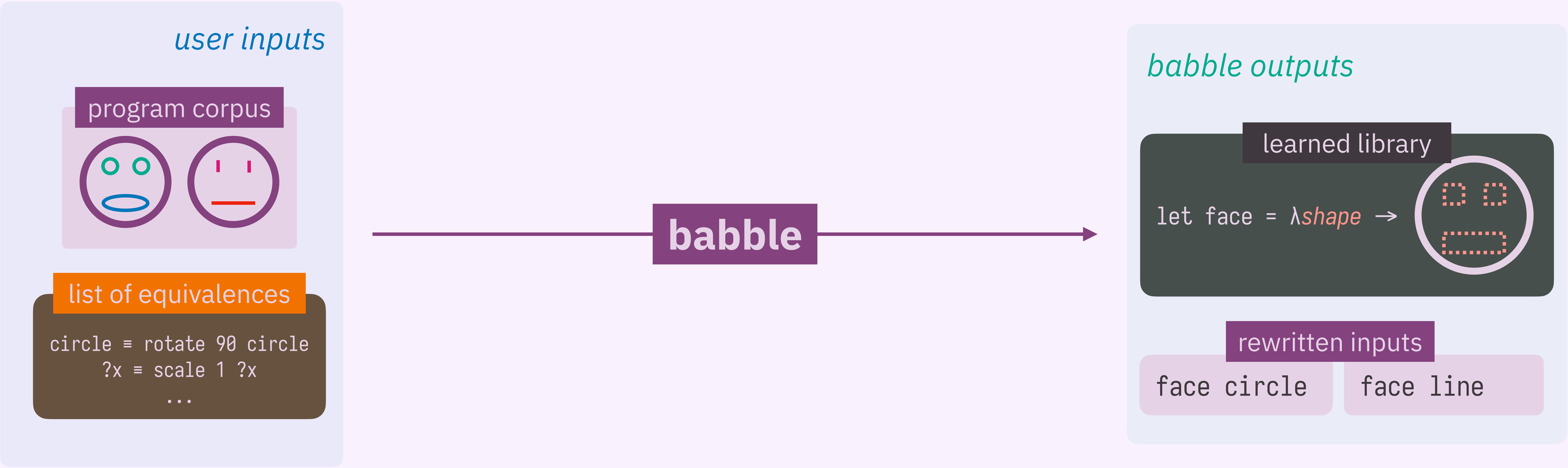
```
face circle
```

```
face line
```

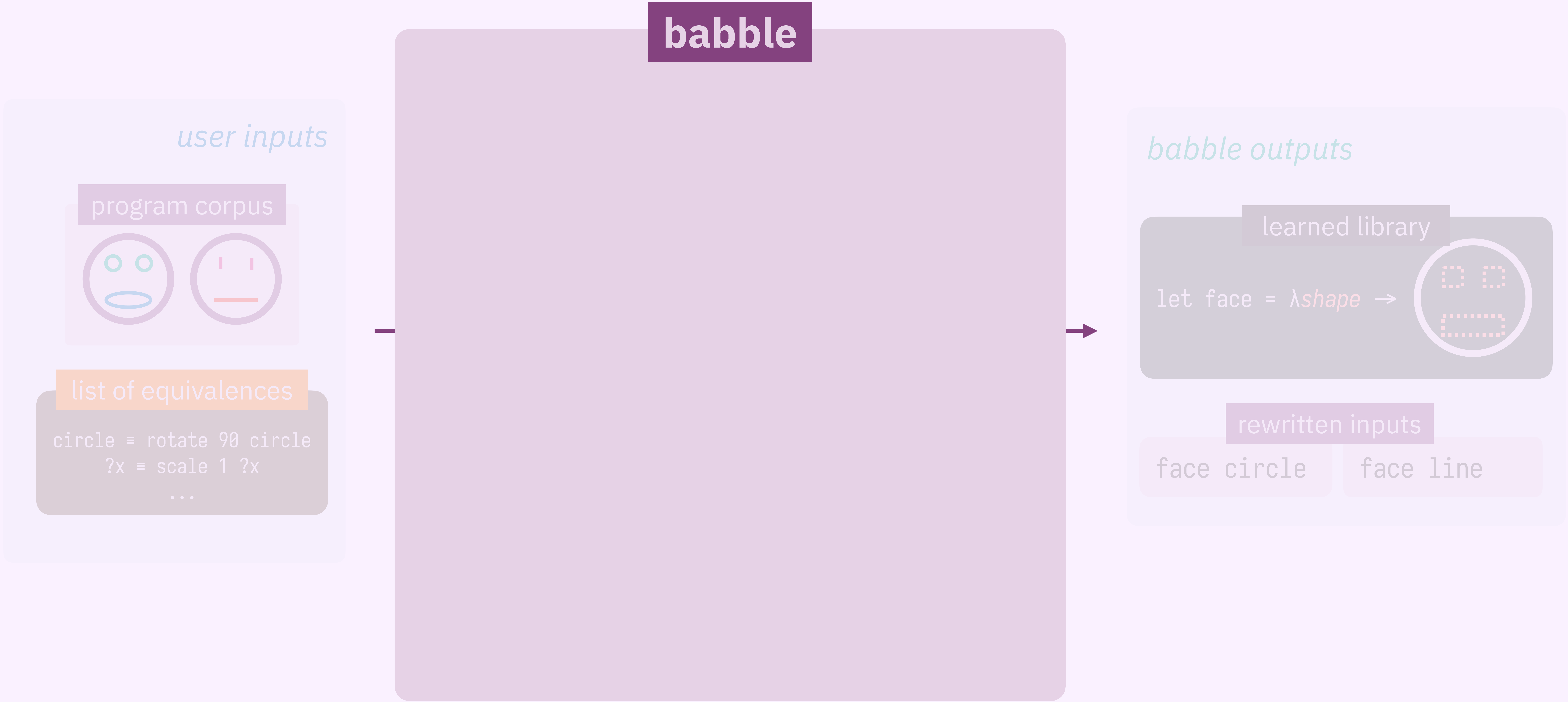
how does babble work?
changing the problem



how does babble work?
contribution 1: library learning modulo theory (LLMT)



how does babble work?
library learning modulo theory (LLMT)



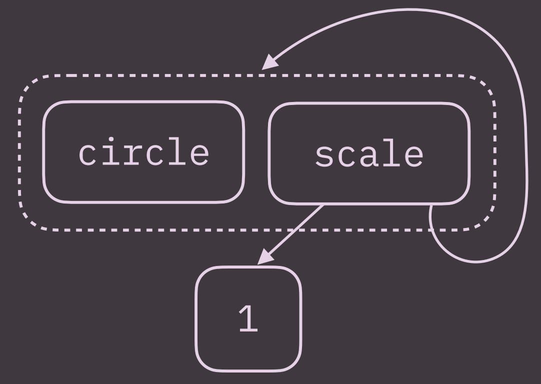
how does babble work?
library learning modulo theory (LLMT)

babble

insight

use e-graphs

to create and store infinite equivalent input variants



user inputs

program corpus

list of equivalences

```
circle ≡ rotate 90 circle
?x ≡ scale 1 ?x
...
```

babble outputs

learned library

`let face = λ`*`shape`*` →`

rewritten inputs

`face circle` `face line`

how does babble work?
library learning modulo theory (LLMT)

user inputs

program corpus

list of equivalences

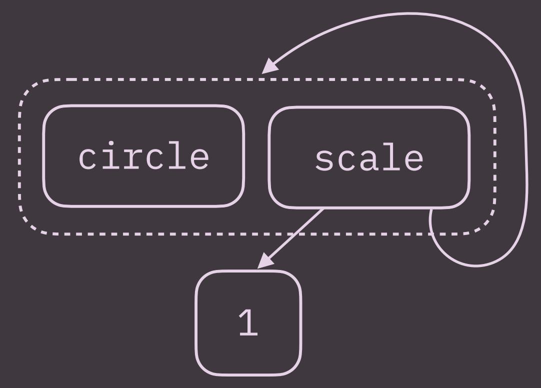
```
circle ≡ rotate 90 circle
?x ≡ scale 1 ?x
...
```

babble

insight

use e-graphs

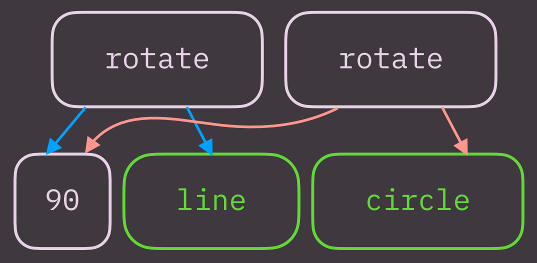
to create and store infinite equivalent input variants



contribution 2

e-graph anti-unification

to propose candidate abstractions in the presence of input variants



rotate 90 ?x

babble outputs

learned library

let face = λ*shape* →

rewritten inputs

`face circle` `face line`

how does babble work?
library learning modulo theory (LLMT)

user inputs

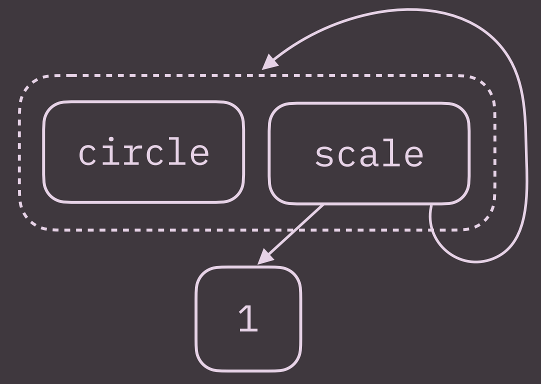
program corpus

list of equivalences

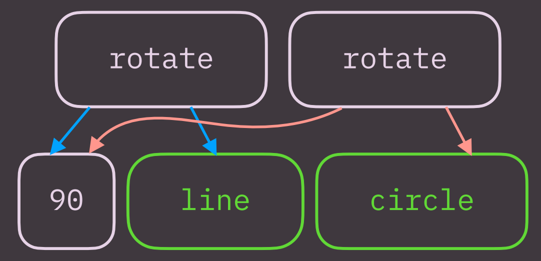
```
circle ≡ rotate 90 circle
?x ≡ scale 1 ?x
...
```

babble

insight
use e-graphs
to create and store infinite equivalent input variants



contribution 2
e-graph anti-unification
to propose candidate abstractions in the presence of input variants




```
rotate 90 ?x
```

contribution 3
targeted CSE
to pick the best programs and abstractions

```
let f = λx -> … in
 let f = λx -> … in
  let f = λx -> … in
   f 2
```

babble outputs

learned library

let face = λ*shape* →

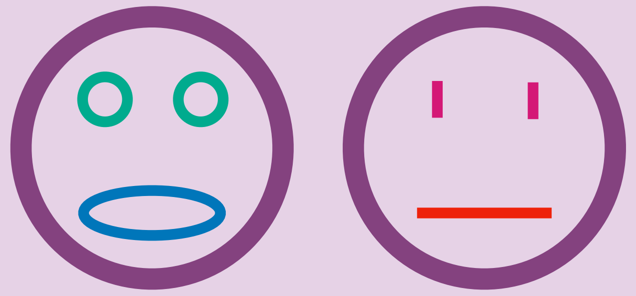
rewritten inputs

```
face circle    face line
```

how does babble work?
library learning modulo theory (LLMT)

user inputs

program corpus

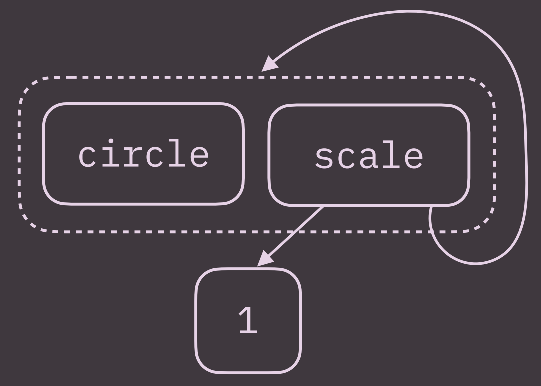


list of equivalences

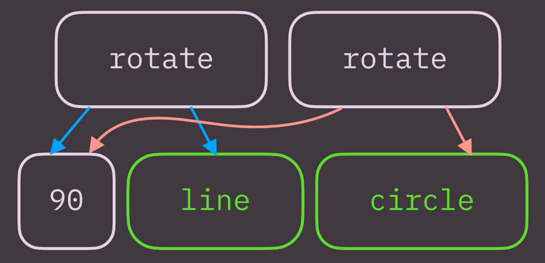
```
circle ≡ rotate 90 circle
?x ≡ scale 1 ?x
...
```

babble

insight
use e-graphs
to create and store infinite equivalent input variants



contribution 2
e-graph anti-unification
to propose candidate abstractions in the presence of input variants




```
rotate 90 ?x
```

contribution 3
targeted CSE
to pick the best programs and abstractions

```
let f = λx -> … in
 let f = λx -> … in
  let f = λx -> … in
   f 2
```

babble outputs

learned library

let face = λ*shape* →

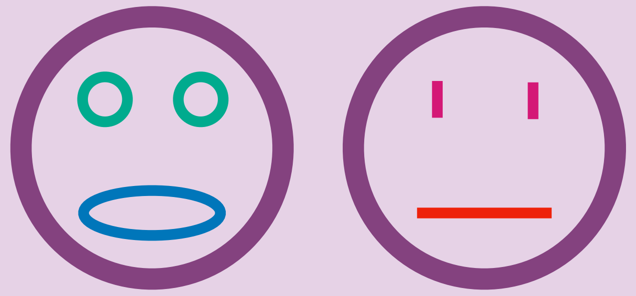
rewritten inputs

```
face circle
face line
```

how does babble work?
library learning modulo theory (LLMT)

user inputs

program corpus

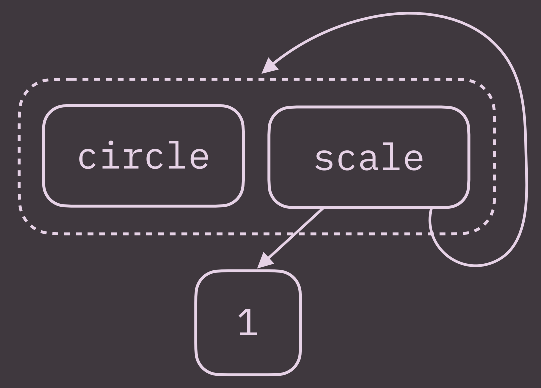


list of equivalences

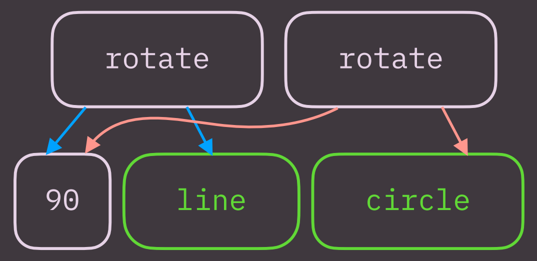
```
circle ≡ rotate 90 circle
?x ≡ scale 1 ?x
...
```

babble

insight
use e-graphs
to create and store infinite equivalent input variants



contribution 2
e-graph anti-unification
to propose candidate abstractions in the presence of input variants



`rotate 90 ?x`

contribution 3
targeted CSE
to pick the best programs...

see the paper for more on this!

```
let f = λx -> … in
let f = λx -> … in
let f = λx -> … in
f 2
```

babble outputs

learned library

`let face = λshape →`

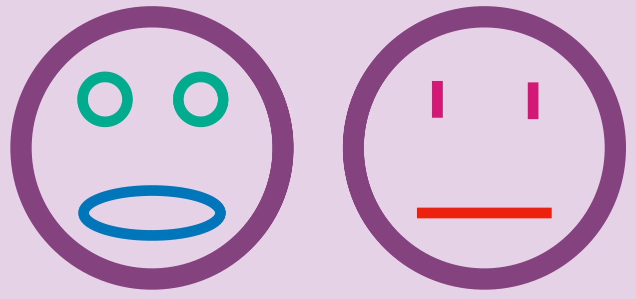
rewritten inputs

`face circle` `face line`

how does babble work?
library learning modulo theory (LLMT)

user inputs

program corpus

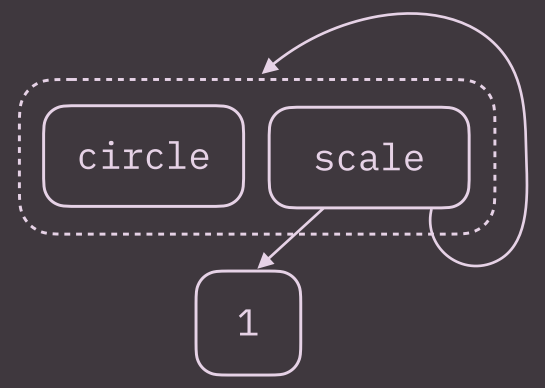


list of equivalences

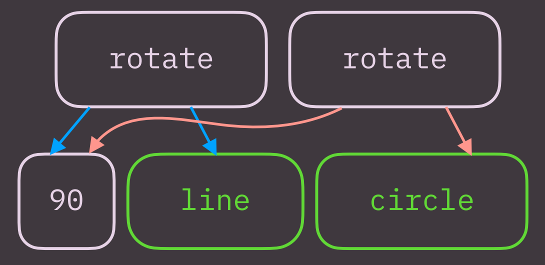
```
circle ≡ rotate 90 circle
?x ≡ scale 1 ?x
...
```

babble

insight
use e-graphs
to create and store infinite equivalent input variants



contribution 2
e-graph anti-unification
to propose candidate abstractions in the presence of input variants



`rotate 90 ?x`

contribution 3
targeted CSE
to pick the best programs ...

see the paper for more on this!

```
let f = λx -> … in
 let f = λx -> … in
  let f = λx -> … in
   f 2
```

babble outputs

learned library

`let face = λshape →`

rewritten inputs

`face circle` `face line`

how do **e-graphs** work?

how do e-graphs work?

why e-graphs?

```
[scale 5 circle,
 move -2 -1.5 circle,
 move 2 -1.5 circle,
 move 0 2 (x-scale 3 circle)]
```

```
[scale 5 circle,
 move -2 -1.5 (rotate 90 line),
 move 2 -1.5 (rotate 90 line),
 move 0 2 (x-scale 3 line)]
```

how do **e-graphs** work?

why e-graphs?

user inputs

```
[scale 5 circle,
 move -2 -1.5 circle,
 move 2 -1.5 circle,
 move 0 2 (x-scale 3 circle)]
```

```
[scale 5 circle,
 move -2 -1.5 (rotate 90 line),
 move 2 -1.5 (rotate 90 line),
 move 0 2 (x-scale 3 line)]
```

list of equivalences

```
circle ≡ rotate 90 circle
?x ≡ scale 1 ?x
...
```

≡

input 1, version 2

```
[scale 5 circle,
 move -2 -1.5 (rotate 90 circle),
 move 2 -1.5 (rotate 90 circle),
 move 0 2 (x-scale 3 circle)]
```

≡

input 1, version 3

```
[scale 5 circle,
 move -2 -1.5 (rotate
 move 2 -1.5 (rotate 9
 move 0 2 (x-scale 3 c
```

how do **e-graphs** work?

why e-graphs?

user inputs

```
[scale 5 circle,
 move -2 -1.5 circle,
 move 2 -1.5 circle,
 move 0 2 (x-scale 3 circle)]
```

```
[scale 5 circle,
 move -2 -1.5 (rotate 90 line),
 move 2 -1.5 (rotate 90 line),
 move 0 2 (x-scale 3 line)]
```

list of equivalences

```
circle ≡ rotate 90 circle
?x ≡ scale 1 ?x
...
```

≡

input 1, version 2

```
[scale 5 circle,
 move -2 -1.5 (rotate 90 circle),
 move 2 -1.5 (rotate 90 circle),
 move 0 2 (x-scale 3 circle)]
```

≡

input 1, version 3

```
[scale 5 circle,
 move -2 -1.5 (rotate
 move 2 -1.5 (rotate 9
 move 0 2 (x-scale 3 c
```

challenge:

when do we stop rewriting?

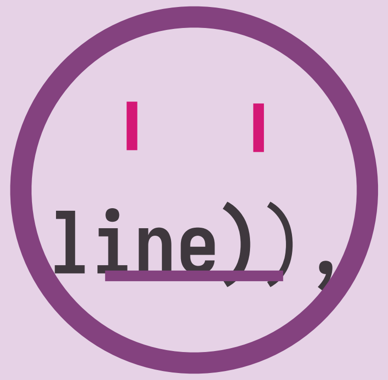
how do **e-graphs** work?

why e-graphs?

user inputs

```
[scale 5 circle,
 move -2 -1.5 circle,
 move 2 -1.5 circle,
 move 0 2 (x-scale 3 circle)]
```

```
[scale 5 circle,
 move -2 -1.5 (rotate 90 line),
 move 2 -1.5 (rotate 90 (scale 1 line)),
 move 0 2 (x-scale 3 line)]
```



list of equivalences

```
circle ≡ rotate 90 circle
?x ≡ scale 1 ?x
...
```

≡

input 1, version 2

```
[scale 5 circle,
 move -2 -1.5 (rotate 90 circle),
 move 2 -1.5 (rotate 90 circle),
 move 0 2 (x-scale 3 circle)]
```

≡

input 1, version 3

```
[scale 5 circle,
 move -2 -1.5 (rotate
 move 2 -1.5 (rotate 9
 move 0 2 (x-scale 3 c
```

challenge:

when do we stop rewriting?

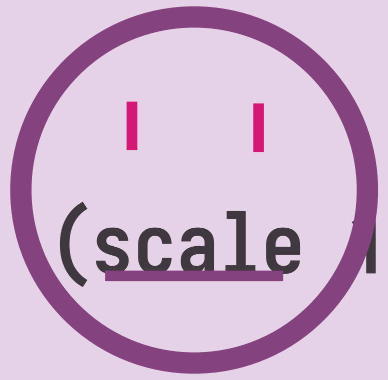
how do **e-graphs** work?

why e-graphs?

user inputs

```
[scale 5 circle,
 move -2 -1.5 circle,
 move 2 -1.5 circle,
 move 0 2 (x-scale 3 circle)]
```

```
[scale 5 circle,
 move -2 -1.5 (rotate 90 line),
 move 2 -1.5 (rotate 90 (scale 1 (scale 1 line))),
 move 0 2 (x-scale 3 line)]
```



list of equivalences

```
circle ≡ rotate 90 circle
?x ≡ scale 1 ?x
...
```

≡

input 1, version 2

```
[scale 5 circle,
 move -2 -1.5 (rotate 90 circle),
 move 2 -1.5 (rotate 90 circle),
 move 0 2 (x-scale 3 circle)]
```

≡

input 1, version 3

```
[scale 5 circle,
 move -2 -1.5 (rotate
 move 2 -1.5 (rotate 9
 move 0 2 (x-scale 3 c
```

challenge:

when do we stop rewriting?

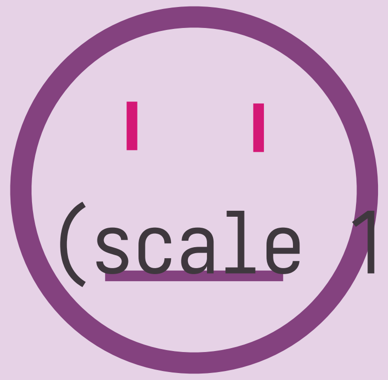
how do e-graphs work?

why e-graphs?

user inputs

```
[scale 5 circle,
 move -2 -1.5 circle,
 move 2 -1.5 circle,
 move 0 2 (x-scale 3 circle)]
```

```
[scale 5 circle,
 move -2 -1.5 (rotate 90 line),
 move 2 -1.5 (rotate 90 (scale 1 (scale 1 (scale 1 line)))),
 move 0 2 (x-scale 3 line)]
```



list of equivalences

```
circle ≡ rotate 90 circle
?x ≡ scale 1 ?x
...
```

≡

input 1, version 2

```
[scale 5 circle,
 move -2 -1.5 (rotate 90 circle),
 move 2 -1.5 (rotate 90 circle),
 move 0 2 (x-scale 3 circle)]
```

≡

input 1, version 3

```
[scale 5 circle,
 move -2 -1.5 (rotate
 move 2 -1.5 (rotate 9
 move 0 2 (x-scale 3 c
```

challenge:
when do we stop rewriting?

why e-graphs?

user inputs

```
[scale 5 circle,
 move -2 -1.5 circle,
 move 2 -1.5 circle,
 move 0 2 (x-scale 3 circle)]
```

```
[scale 5 circle,
 move -2 -1.5 (rotate 90 line),
 move 2 -1.5 (rotate 90 (scale 1 (scale 1 (scale 1 (scale 1 line))))),
 move 0 2 (x-scale 3 line)]
```

list of equivalences

```
circle ≡ rotate 90 circle
?x ≡ scale 1 ?x
...
```

≡

input 1, version 2

```
[scale 5 circle,
 move -2 -1.5 (rotate 90 circle),
 move 2 -1.5 (rotate 90 circle),
 move 0 2 (x-scale 3 circle)]
```

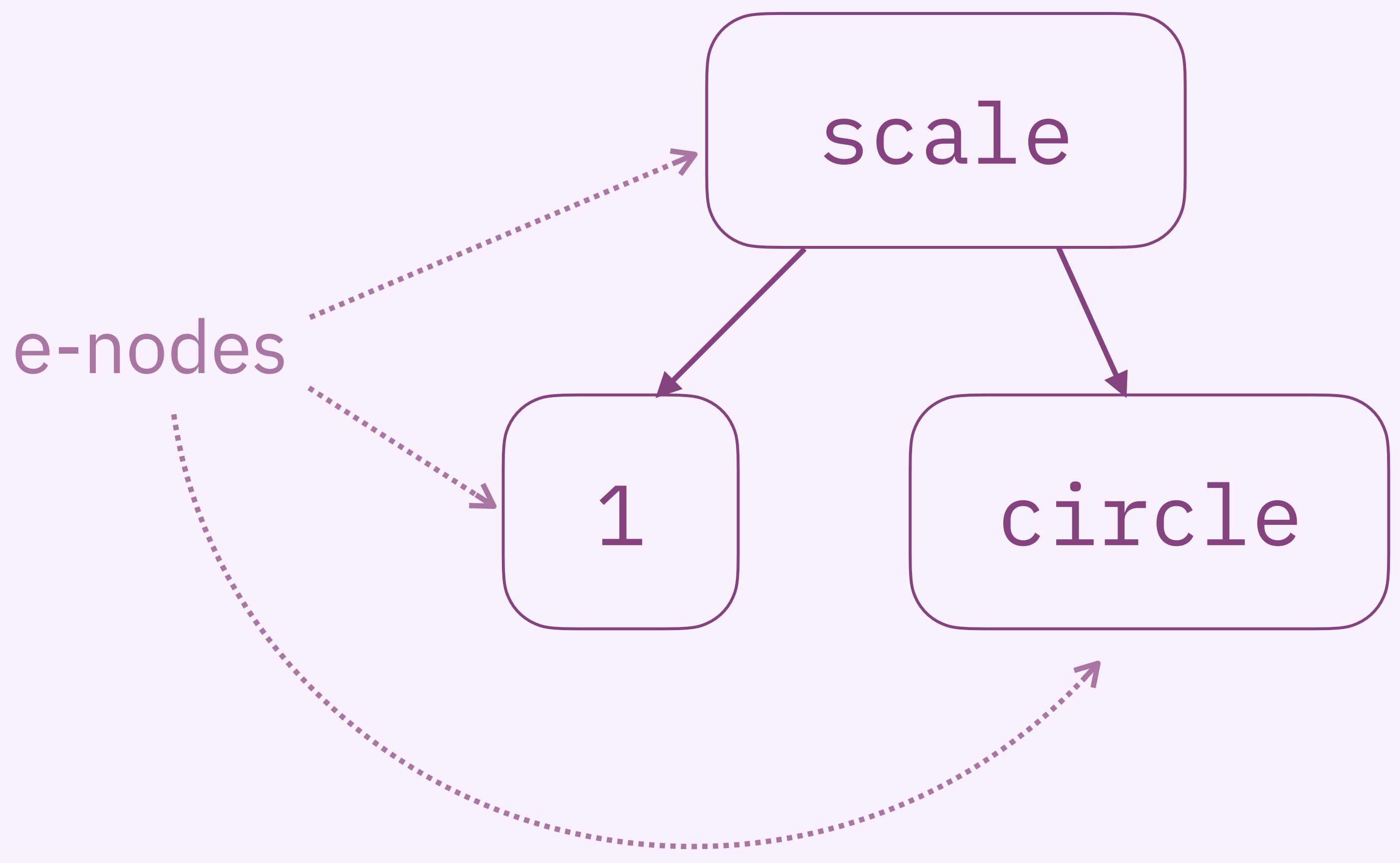
≡

input 1, version 3

```
[scale 5 circle,
 move -2 -1.5 (rotate
 move 2 -1.5 (rotate 9
 move 0 2 (x-scale 3 c
```

challenge:
when do we stop rewriting?

how do **e-graphs** work?

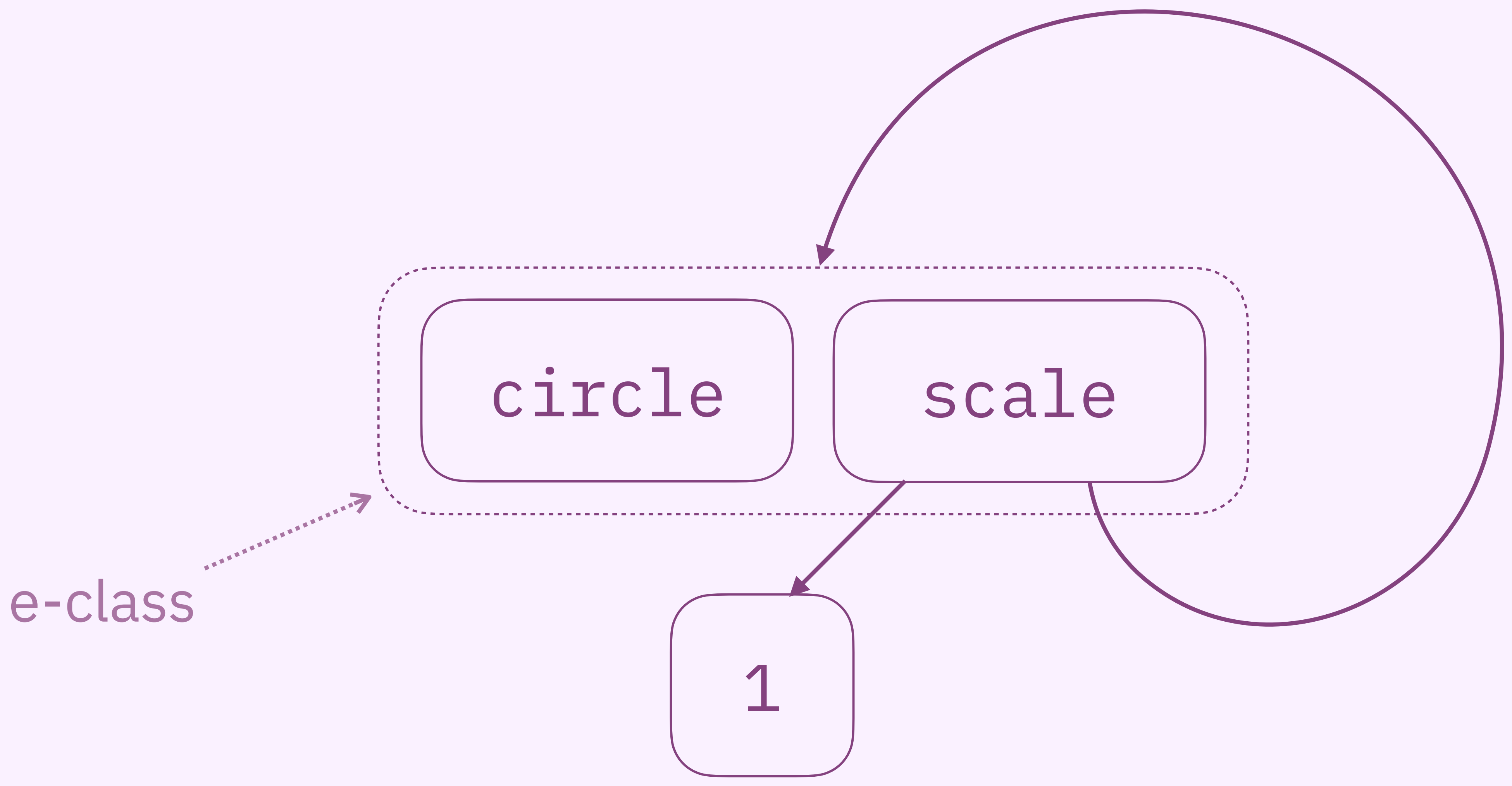


scale 1 circle

e-graphs compactly represent sets of equivalent terms!

[Tate et al. 2009]
[Willsey et al. 2021]

how do **e-graphs** work?



e-graphs compactly represent sets of equivalent terms!

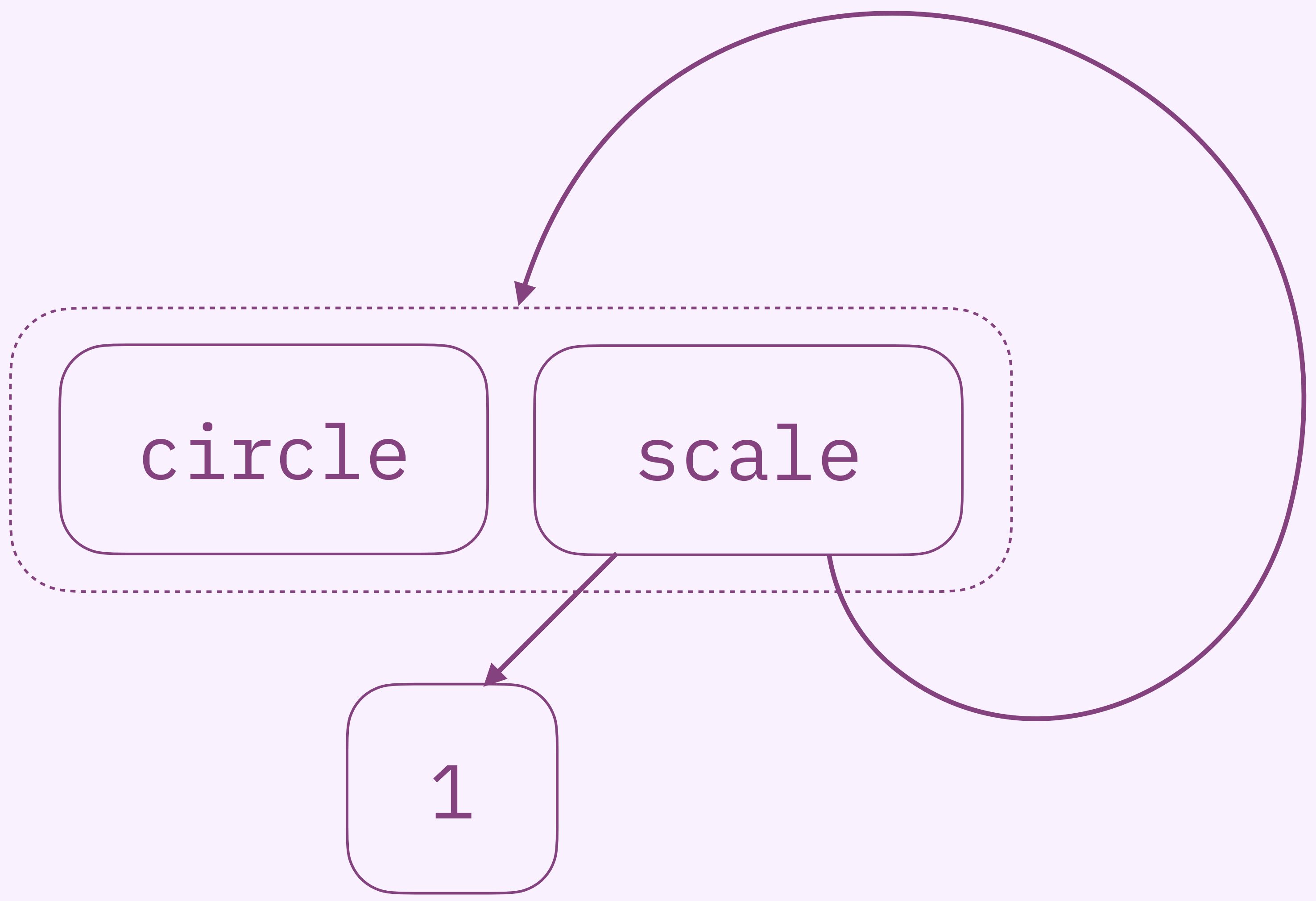
`circle, scale 1 circle, scale 1 (scale 1 circle), ...`

[Tate et al. 2009]
[Willsey et al. 2021]

how do **e-graphs** work?

equivalences

```
circle ≡ rotate 90 circle
?x ≡ scale 1 ?x
...
```

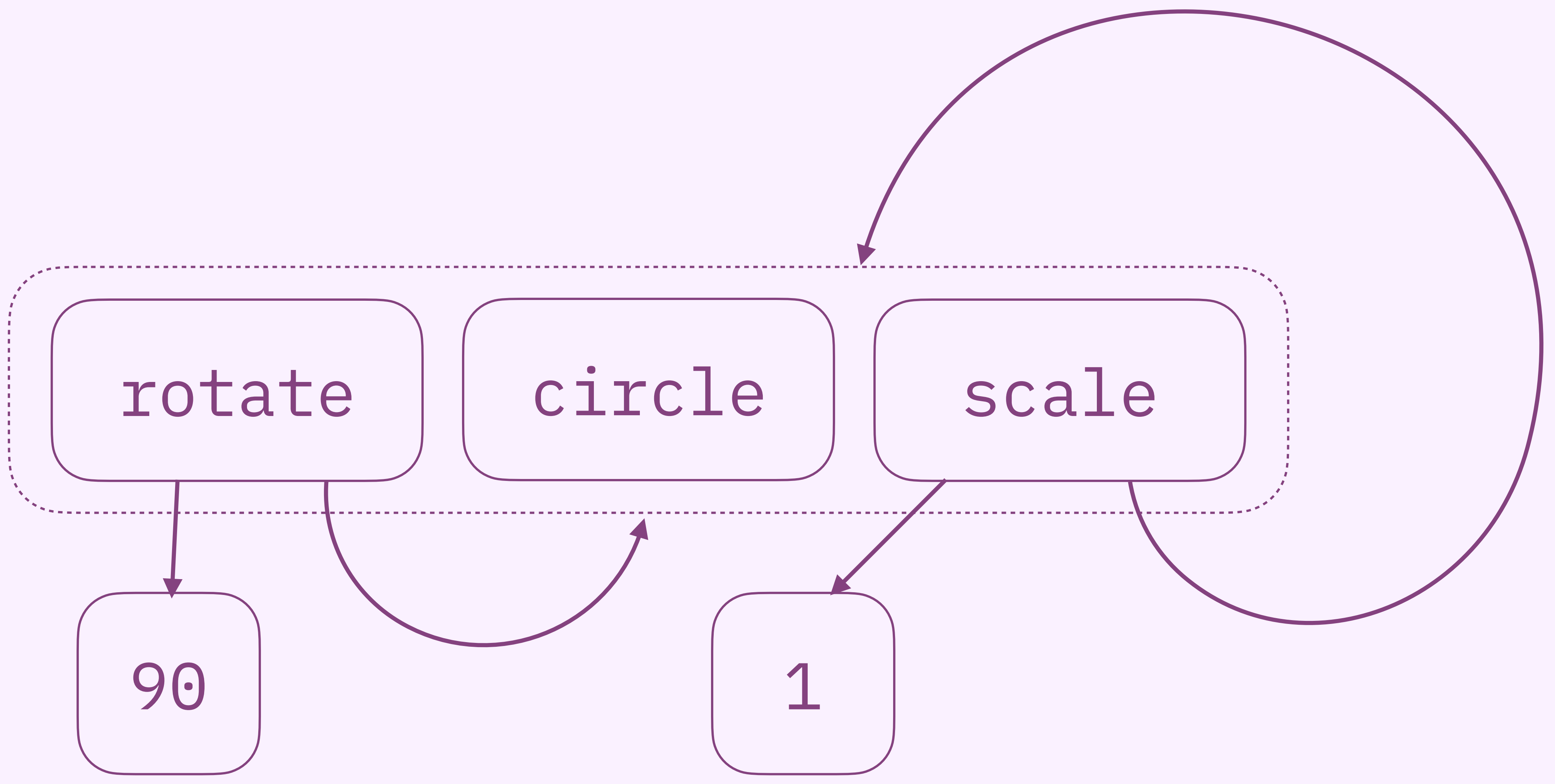


`circle, scale 1 circle, scale 1 (scale 1 circle), ...`

how do **e-graphs** work?

equivalences

```
circle ≡ rotate 90 circle
?x ≡ scale 1 ?x
...
```



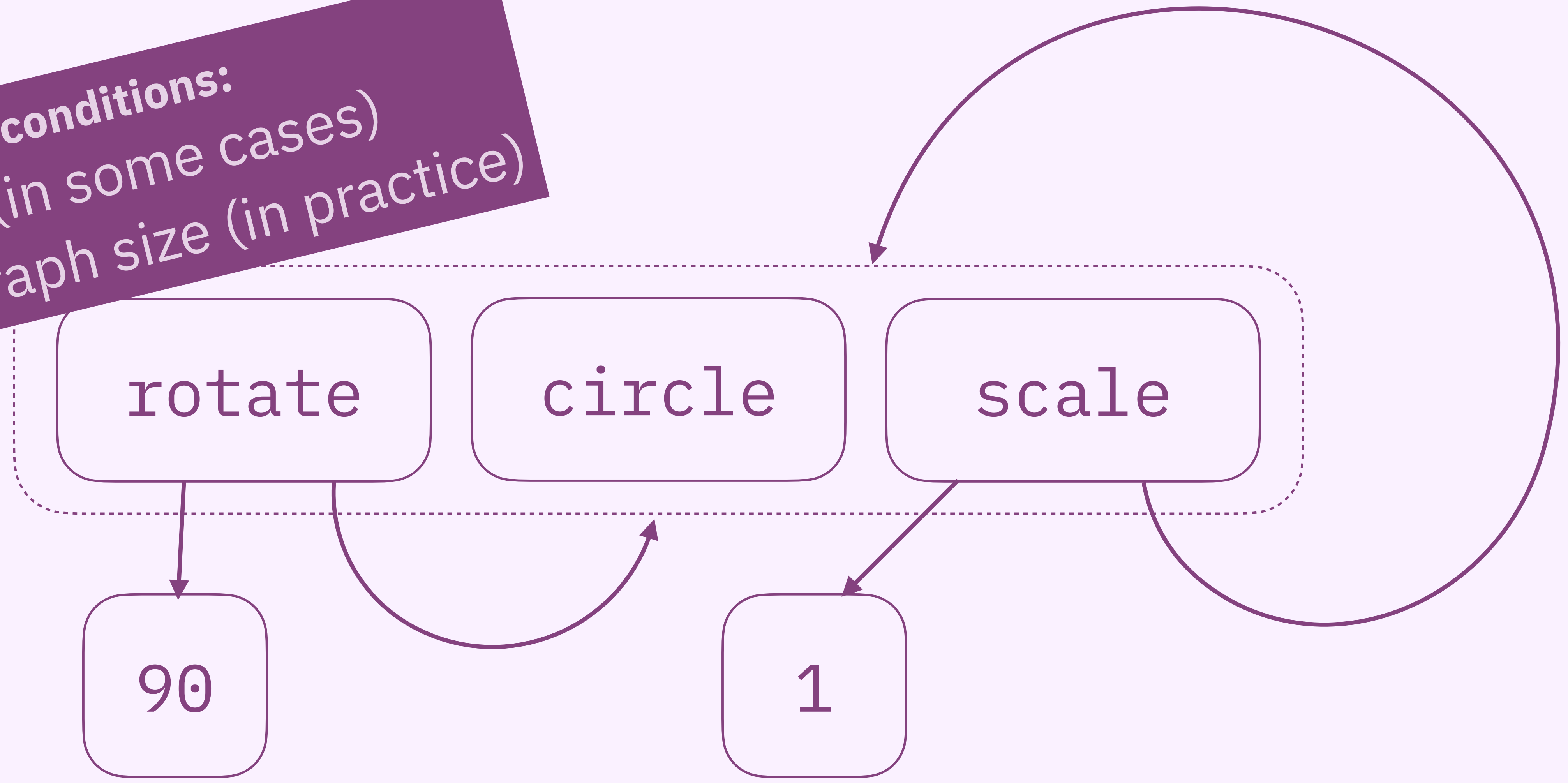
circle, scale 1 circle, rotate 90 circle,
rotate 90 (scale 1 circle), scale 1 (rotate 90 circle), ...

how do **e-graphs** work?

equivalences

```
circle ≡ rotate 90 circle
?x ≡ scale 1 ?x
...
```

stop conditions:
fixpoint (in some cases)
timeout, e-graph size (in practice)



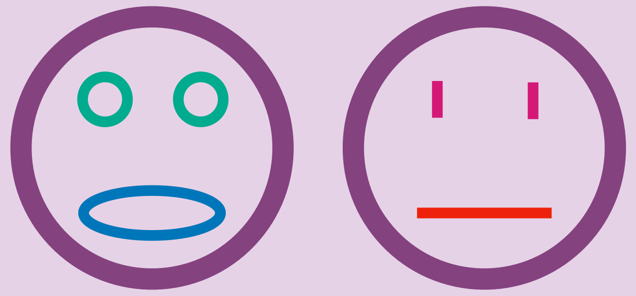

```
circle, scale 1 circle, rotate 90 circle,
rotate 90 (scale 1 circle), scale 1 (rotate 90 circle), ...
```

how do **e-graphs** work?

how does babble work?

user inputs

program corpus



list of equivalences

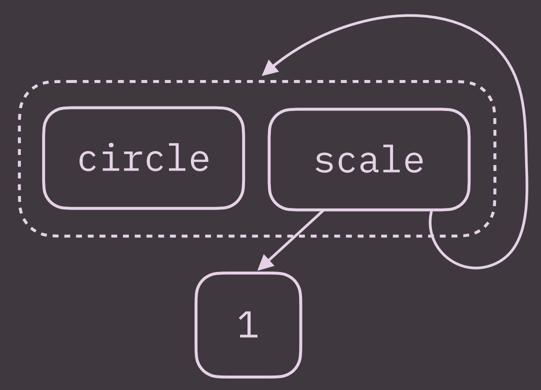
```
circle ≡ rotate 90 circle
?x ≡ scale 1 ?x
...
```

babble

insight

use e-graphs

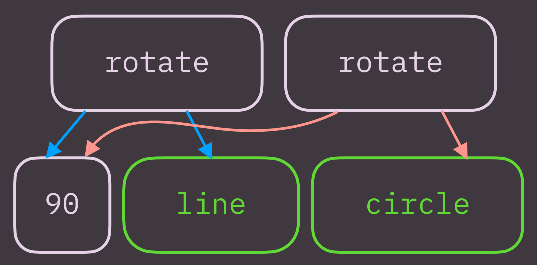
to create and store infinite equivalent input variants



contribution 2

e-graph anti-unification

to propose candidate abstractions in the presence of input variants



rotate 90 ?x

contribution 3

targeted CSE

to pick the best programs and abstractions

```
let f = λx -> … in
 let f = λx -> … in
  let f = λx -> … in
   f 2
```

babble outputs

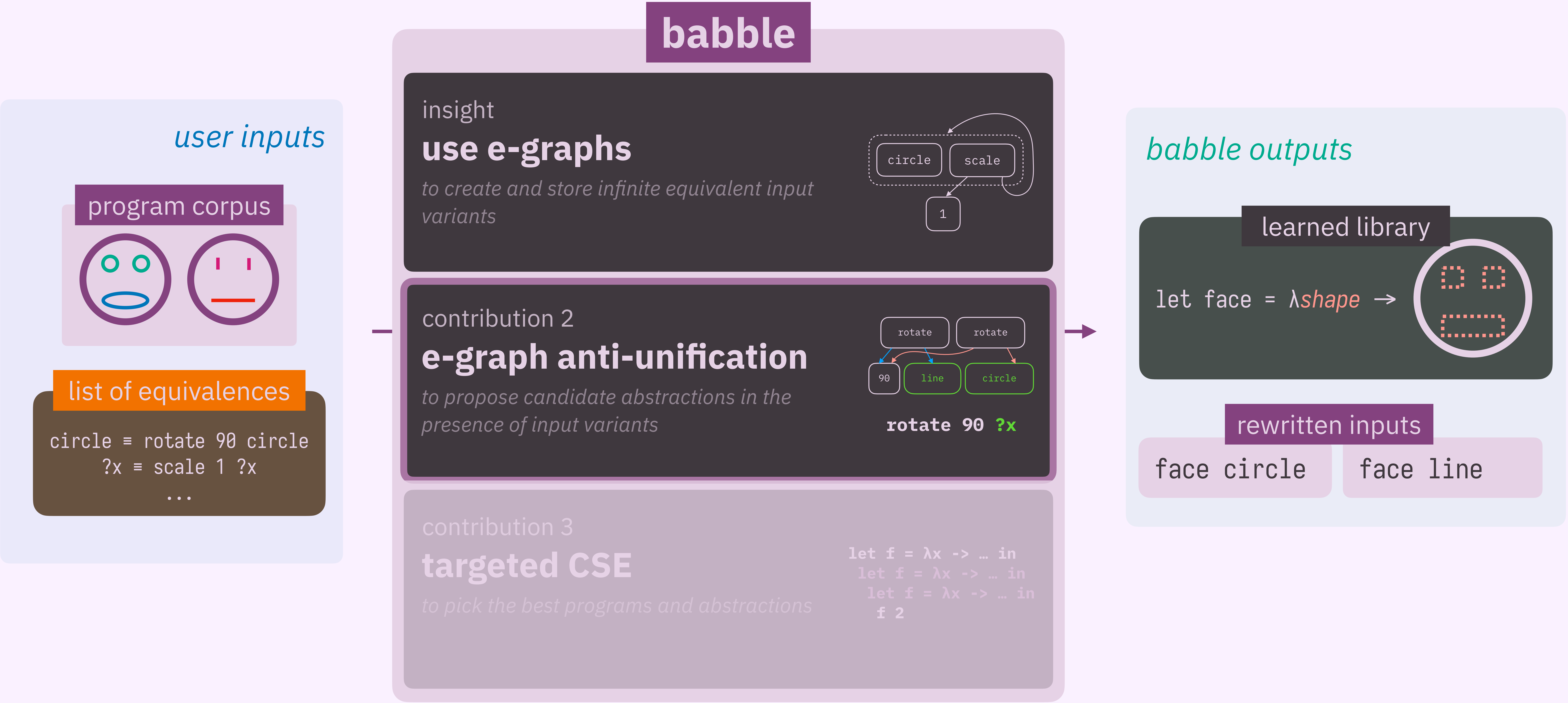
learned library

let face = λ*shape* →

rewritten inputs

`face circle` `face line`

how does babble work?



how does **e-graph anti-unification** work?

how does e-graph anti-unification work?

finding common structure

equivalences

```
circle ≡ rotate 90 circle
?x ≡ scale 1 ?x
...
```

input 1

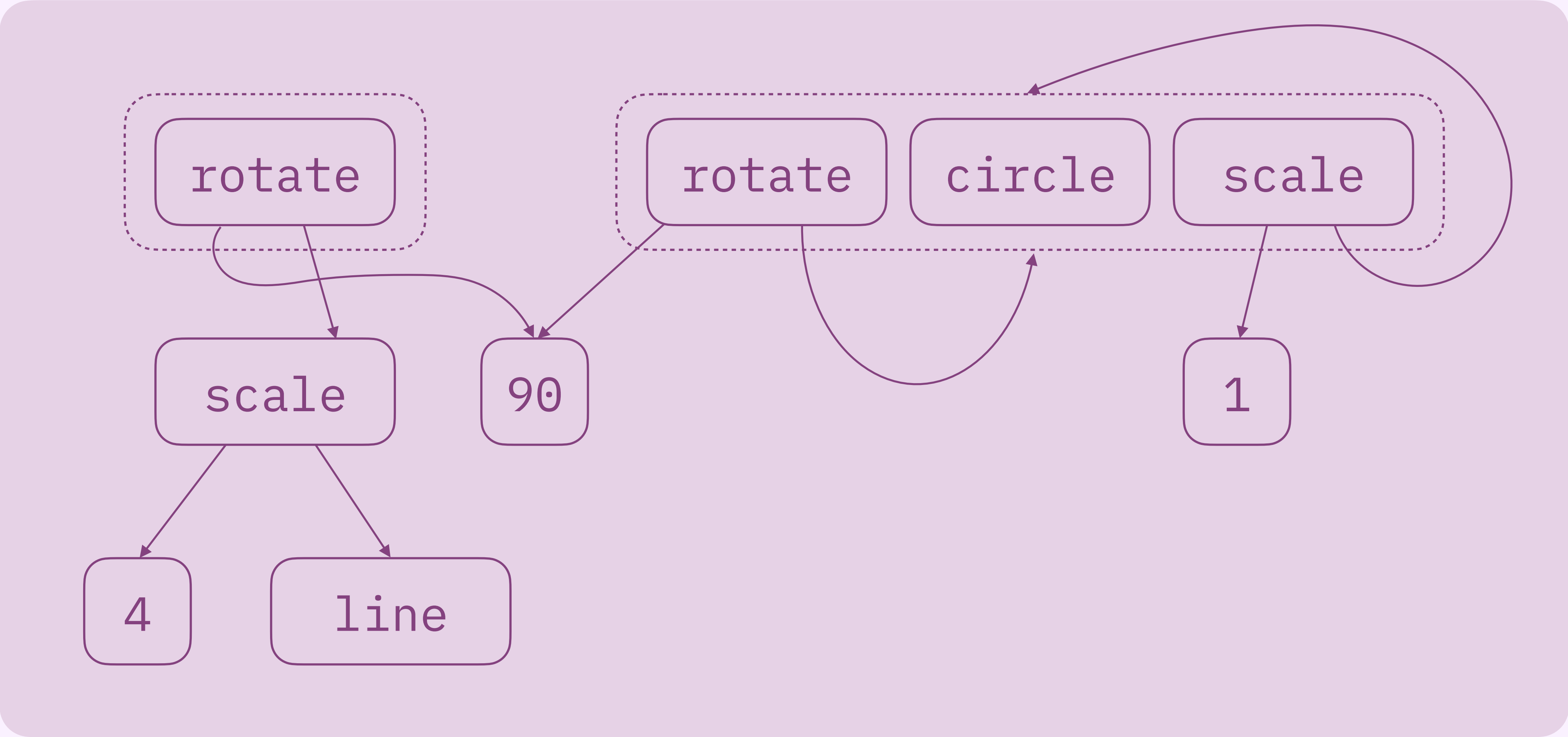
```
rotate 90 (scale 4 line)
```

input 2

```
circle
```

i.e. scale 1 circle
rotate 90 circle
scale 1 (rotate 90 circle)
rotate 90 (scale 1 circle)

etc.



how does e-graph anti-unification work?

finding common structure

equivalences

```
circle ≡ rotate 90 circle
?x ≡ scale 1 ?x
...
```

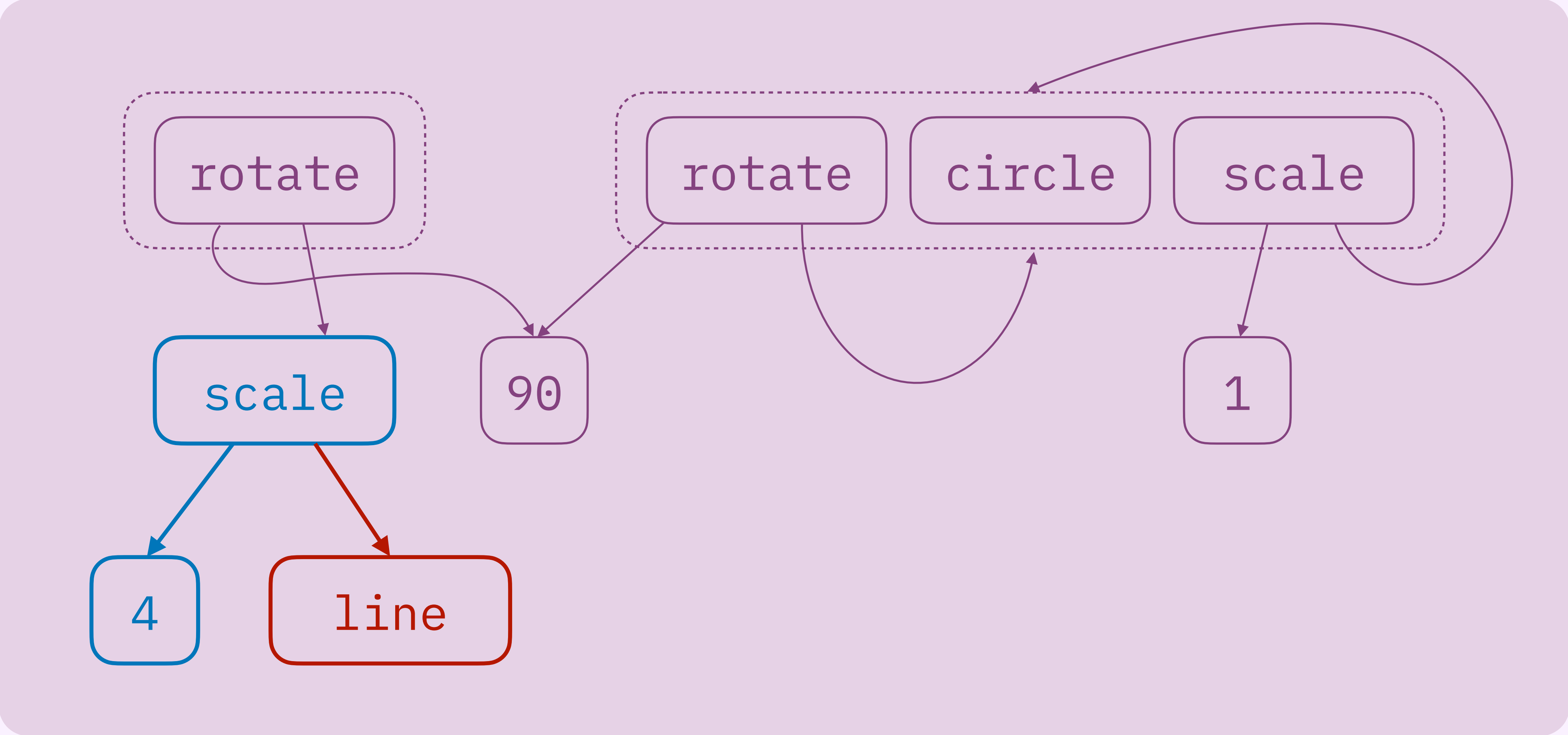
input 1

```
rotate 90 (scale 4 line)
```

input 2

```
circle
i.e. scale 1 circle
     rotate 90 circle
     scale 1 (rotate 90 circle)
     rotate 90 (scale 1 circle)

     etc.
```



criterion 1. **occurs multiple times** `scale 4 ?x` won't work

how does e-graph anti-unification work?

finding common structure

equivalences

```
circle ≡ rotate 90 circle
?x ≡ scale 1 ?x
...
```

input 1

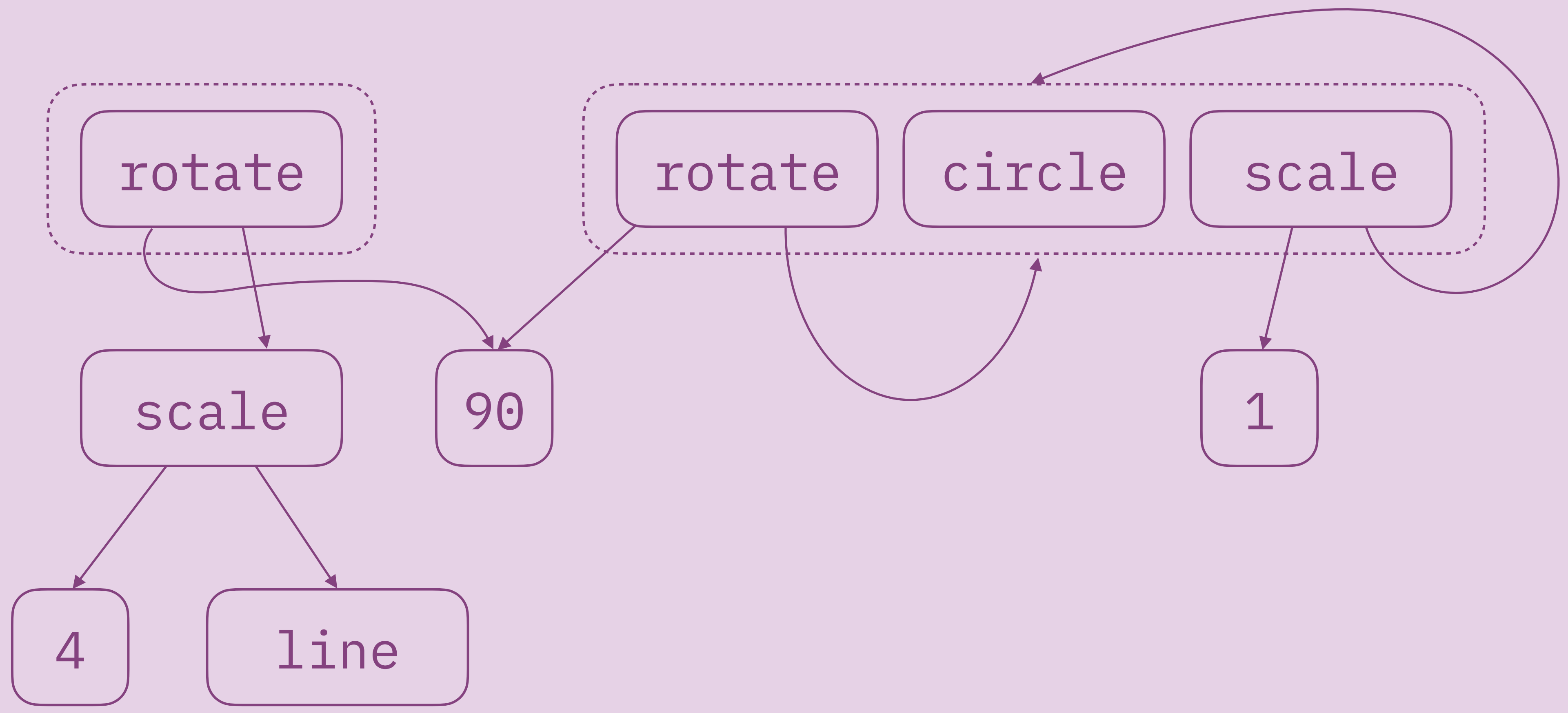
```
rotate 90 (scale 4 line)
```

input 2

```
circle
```

i.e. `scale 1 circle`
`rotate 90 circle`
`scale 1 (rotate 90 circle)`
`rotate 90 (scale 1 circle)`

etc.



criterion 1. **occurs multiple times** `scale 4 ?x` won't work

how does e-graph anti-unification work?

finding common structure

equivalences

```
circle ≡ rotate 90 circle
?x ≡ scale 1 ?x
...
```

input 1

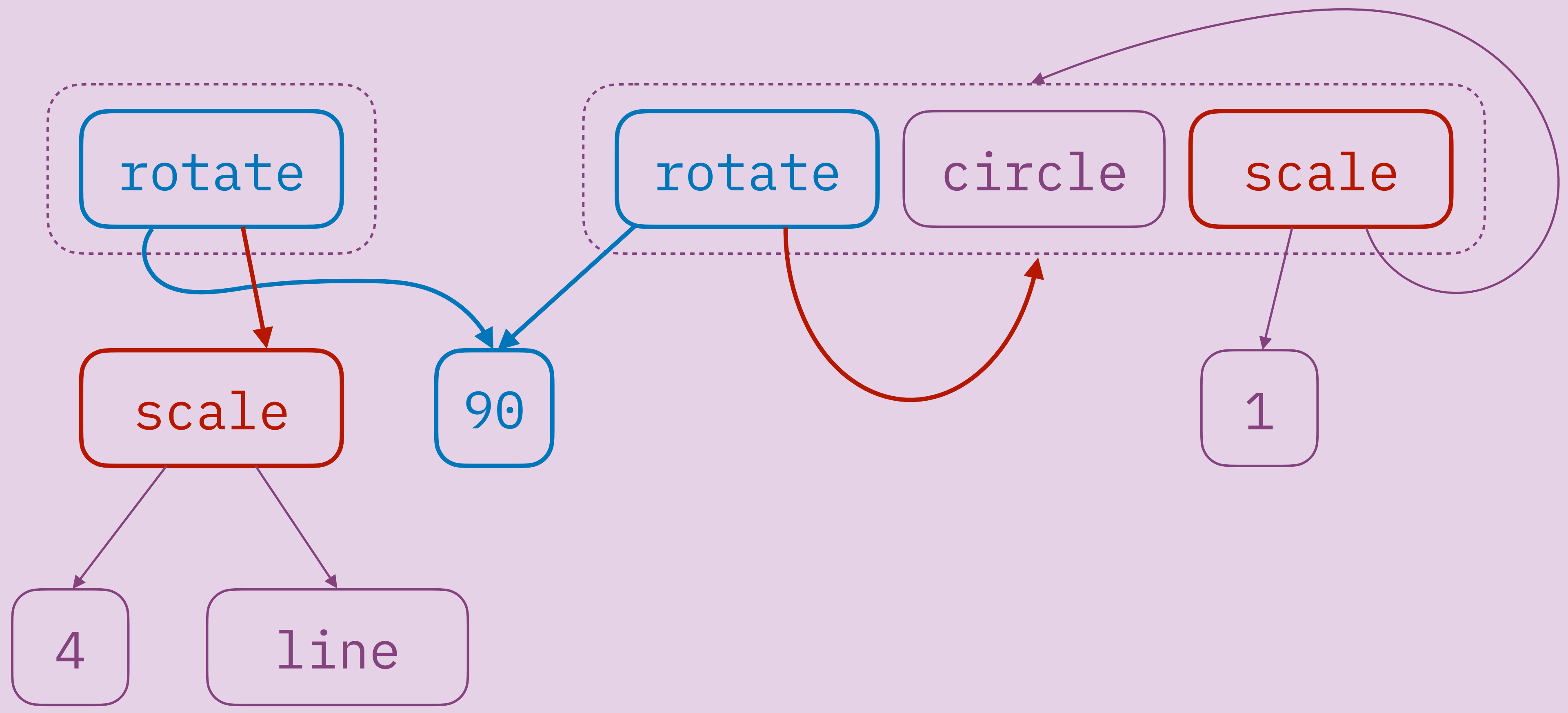
`rotate 90 (scale 4 line)`

input 2

`circle`

i.e. `scale 1 circle`
`rotate 90 circle`
`scale 1 (rotate 90 circle)`
`rotate 90 (scale 1 circle)`

etc.



criterion 1. **occurs multiple times** `scale 4 ?x` `won't work`

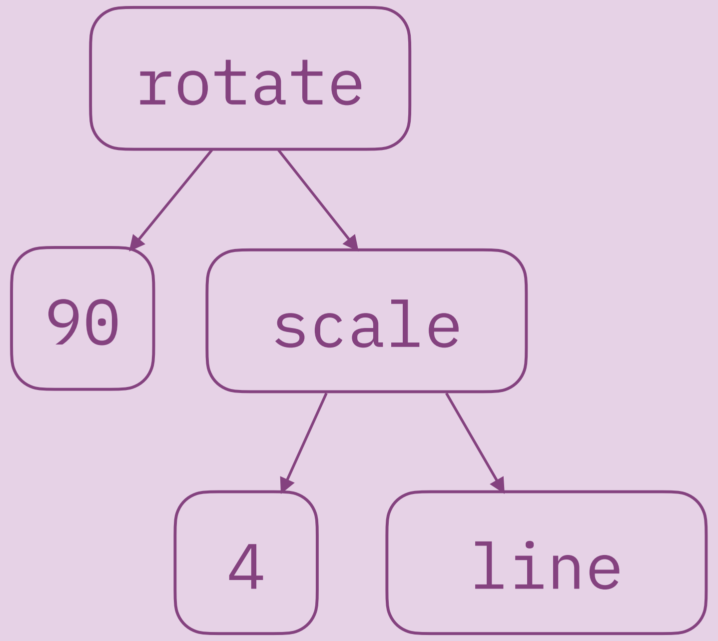
criterion 2. **prefer specific abstractions** `rotate 90 ?x` `works, but we can do better`

how does e-graph anti-unification work?

term anti-unification

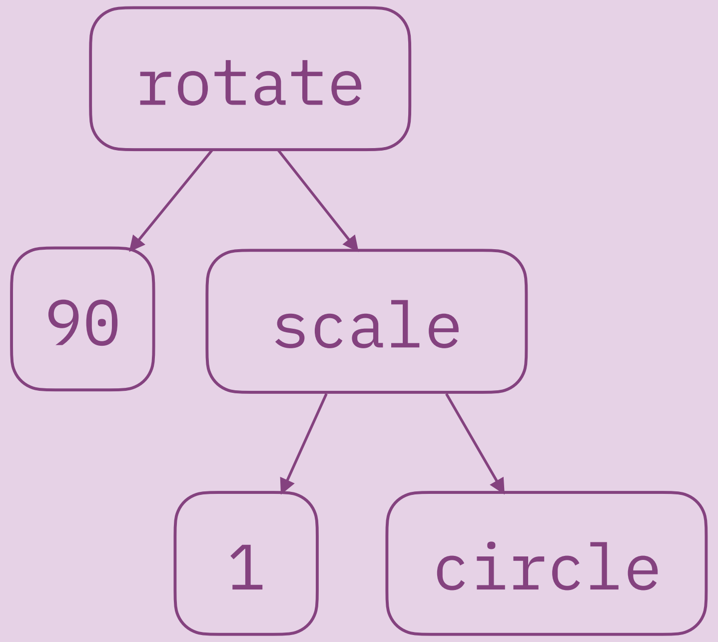
input 1

`rotate 90 (scale 4 line)`



input 2

`rotate 90 (scale 1 circle)`



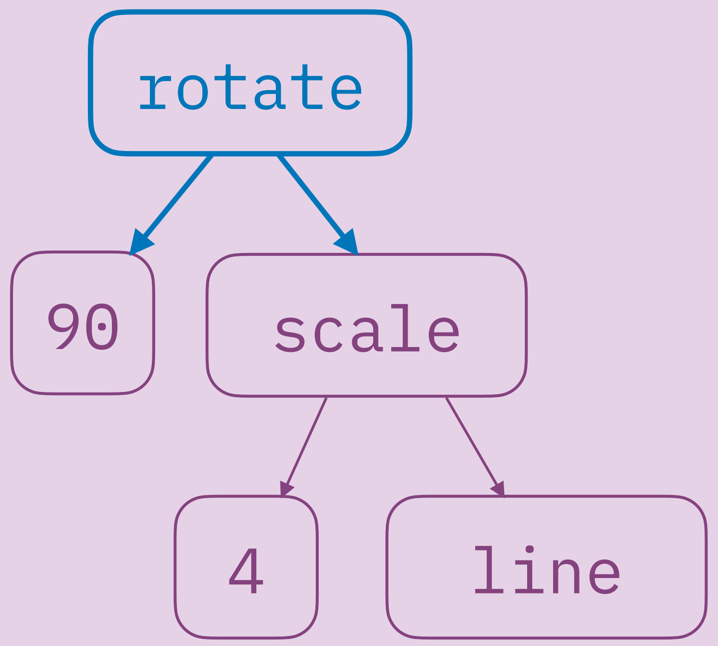
A top-down approach to finding common structure. *(prior work!)*

how does e-graph anti-unification work?

term anti-unification

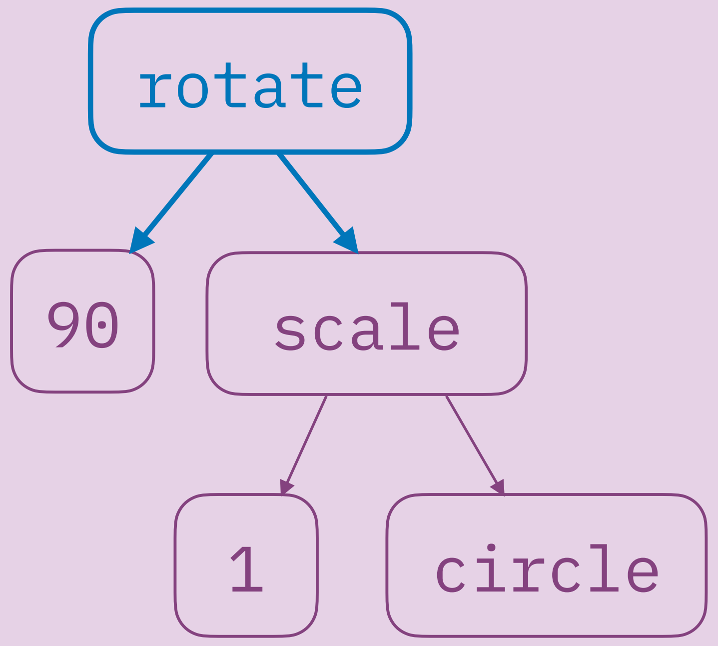
input 1

`rotate 90 (scale 4 line)`



input 2

`rotate 90 (scale 1 circle)`



(starting from the root of both terms)

option 1 **if nodes are same, add to pattern & recurse**

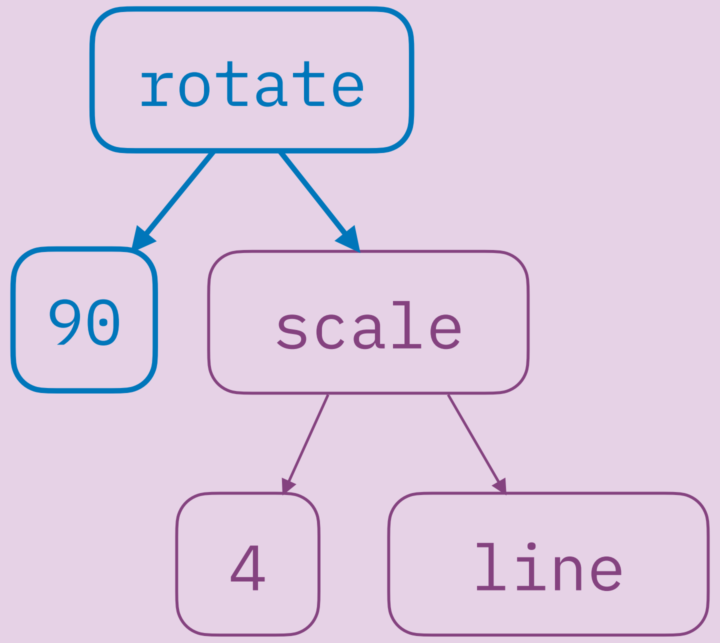
current pattern

`rotate`

term anti-unification

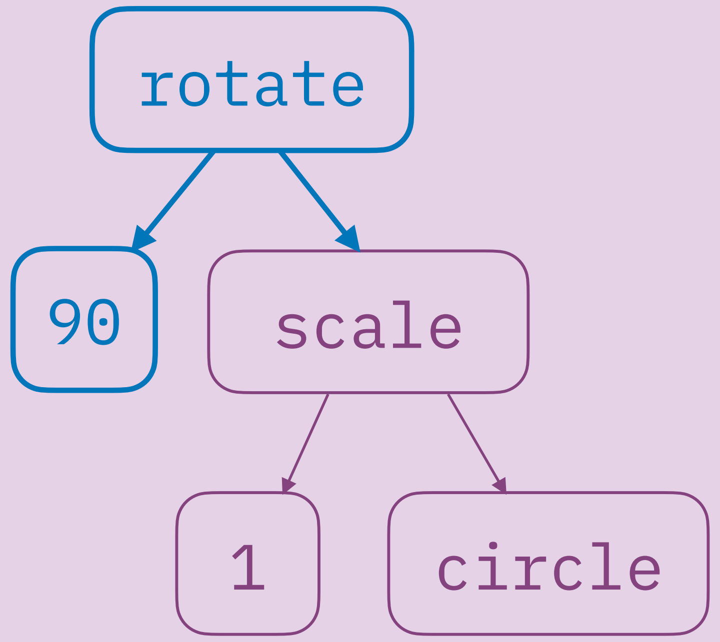
input 1

`rotate 90 (scale 4 line)`



input 2

`rotate 90 (scale 1 circle)`



(starting from the root of both terms)

option 1 **if nodes are same, add to pattern & recurse**

current pattern

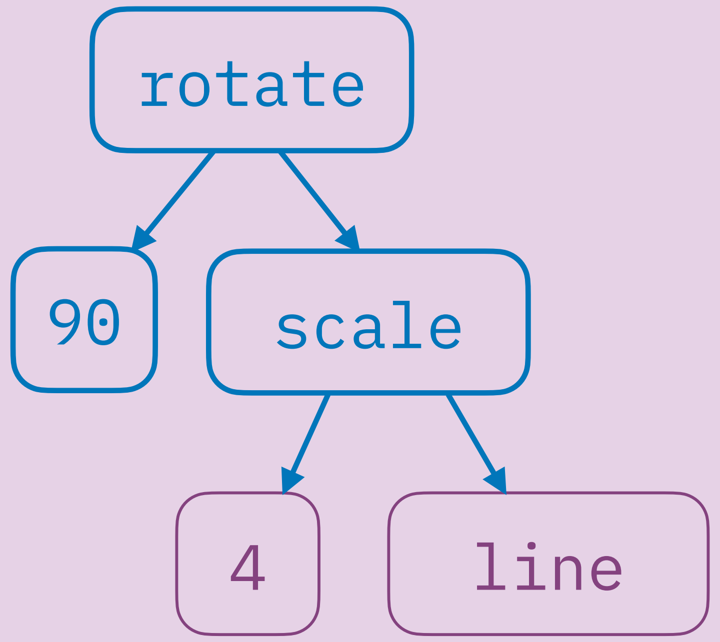
`rotate 90`

how does e-graph anti-unification work?

term anti-unification

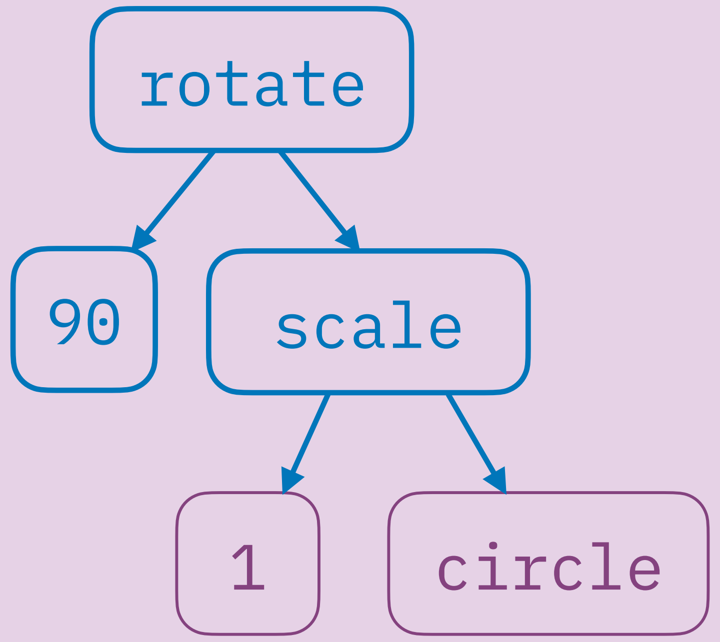
input 1

`rotate 90 (scale 4 line)`



input 2

`rotate 90 (scale 1 circle)`



(starting from the root of both terms)

option 1 **if nodes are same, add to pattern & recurse**

current pattern

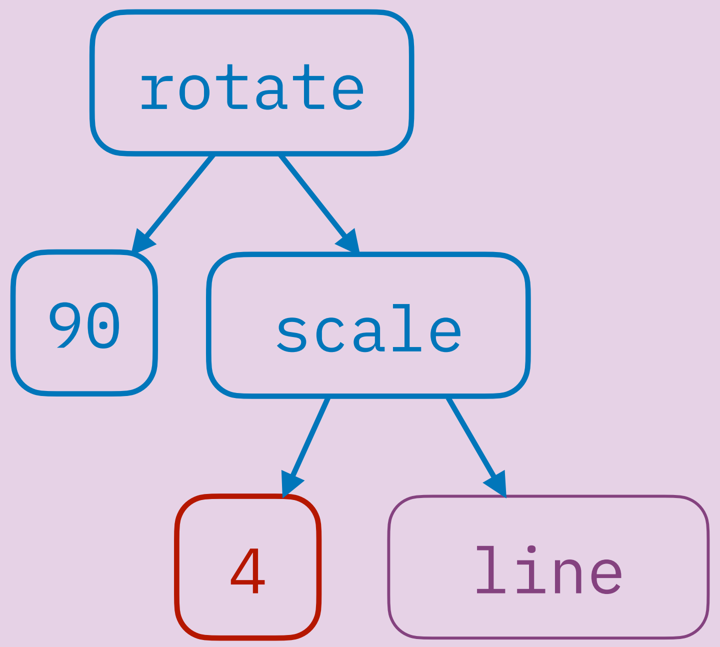
`rotate 90 (scale      )`

how does e-graph anti-unification work?

term anti-unification

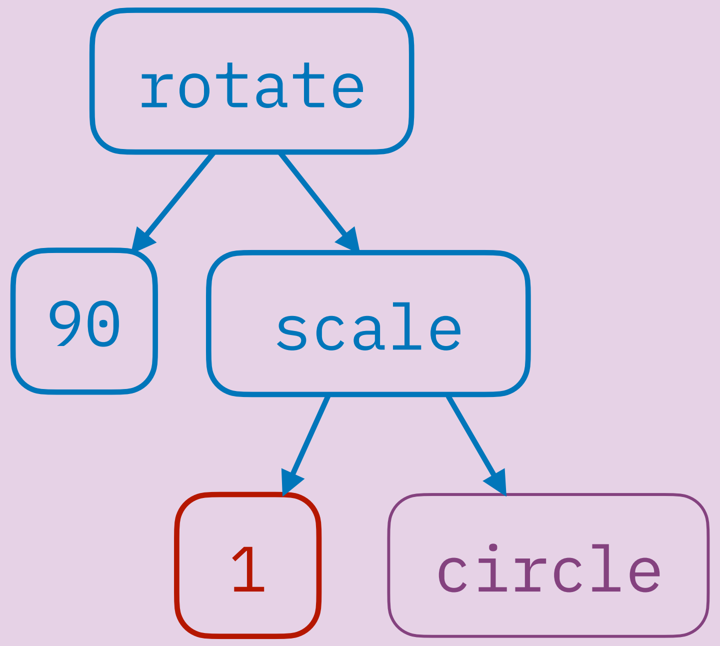
input 1

`rotate 90 (scale 4 line)`



input 2

`rotate 90 (scale 1 circle)`



(starting from the root of both terms)

option 1 **if nodes are same, add to pattern & recurse**

option 2 **if nodes differ, insert hole**

current pattern

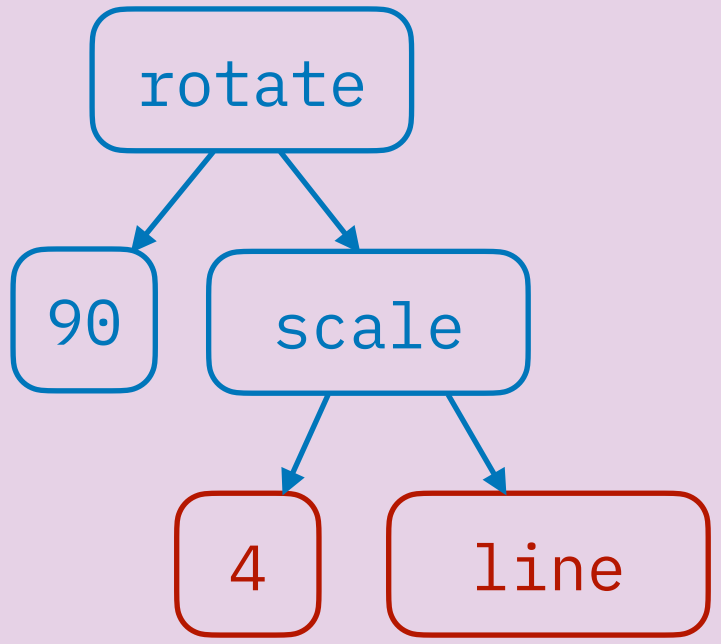
`rotate 90 (scale ?x )`

how does e-graph anti-unification work?

term anti-unification

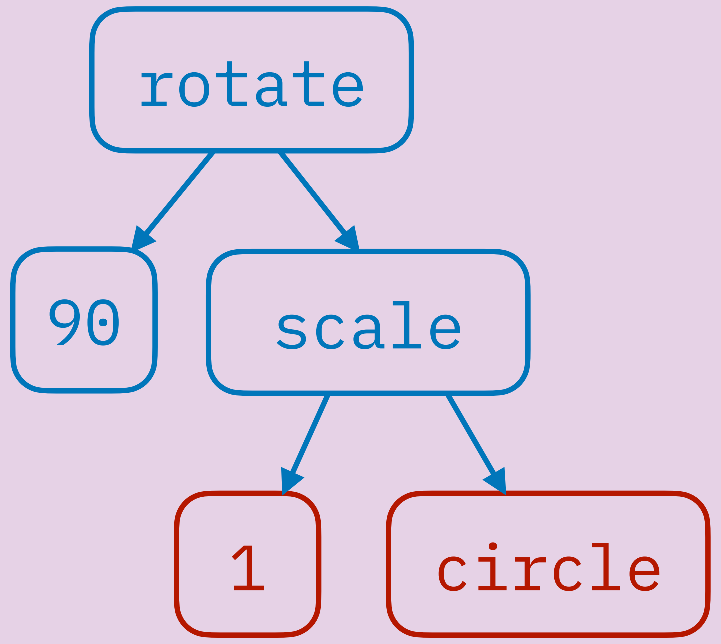
input 1

`rotate 90 (scale 4 line)`



input 2

`rotate 90 (scale 1 circle)`



(starting from the root of both terms)

option 1 **if nodes are same, add to pattern & recurse**

option 2 **if nodes differ, insert hole**

current pattern

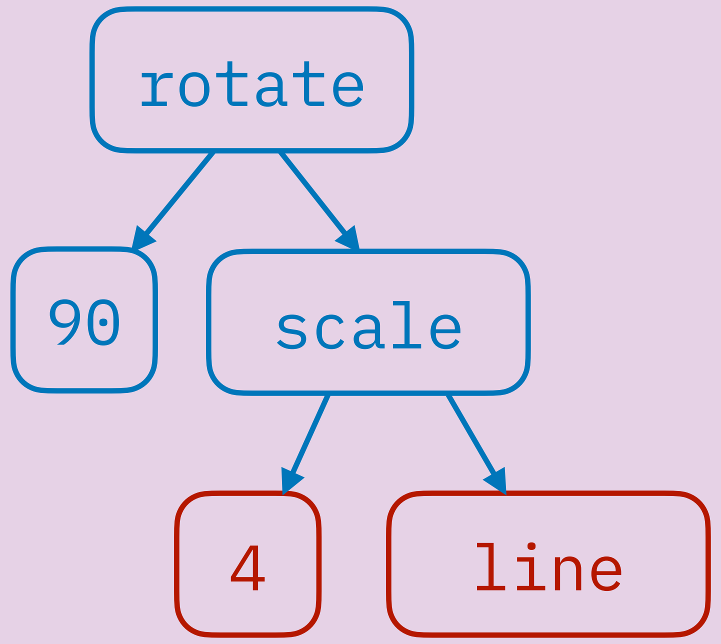
`rotate 90 (scale ?x ?y)`

how does e-graph anti-unification work?

term anti-unification

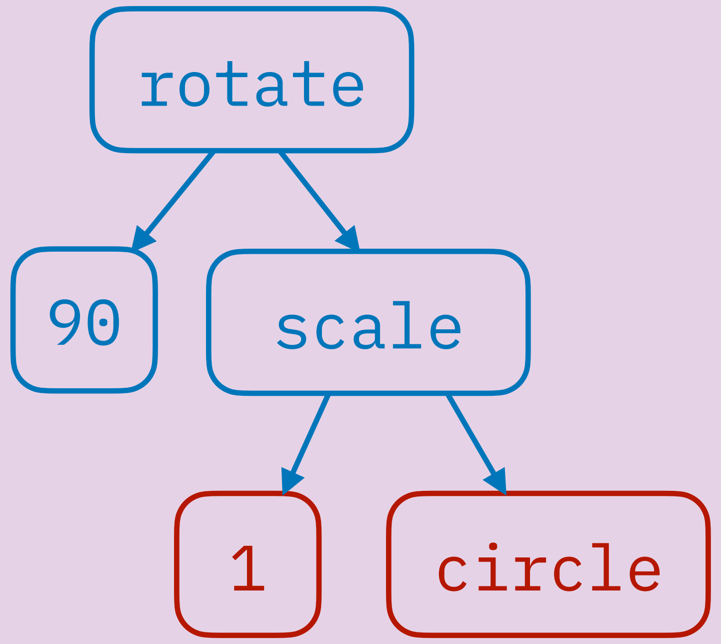
input 1

`rotate 90 (scale 4 line)`



input 2

`rotate 90 (scale 1 circle)`



(starting from the root of both terms)

option 1 **if nodes are same, add to pattern & recurse**

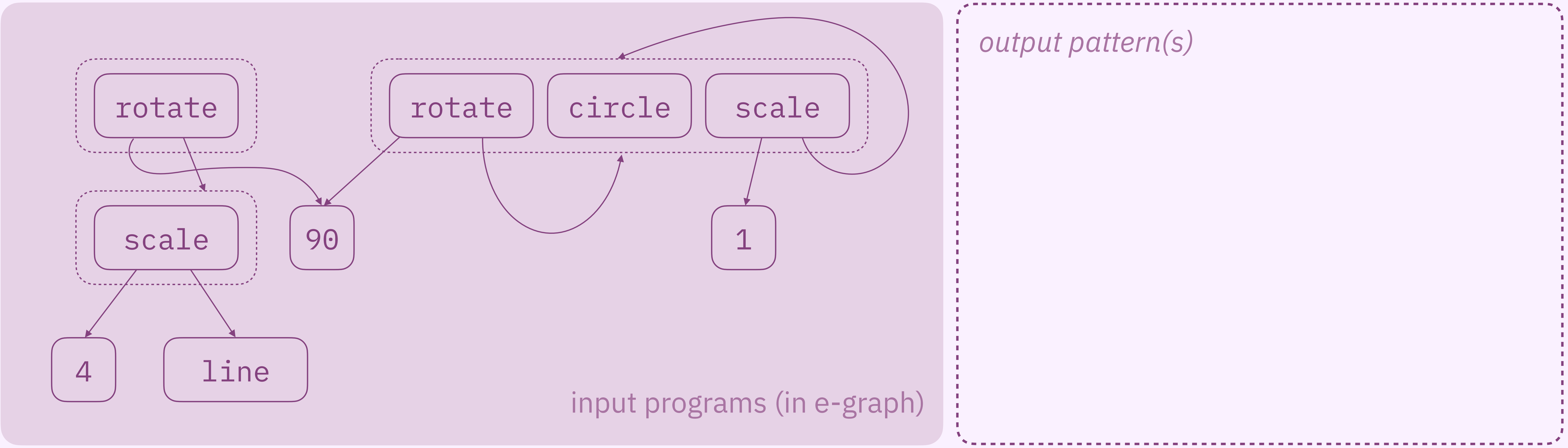
option 2 **if nodes differ, insert hole**

challenge:
how to apply this to e-graphs?

current pattern

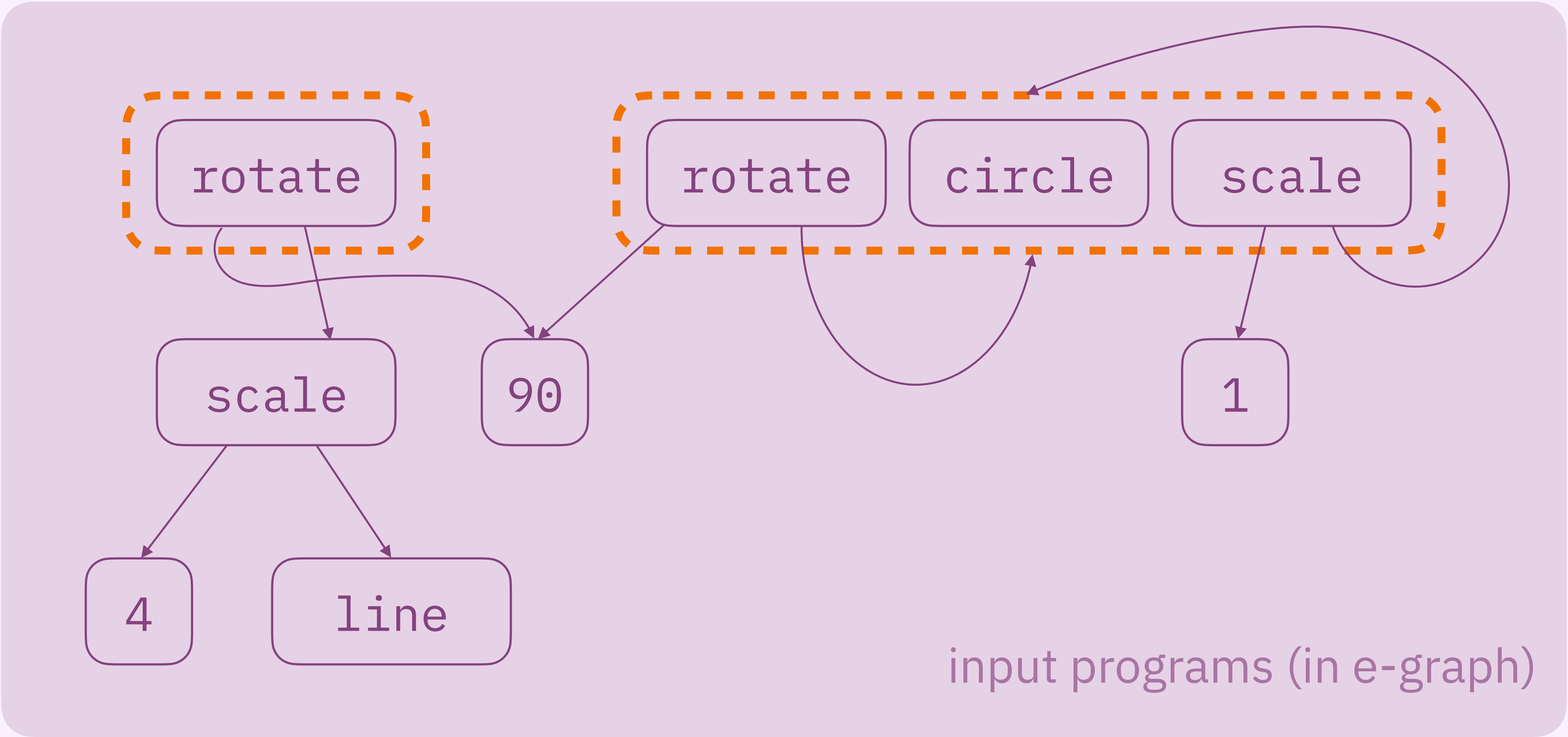
`rotate 90 (scale ?x ?y)`

how does **e-graph anti-unification** work?



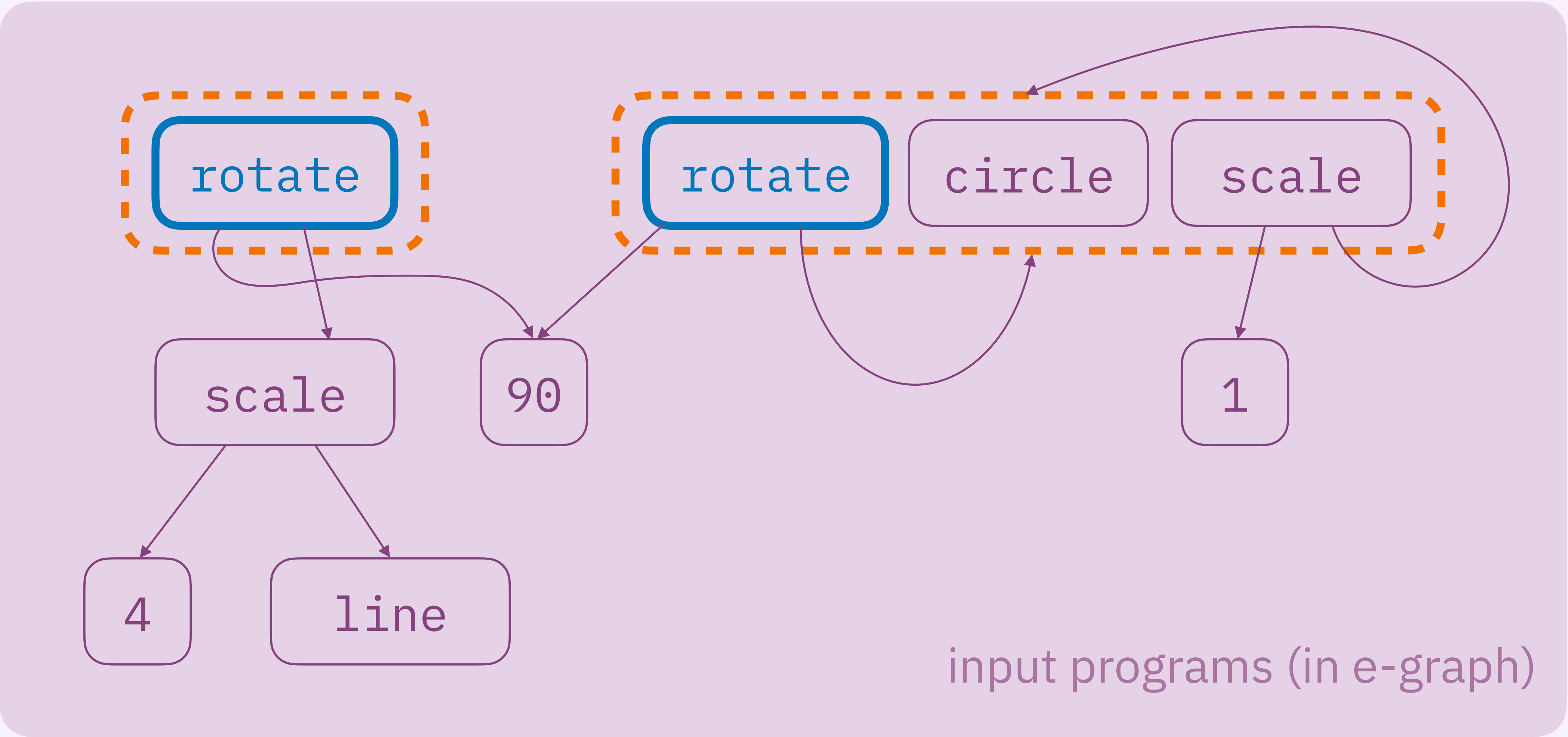
(the intuition behind)
A top-down approach to finding common structure
in the presence of e-graphs.

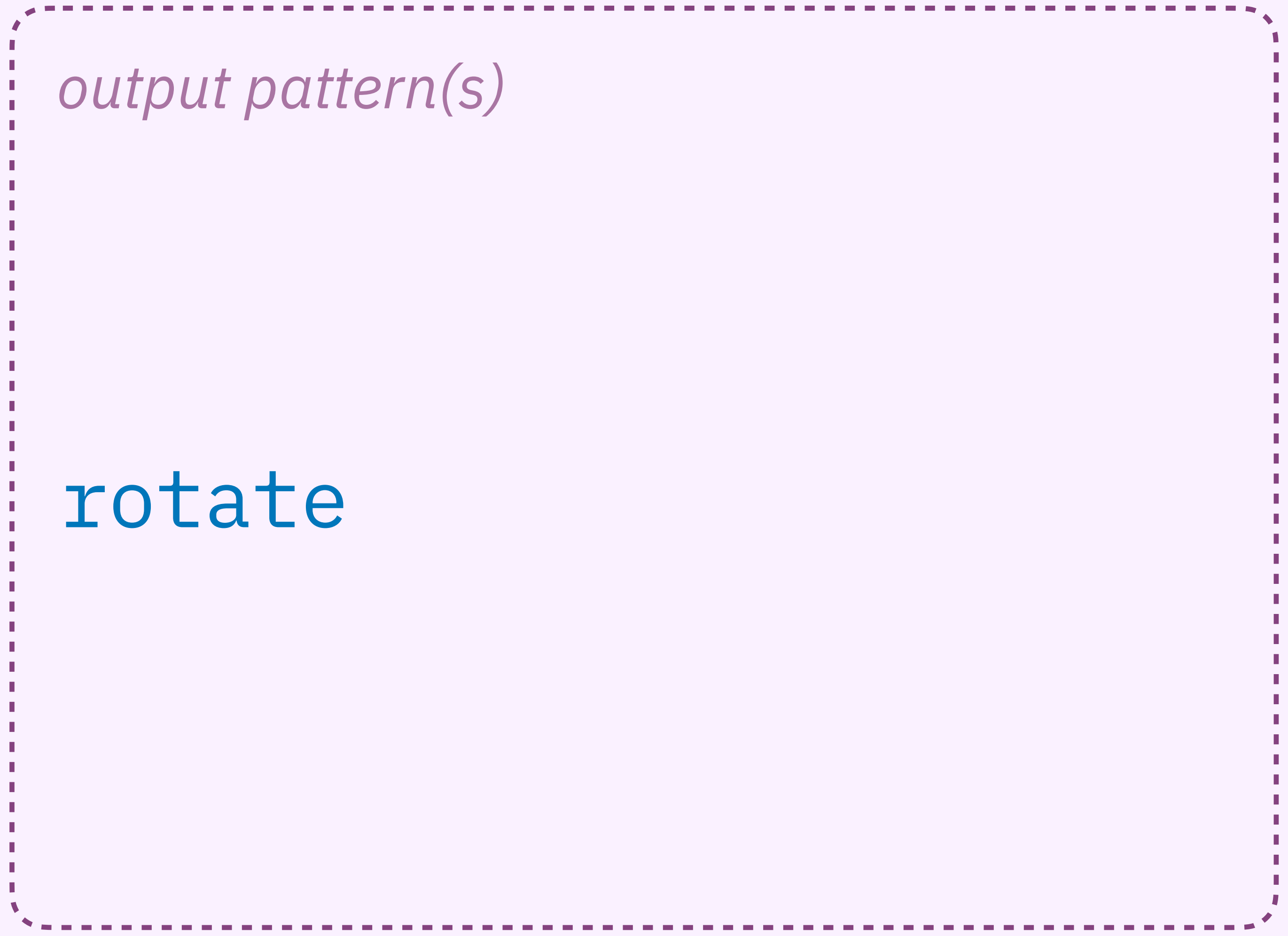
how does **e-graph anti-unification** work?



step 1 **pick two e-classes**

how does **e-graph anti-unification** work?

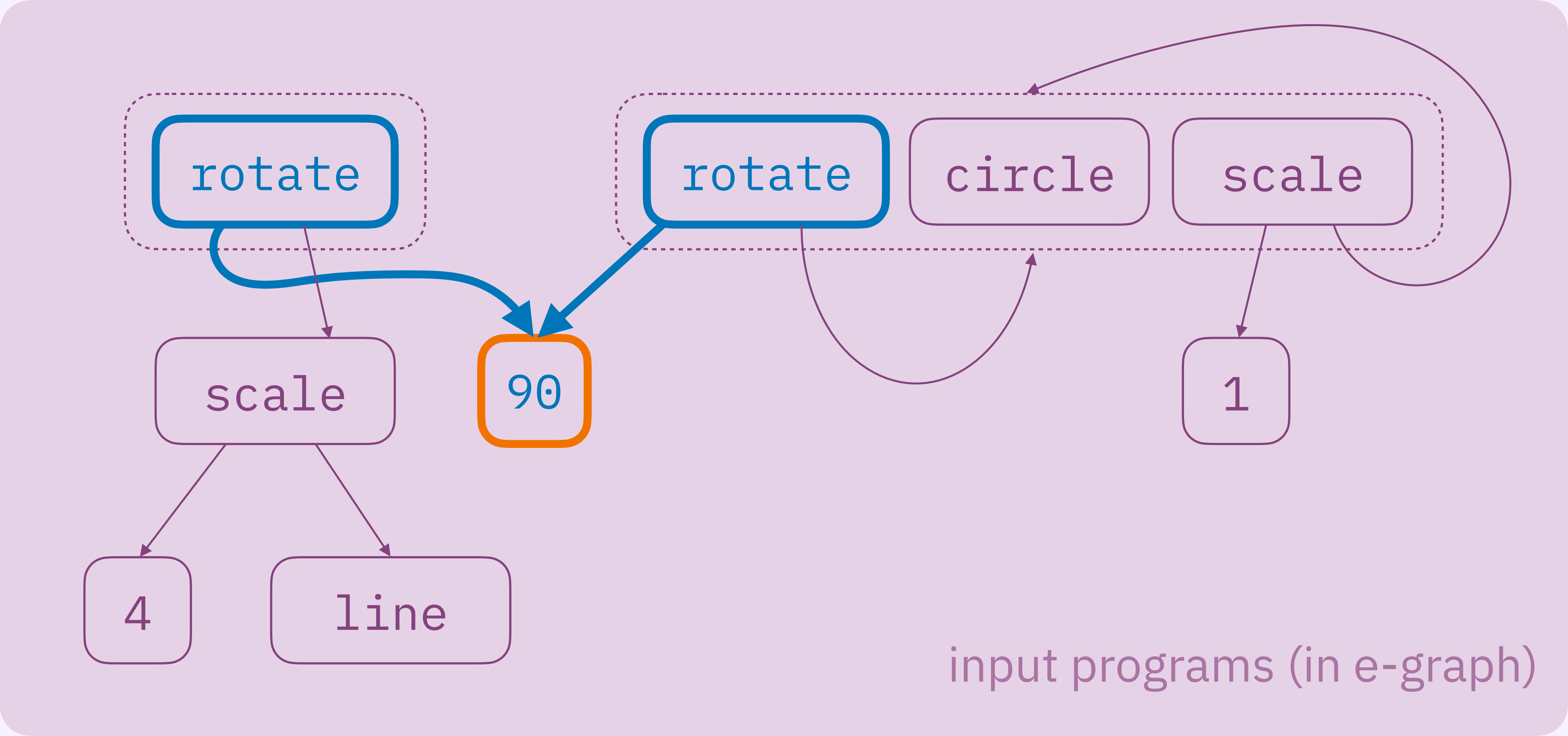


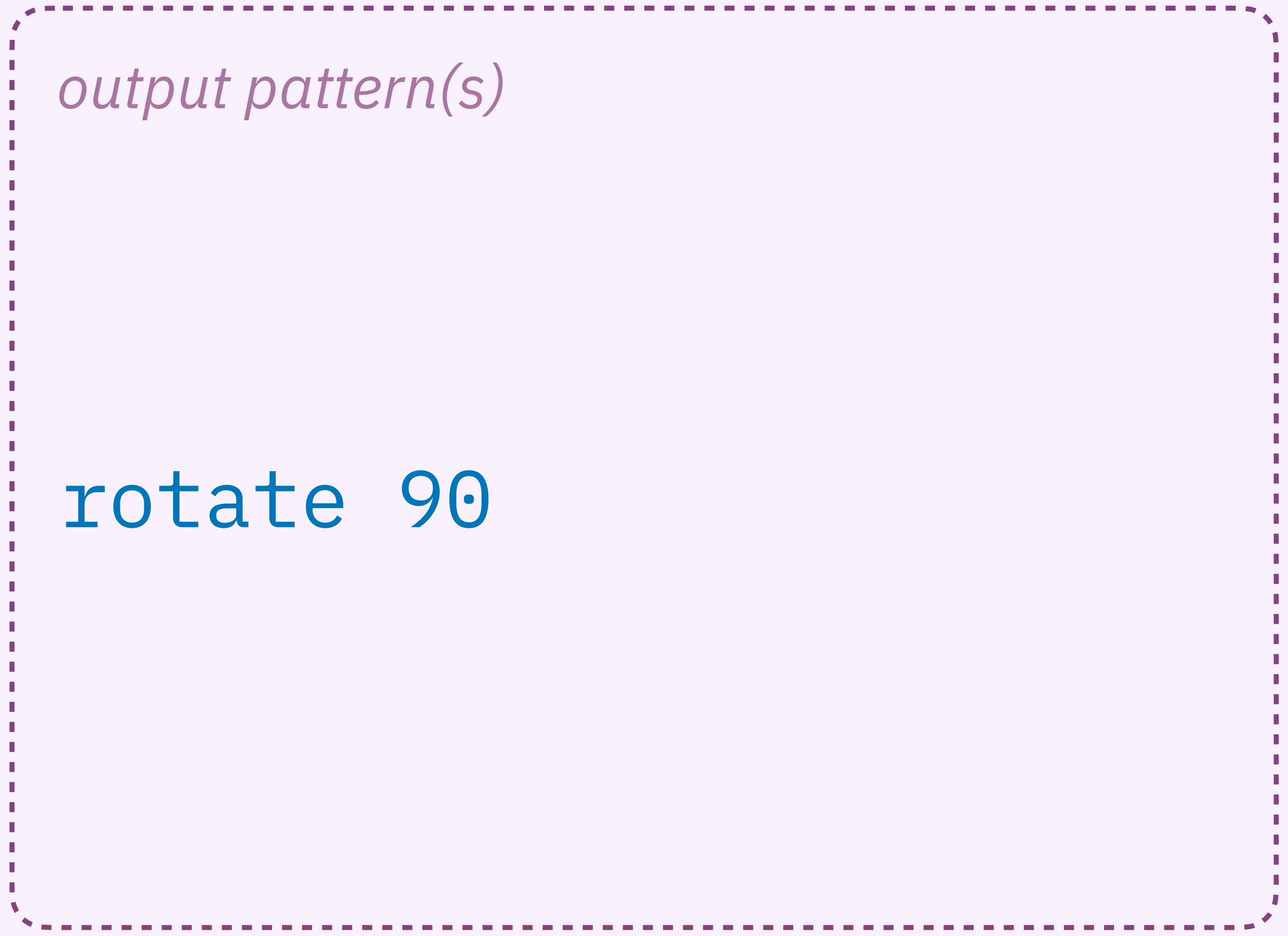


step 1 **pick two e-classes**

step 2a **if e-classes contain matching e-nodes, for each pair of matching e-nodes, add to pattern & run step 2 with matching e-nodes' children**

how does **e-graph anti-unification** work?

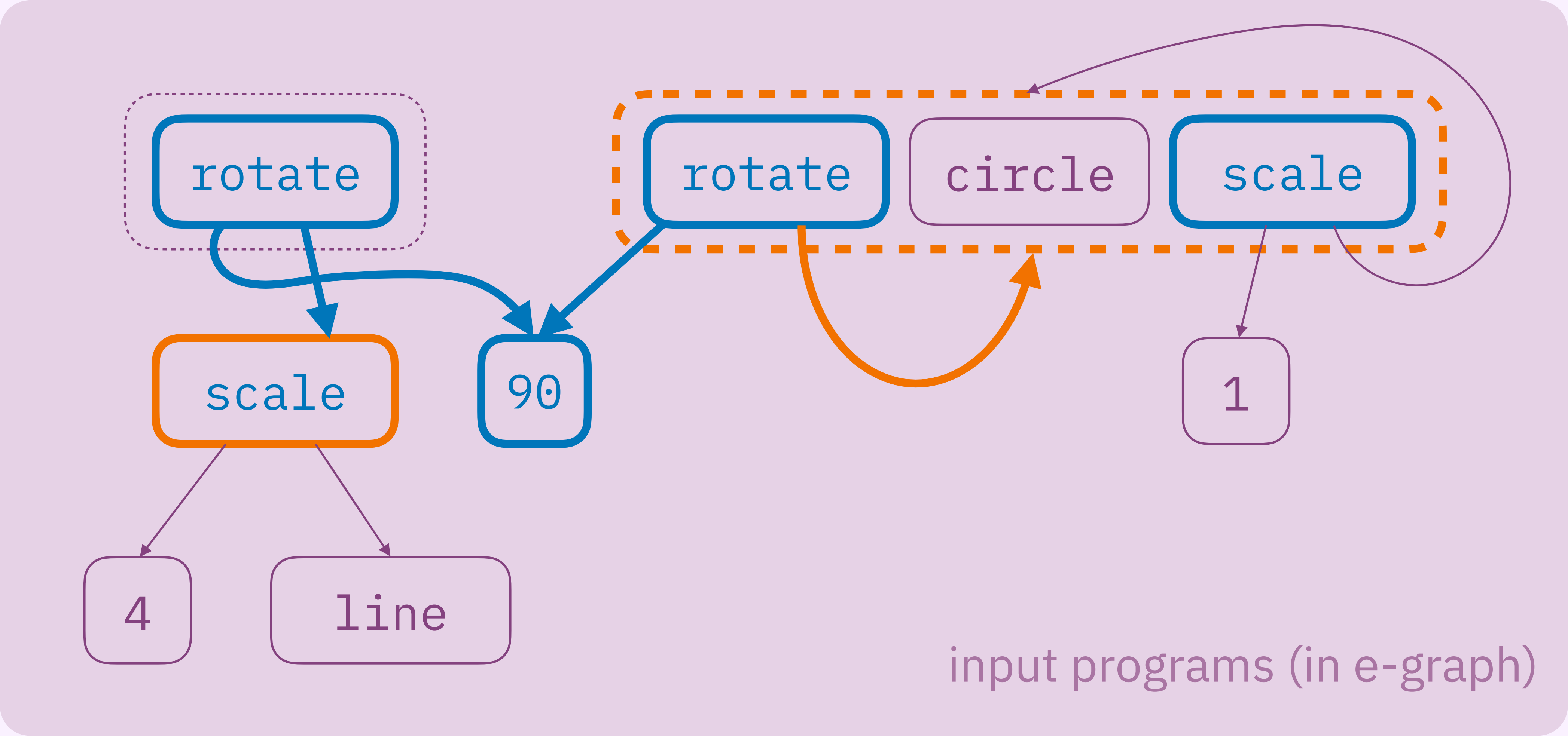


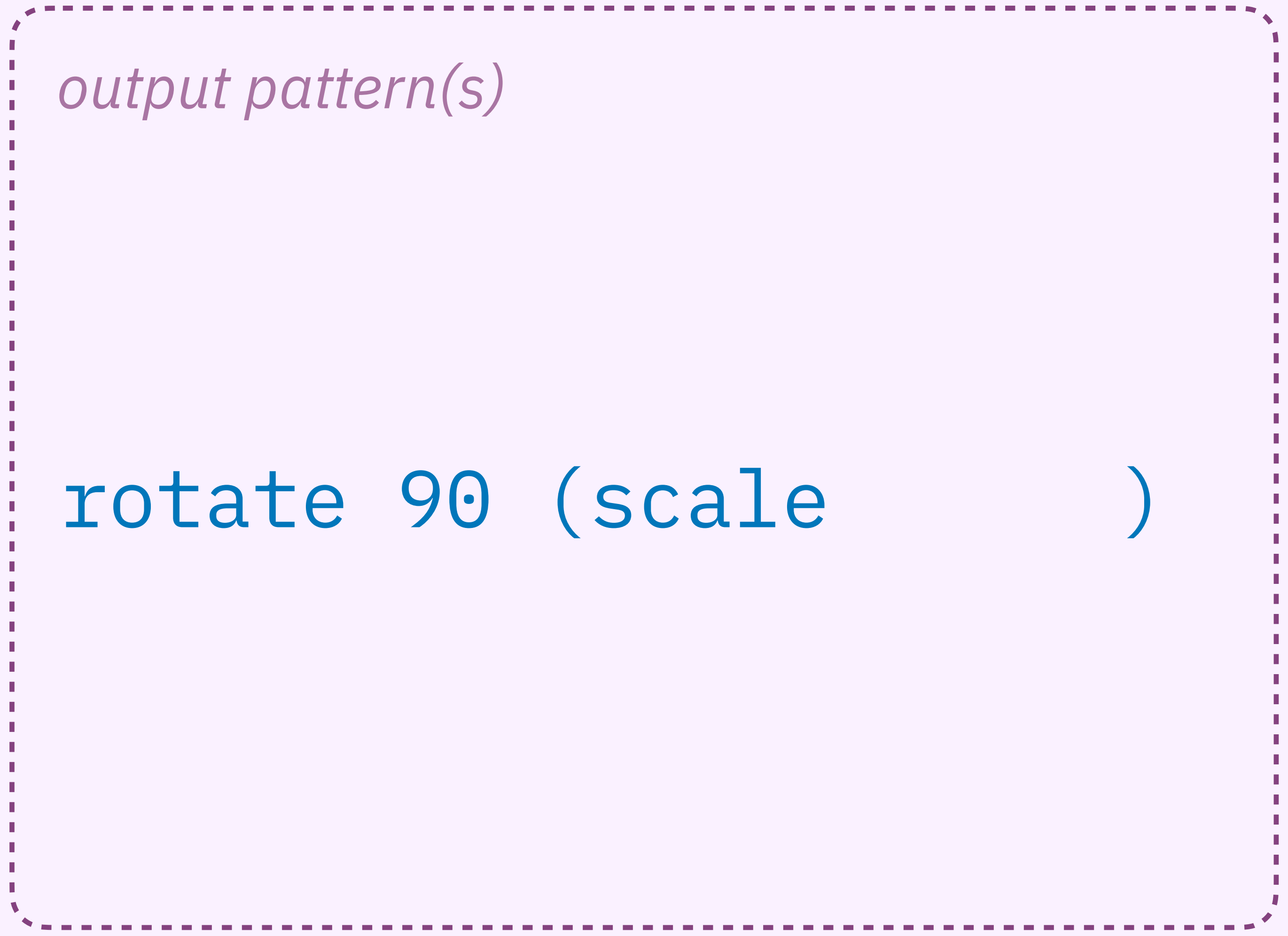


step 1 **pick two e-classes**

step 2a **if e-classes contain matching e-nodes, for each pair of matching e-nodes, add to pattern & run step 2 with matching e-nodes' children**

how does **e-graph anti-unification** work?

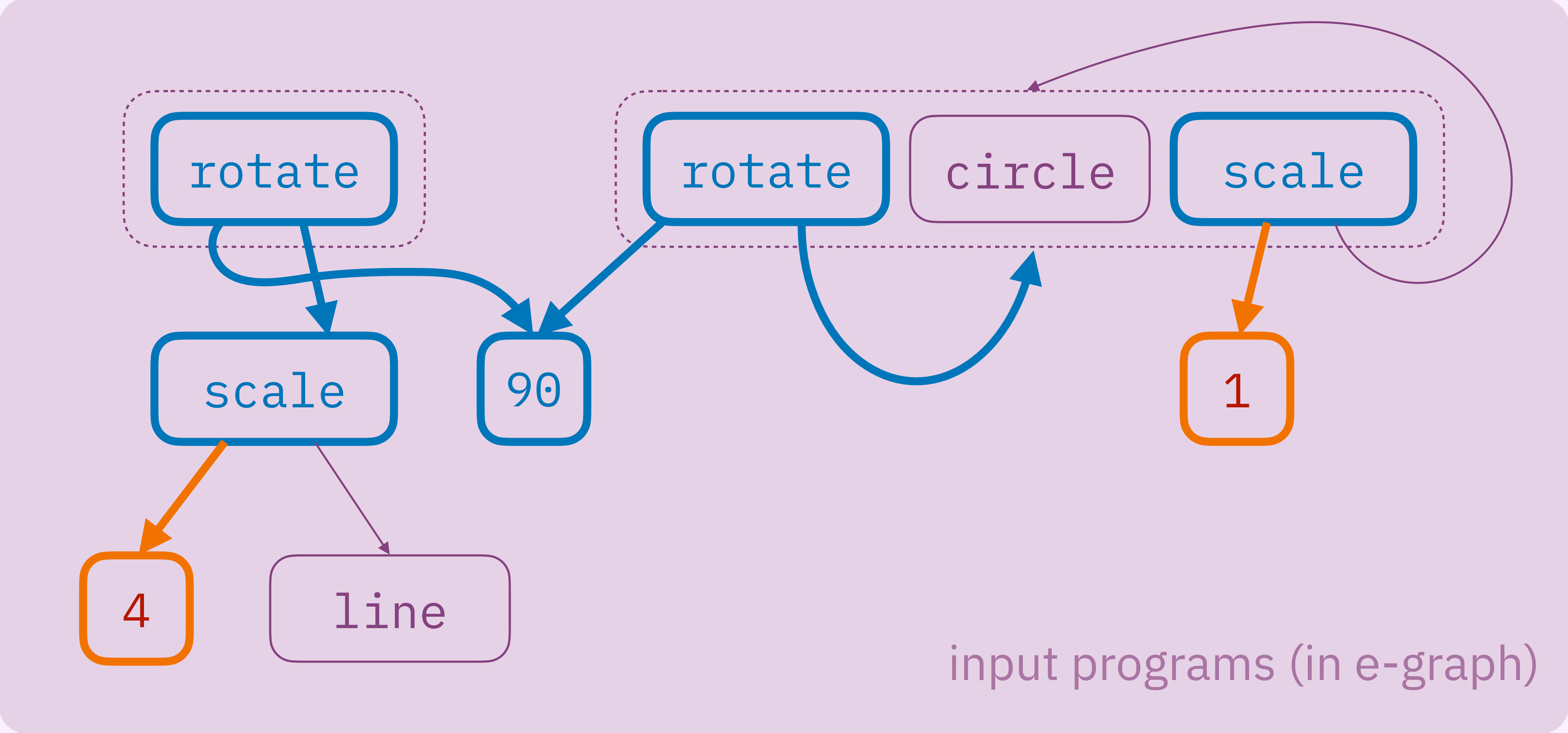




step 1 **pick two e-classes**

step 2a **if e-classes contain matching e-nodes, for each pair of matching e-nodes, add to pattern & run step 2 with matching e-nodes' children**

how does **e-graph anti-unification** work?



output pattern(s)

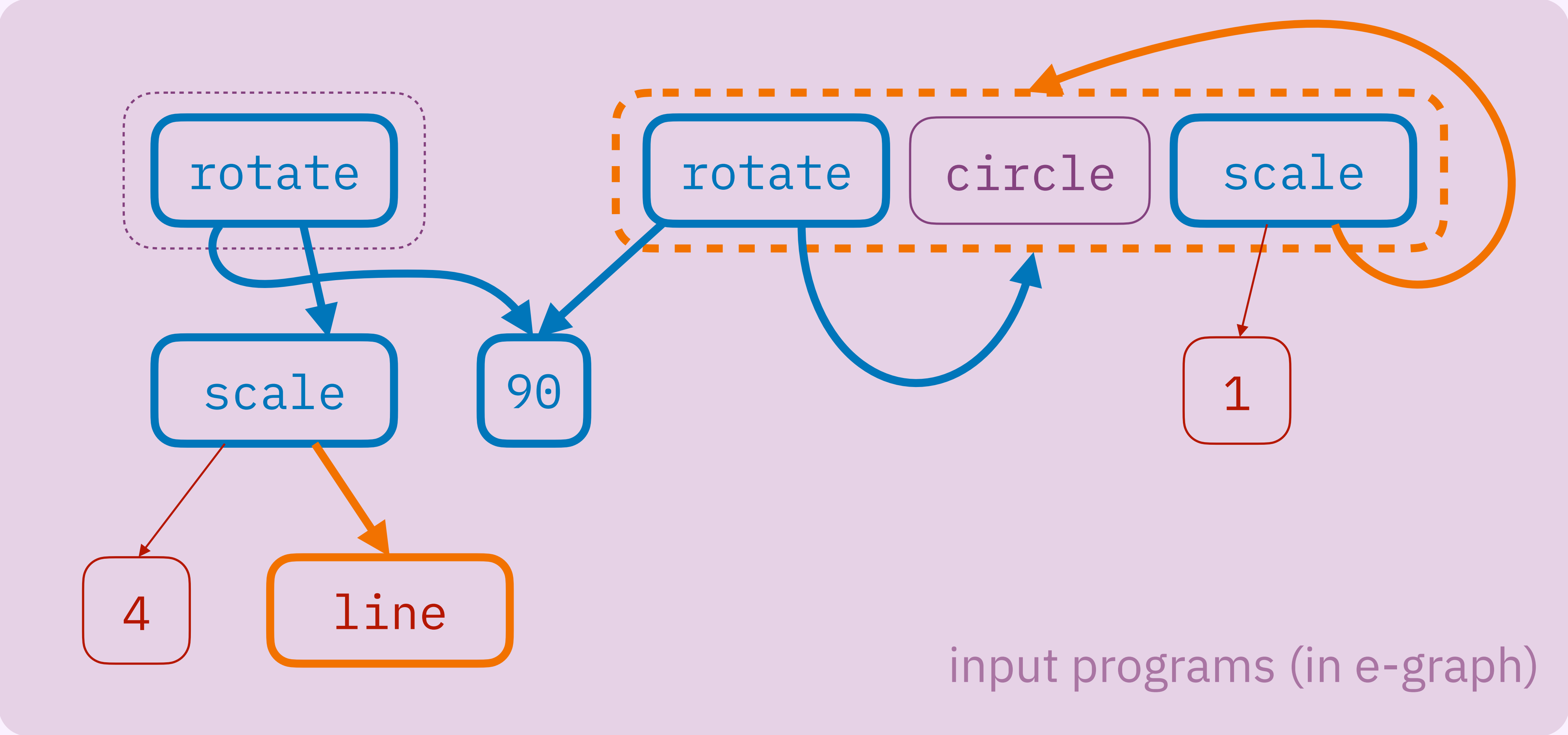
`rotate 90 (scale ?x    )`

step 1 **pick two e-classes**

step 2a **if e-classes contain matching e-nodes, for each pair of matching e-nodes, add to pattern & run step 2 with matching e-nodes' children**

step 2b **otherwise, insert hole in pattern**

how does **e-graph anti-unification** work?



output pattern(s)

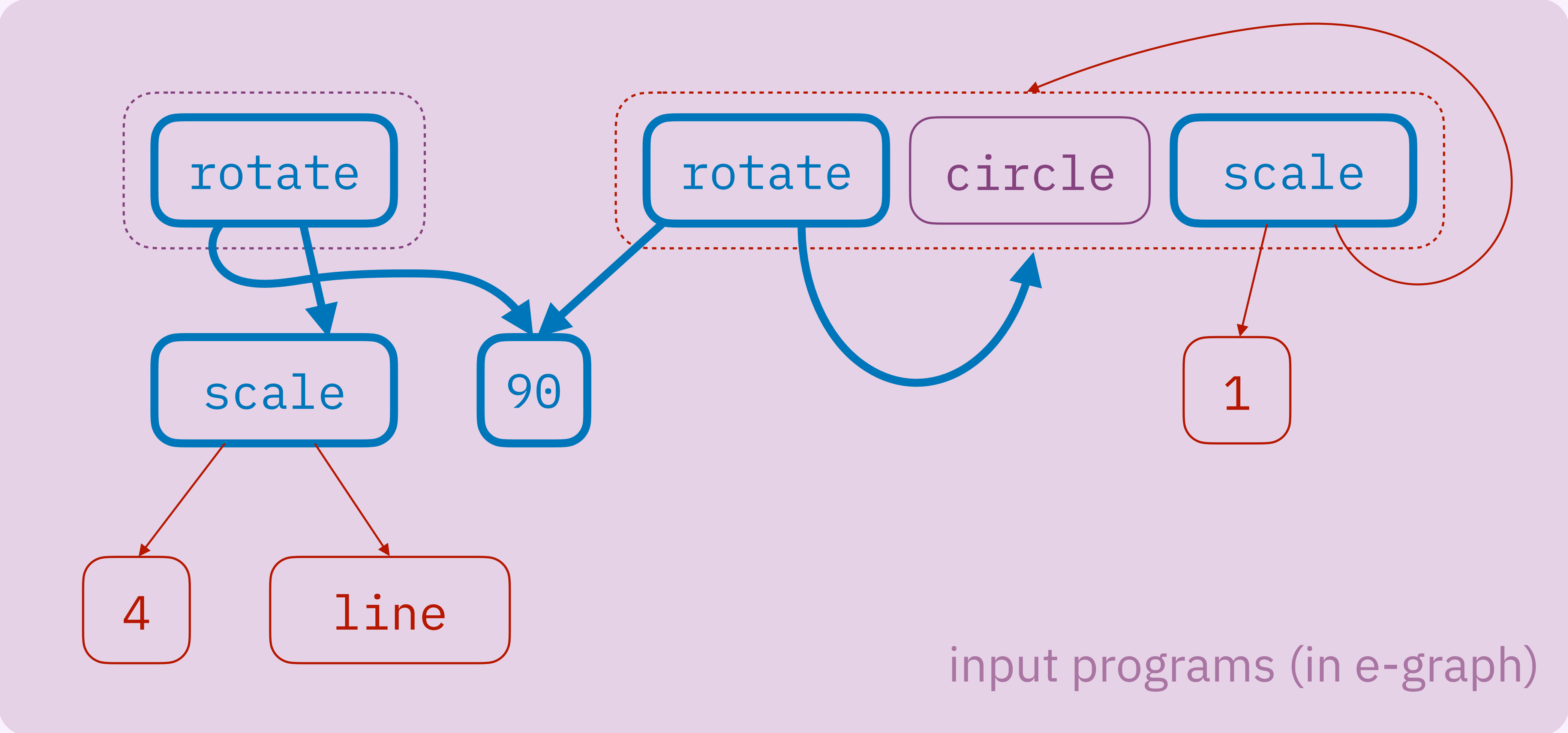
`rotate 90 (scale ?x ?y)`

step 1 **pick two e-classes**

step 2a **if e-classes contain matching e-nodes, for each pair of matching e-nodes, add to pattern & run step 2 with matching e-nodes' children**

step 2b **otherwise, insert hole in pattern**

how does **e-graph anti-unification** work?



output pattern(s)

```
rotate 90 (scale ?x ?y)
```

step 1 **pick two e-classes**

step 2a **if e-classes contain matching e-nodes, for each pair of matching e-nodes, add to pattern & run step 2 with matching e-nodes' children**

step 2b **otherwise, insert hole in pattern**

step 3 **do this for all pairs of e-classes in the e-graph**

how does **e-graph anti-unification** work?

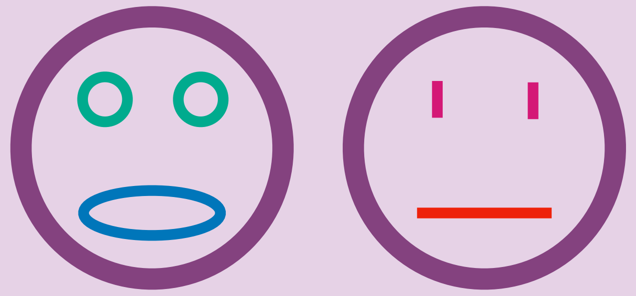
step 1	**pick two e-classes**
step 2a	**if e-classes contain matching e-nodes, for each pair of matching e-nodes, add to pattern & run step 2 with matching e-nodes' children**
step 2b	**otherwise, insert hole in pattern**
step 3	**do this for all pairs of e-classes in the e-graph**

how does **e-graph anti-unification** work?

how does babble work?

user inputs

program corpus



list of equivalences

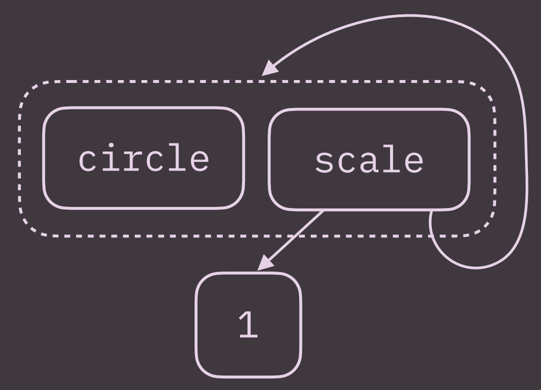
```
circle ≡ rotate 90 circle
?x ≡ scale 1 ?x
...
```

babble

insight

use e-graphs

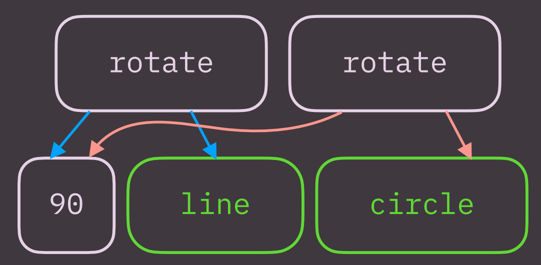
to create and store infinite equivalent input variants



contribution 2

e-graph anti-unification

to propose candidate abstractions in the presence of input variants




```
rotate 90 ?x
```

contribution 3

targeted CSE

to pick the best programs and abstractions

```
let f = λx -> … in
 let f = λx -> … in
  let f = λx -> … in
   f 2
```

babble outputs

learned library

```
let face = λshape →
```

rewritten inputs

```
face circle
```

```
face line
```

what's the challenge?

how does babble work?

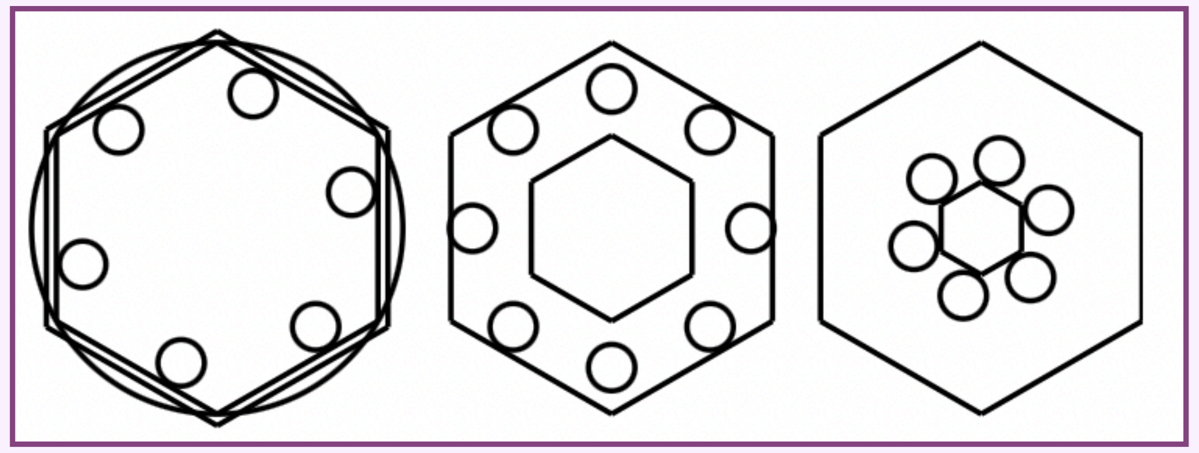
how well does it work?

what's the challenge?

how does babble work?

how well does it work?

nuts & bolts



250 programs

how well does it work?

nuts-bolts.bab — BSCode

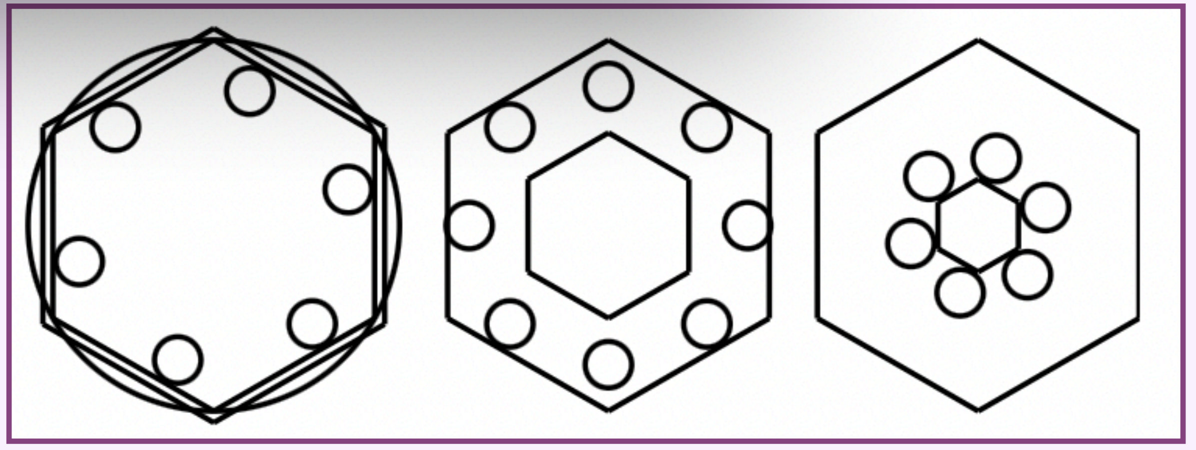
```
(C (C (T (repeat (T 1 (M 1 0 -0.5 (/ 0.5 (tan (/ pi 6))))) 6 (M 1 (/ (* 2 pi) 6) 0
0)) (M 2 0 0 0)) (T (repeat (T 1 (M 1 0 -0.5 (/ 0.5 (tan (/ pi 6))))) 6 (M 1 (/ (* 2
pi) 6) 0 0)) (M 2.25 0 0 0))) (T (repeat (T 1 (M 1 0 -0.5 (/ 0.5 (tan (/ pi 6))))) 6
(M 1 (/ (* 2 pi) 6) 0 0)) (M 1 0 0 0)))
```

babble demo 2 — iTerm

```
Welcome to fish, the friendly interactive shell
Type `help` for instructions on how to use fish
david@mbpro ~/D/d/babble (popl23)> cargo run --bin=drawings --release -- harness/data/
cogsci/nuts-bolts.bab --beams 400 --lps 1 --rounds 5 --max-arity 2 --dsr harness/data/
benchmark-dsrs/drawings.nuts-bolts.rewrites
```

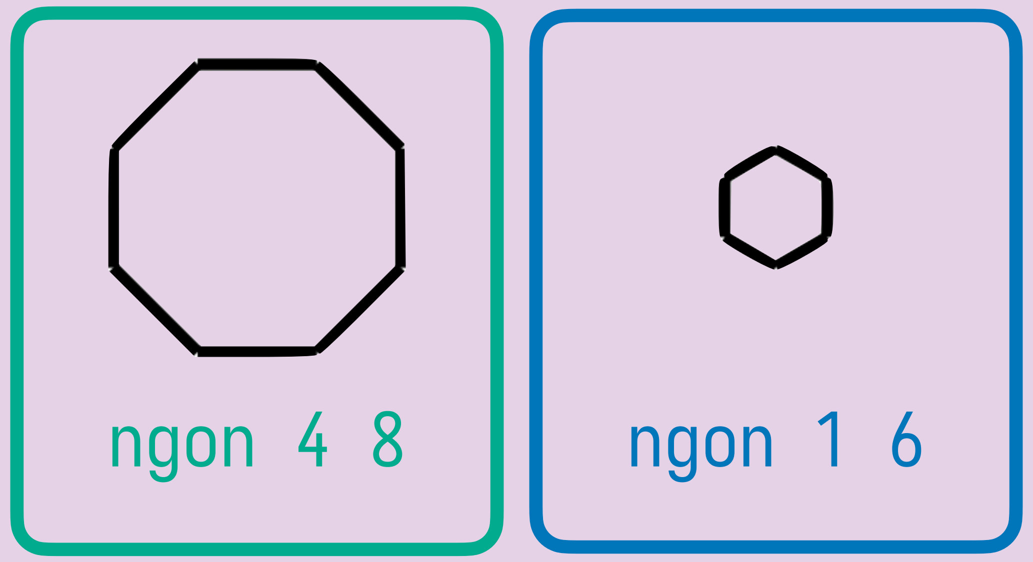
250 programs

nuts & bolts



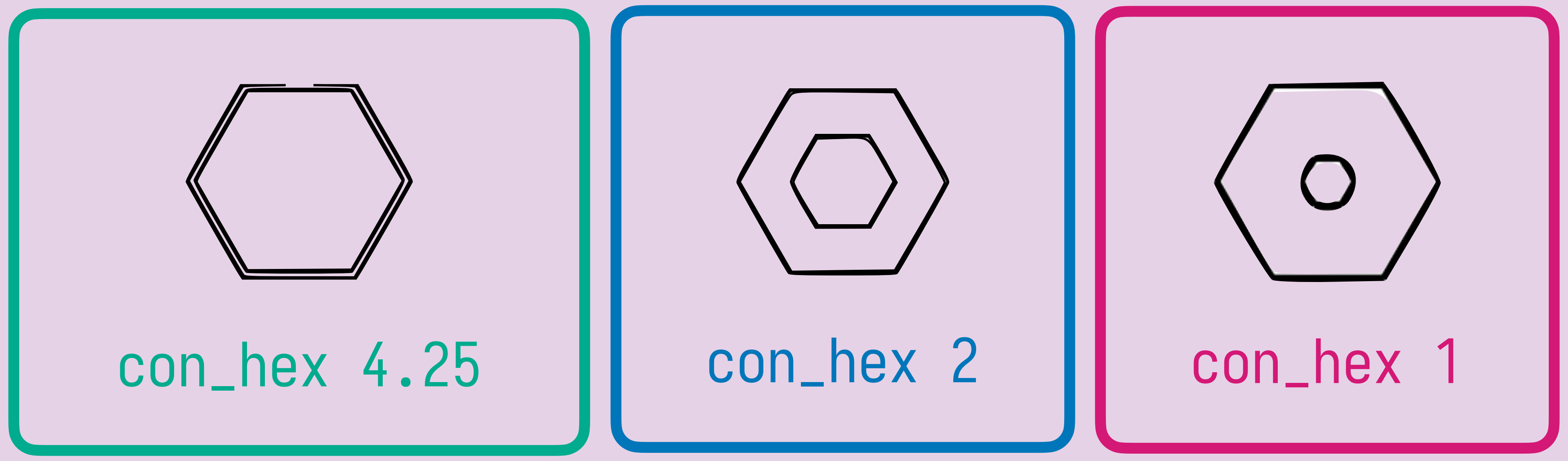
how well does it work?
qualitative eval

```
ngon = λsize sides →
  T (repeat (T l (M 1 0 -0.5 (0.5 / tan (π / sides)))) sides
            (M 1 ((2 * π) / sides) 0 0))
    (M size 0 0 0)
```



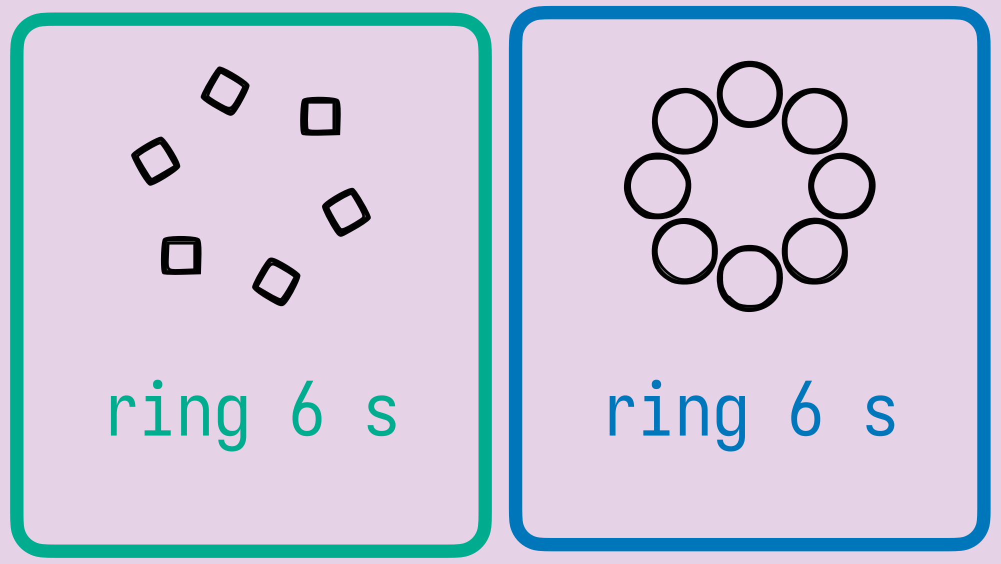
scaled n-gon

```
con_hex = λinner_size →
  C (ngon 4 6) (ngon inner_size 6)
```



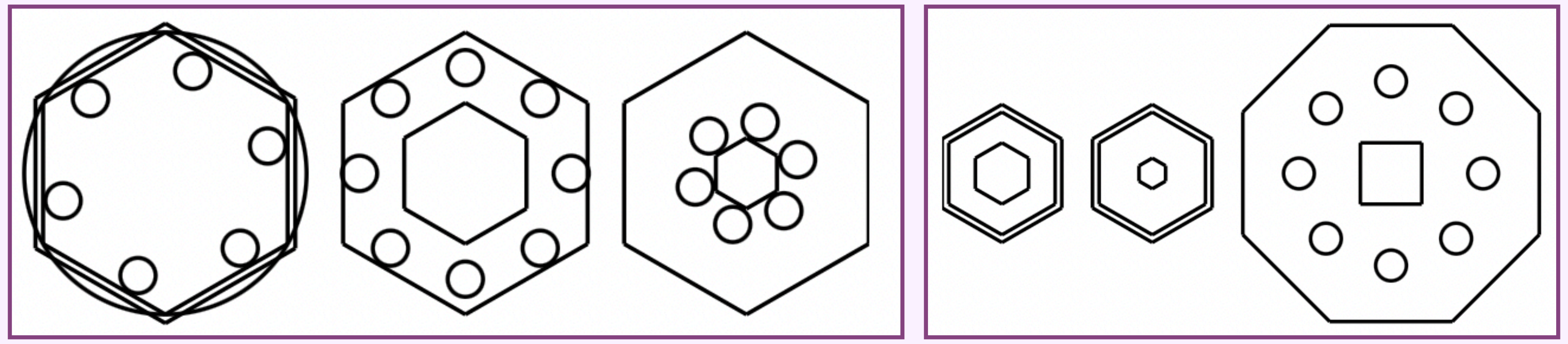
concentric scaled hexagons

```
ring = λn shape →
  repeat (offset 1.5 shape) n (rotate n)
```

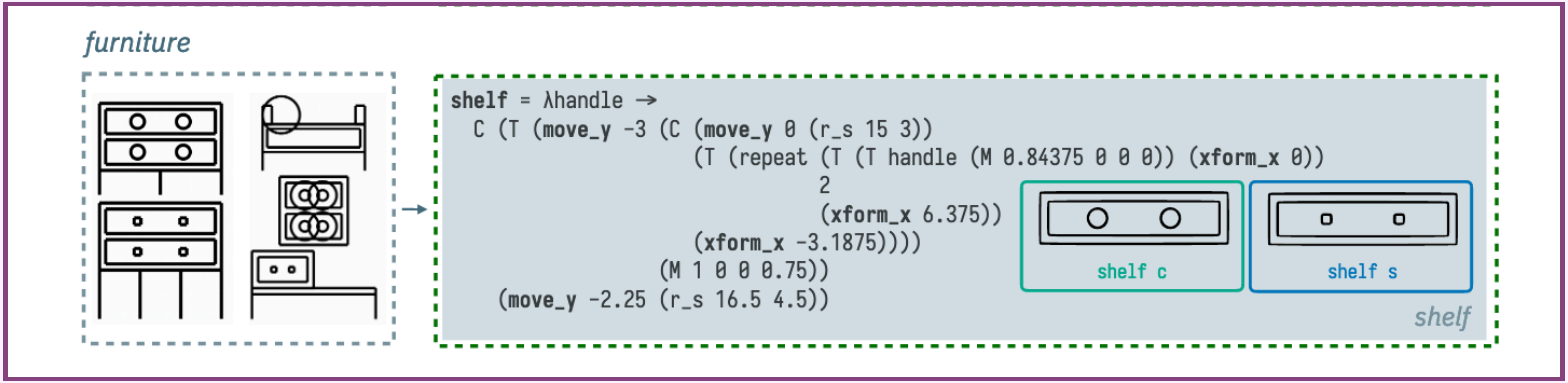


ring of shapes

nuts & bolts

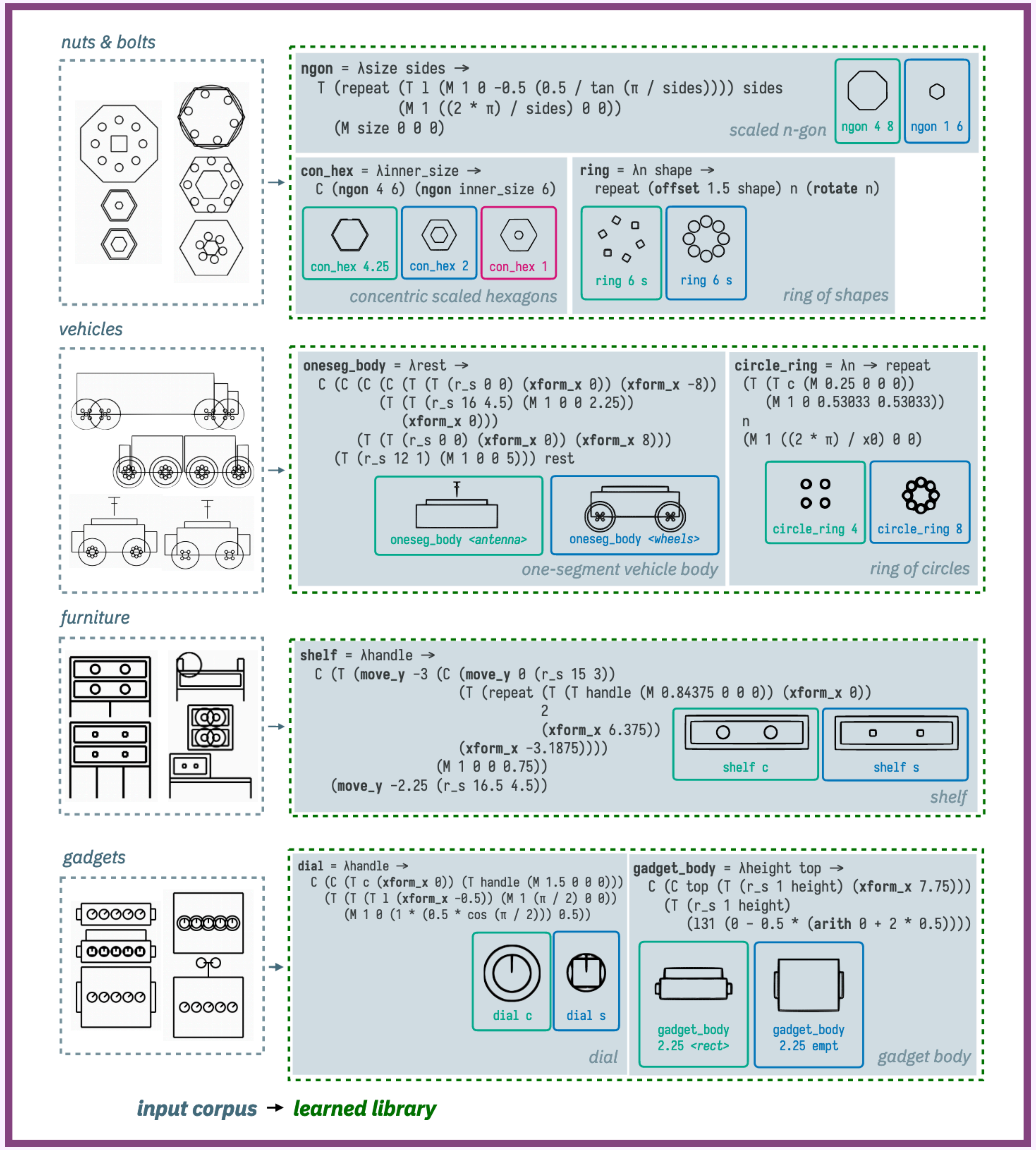


how well does it work?
qualitative eval



how well does it work?
qualitative eval

check the paper for more examples!



how well does it work?

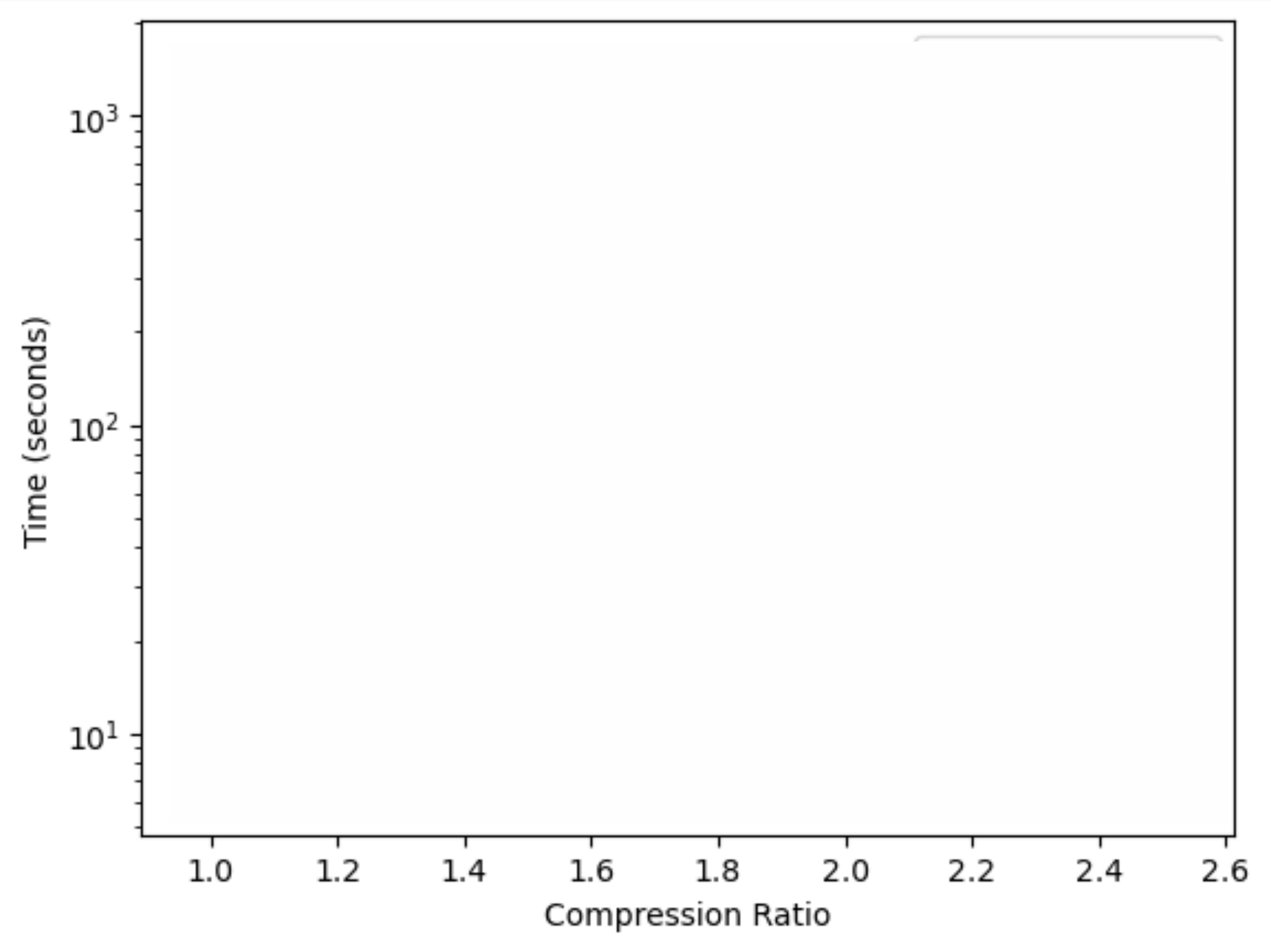
quantitative eval

does it scale?

does it compress?

how well does it work?
quantitative eval

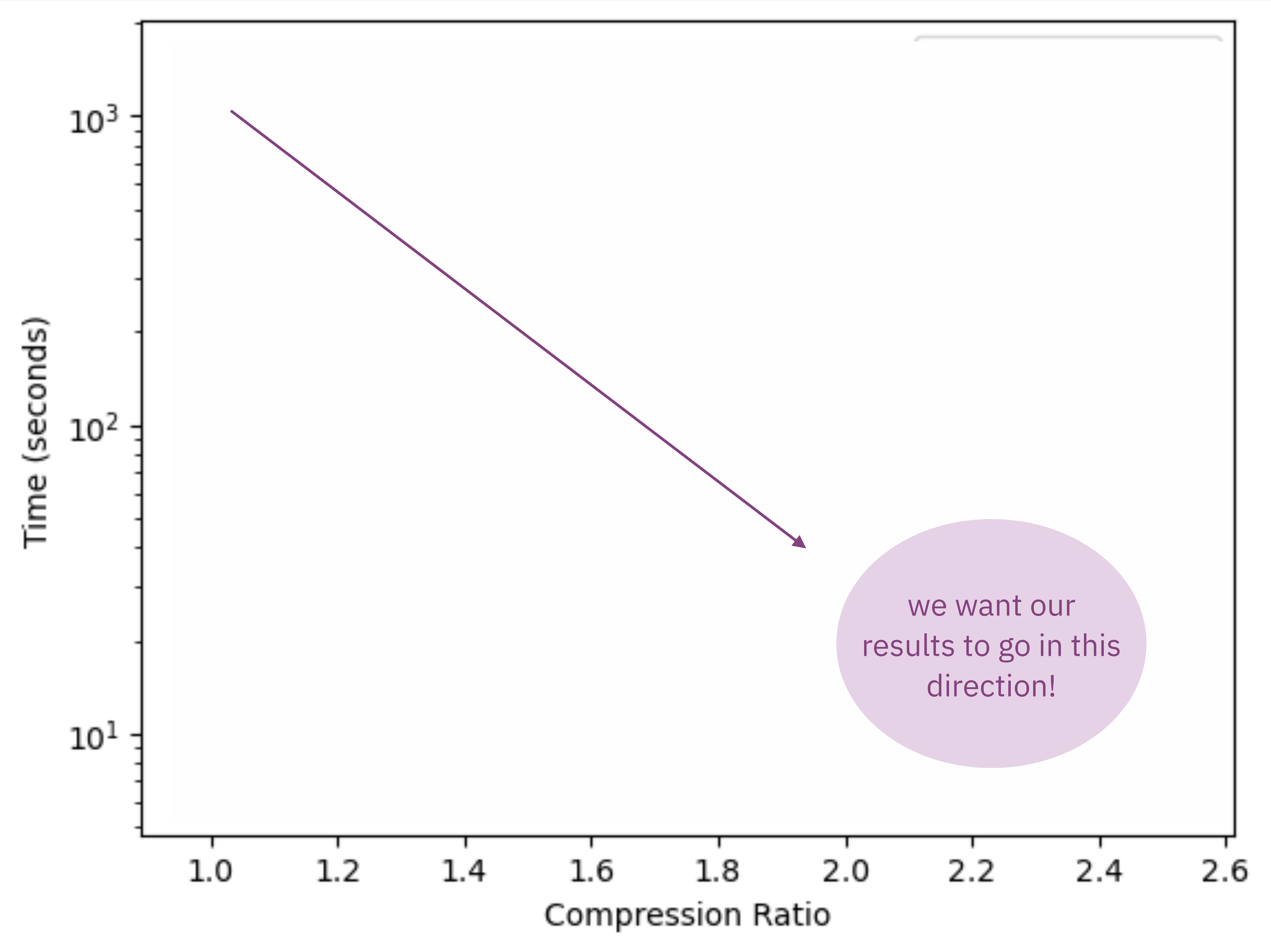
does it scale?



does it compress?

how well does it work?
quantitative eval

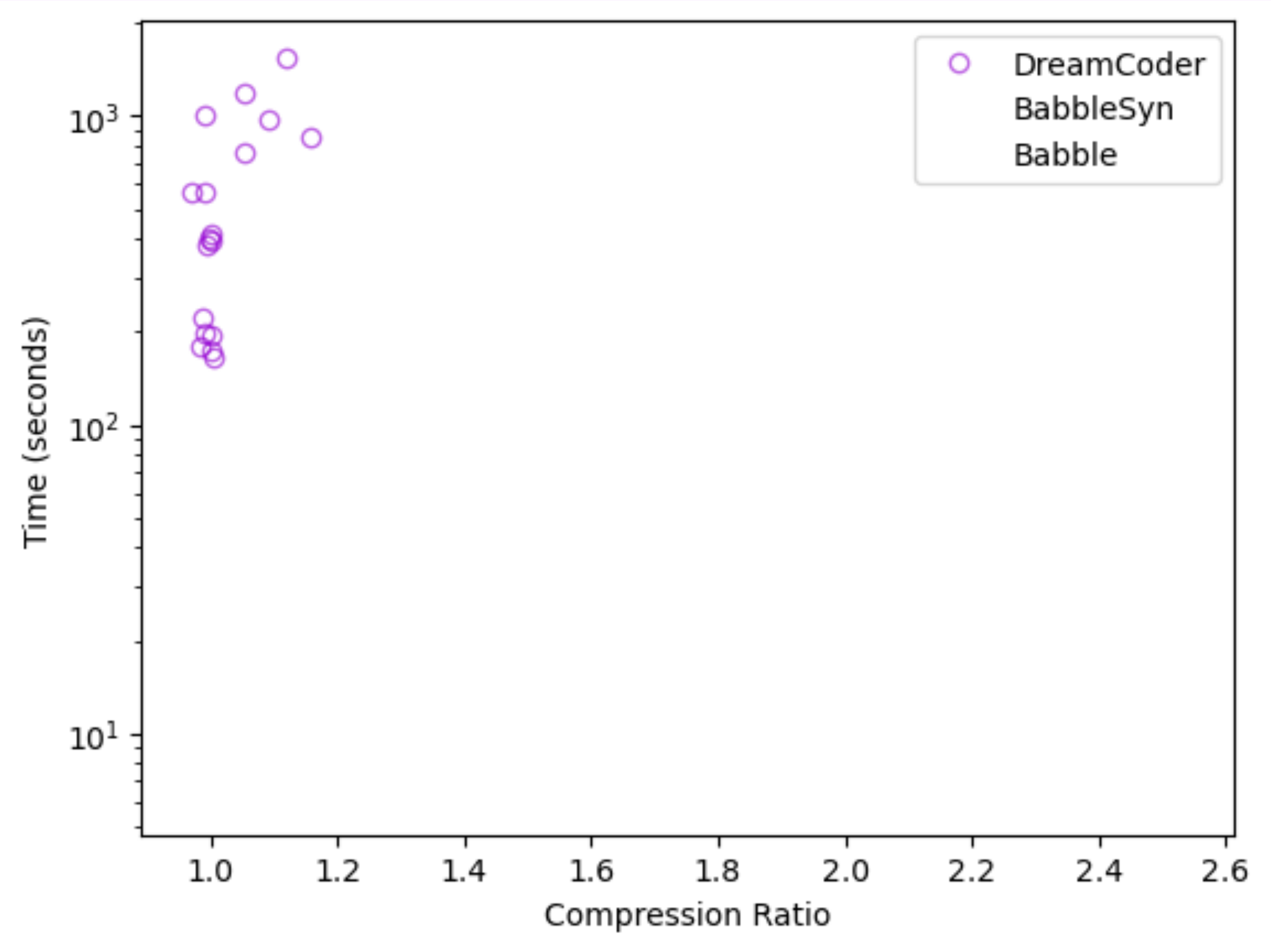
does it scale?



does it compress?

how well does it work?
quantitative eval

physics domain results



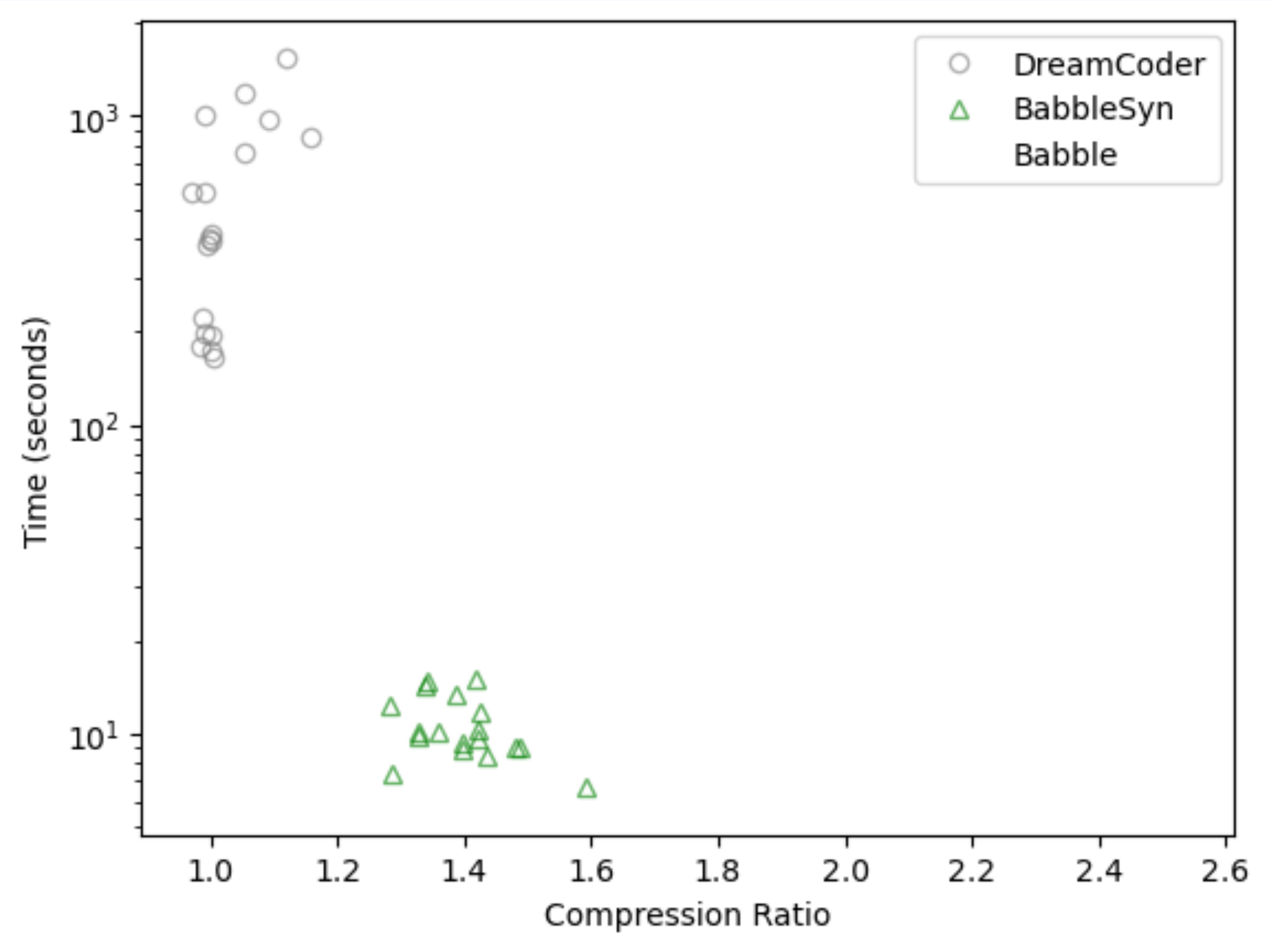
does it scale?

does it compress?

how well does it work?
quantitative eval

physics domain results

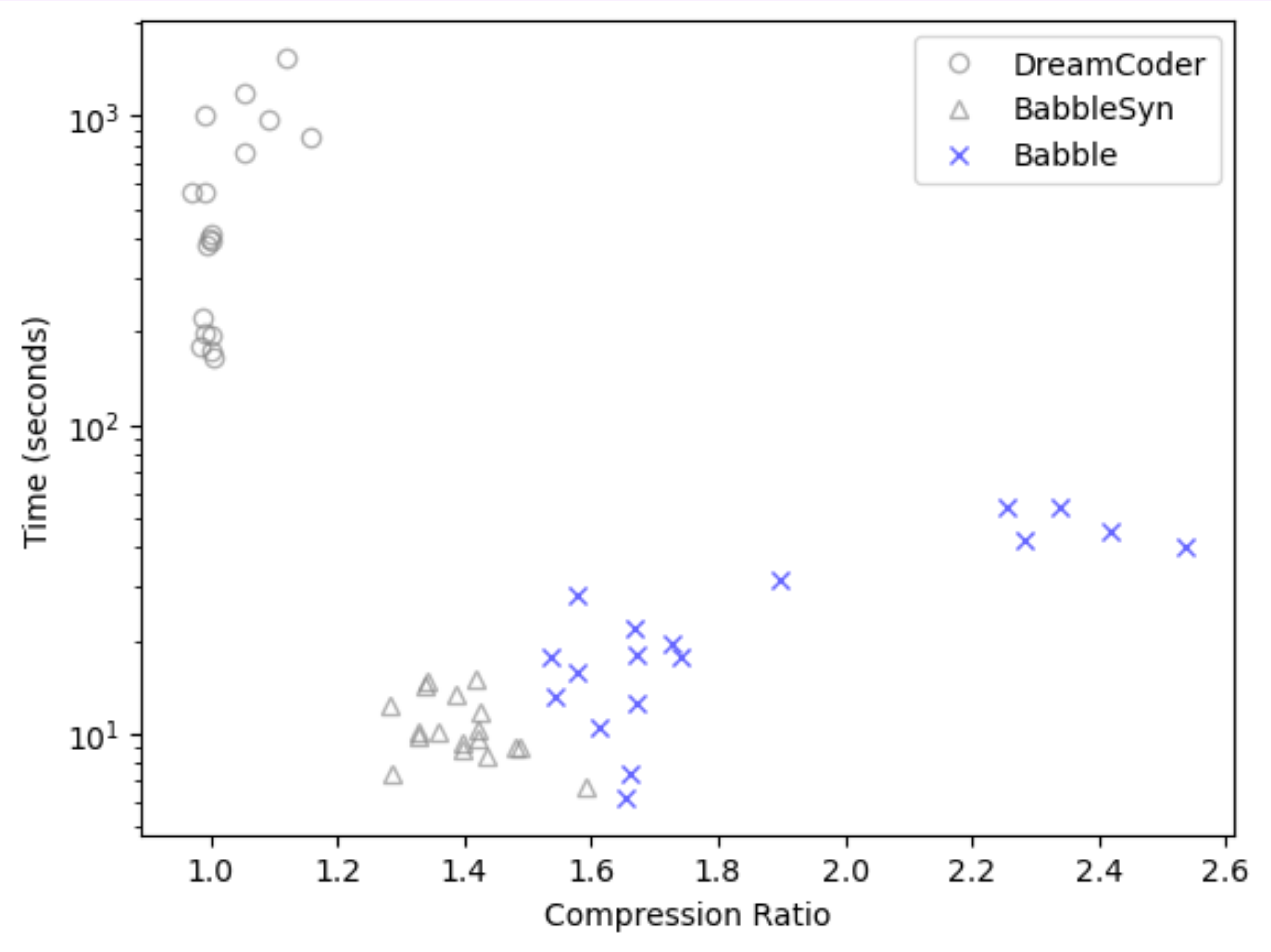
does it scale?



does it compress?

how well does it work?
quantitative eval

physics domain results



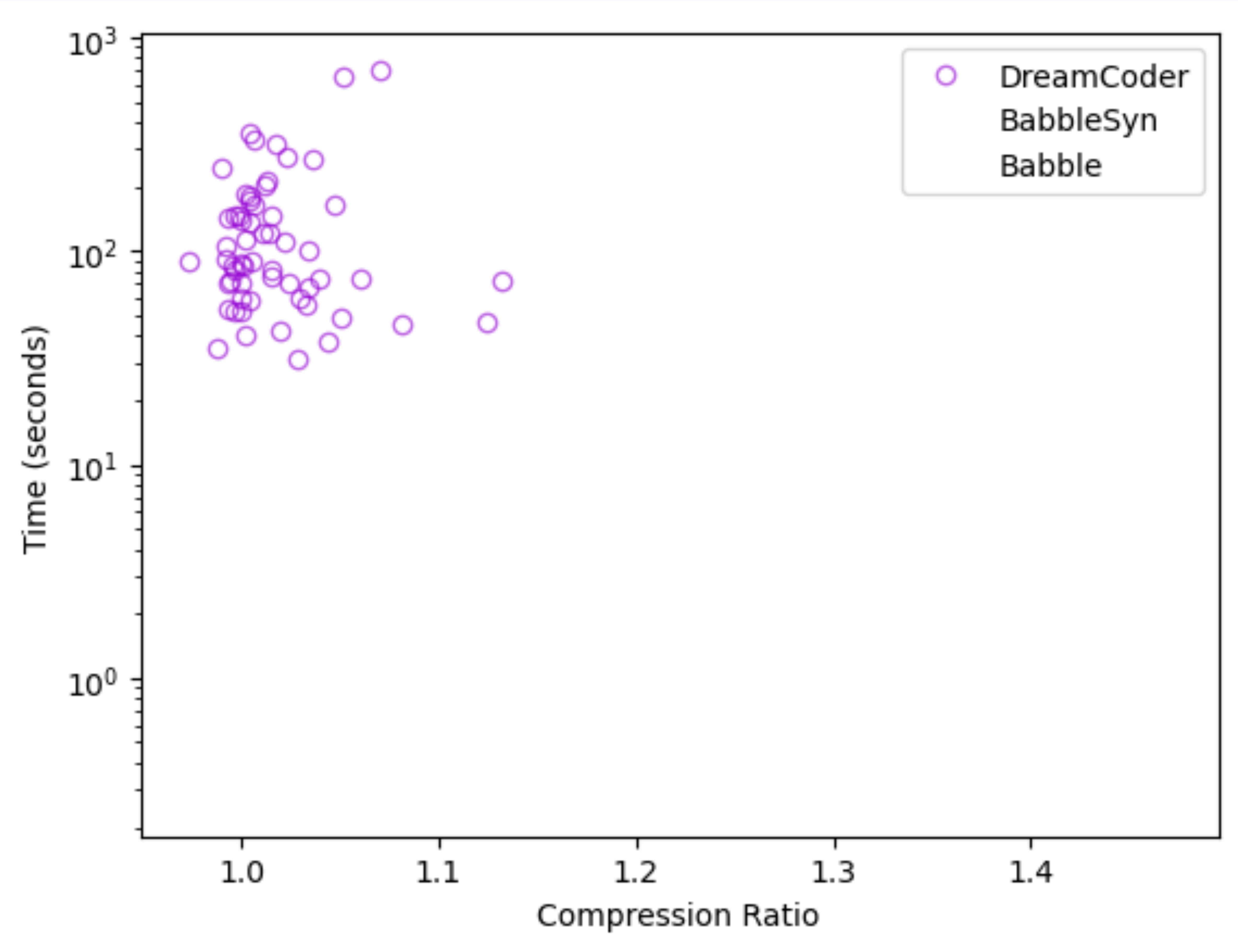
does it scale?

does it compress?

how well does it work?
quantitative eval

list domain results

does it scale?

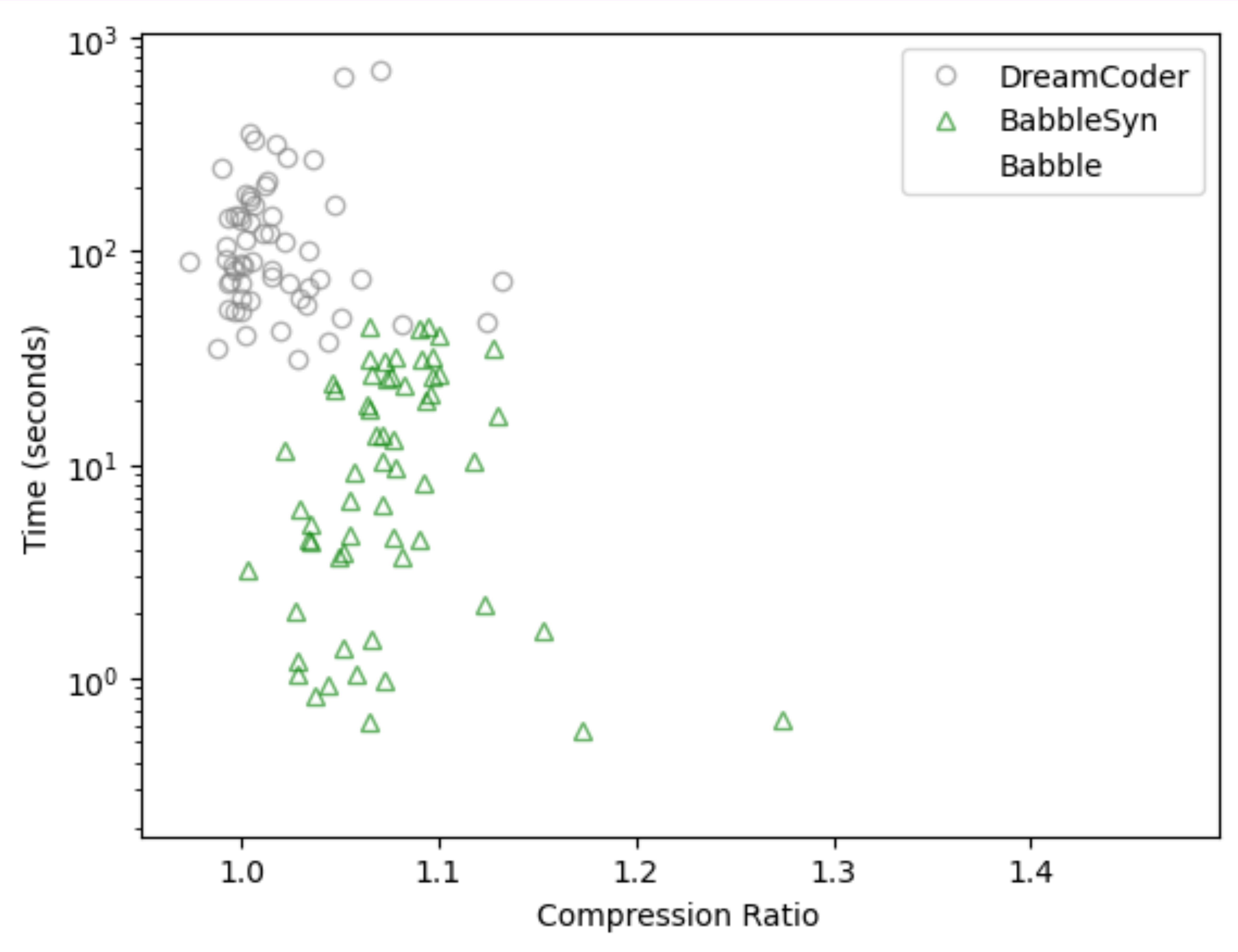


does it compress?

how well does it work?
quantitative eval

list domain results

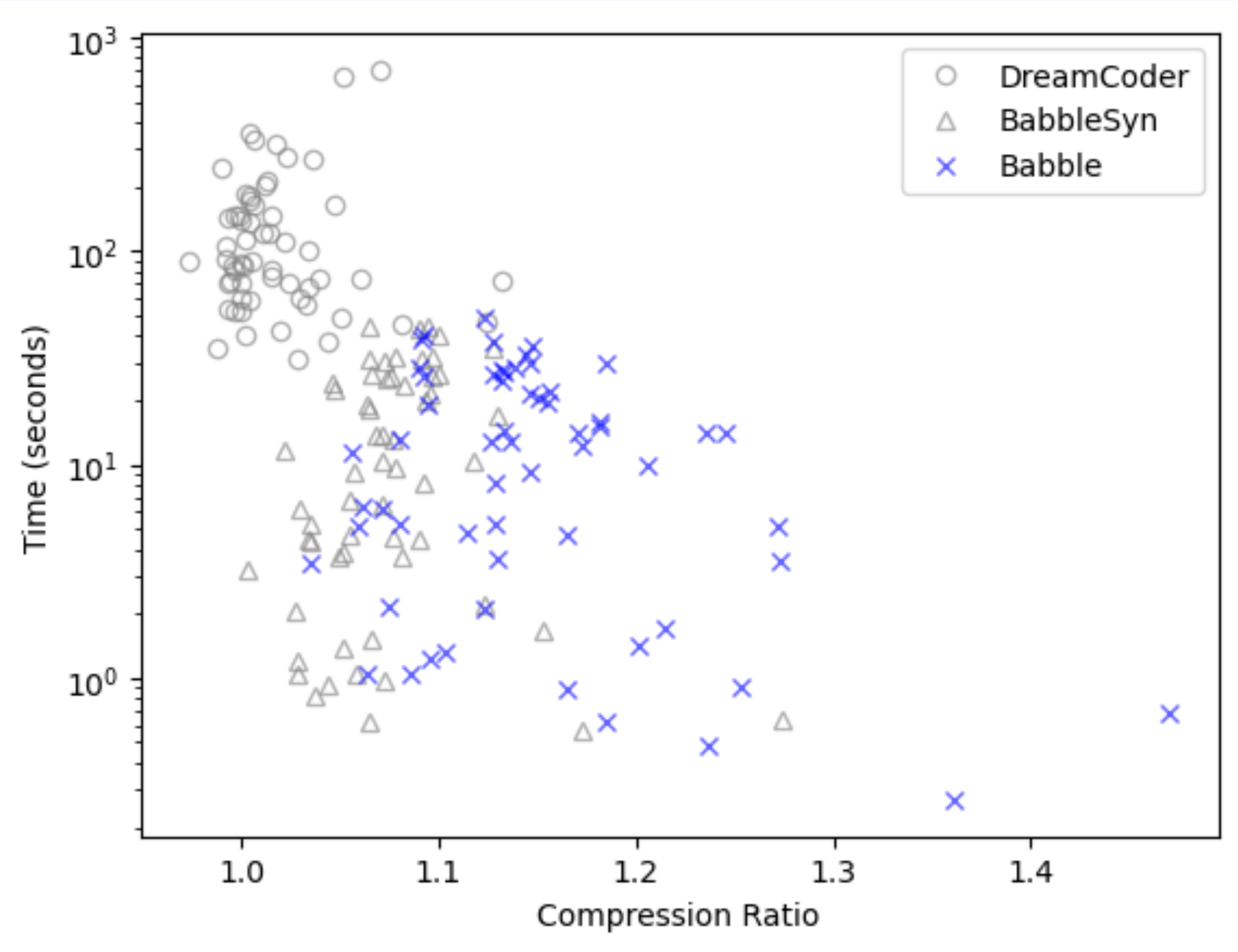
does it scale?



does it compress?

how well does it work?
quantitative eval

list domain results



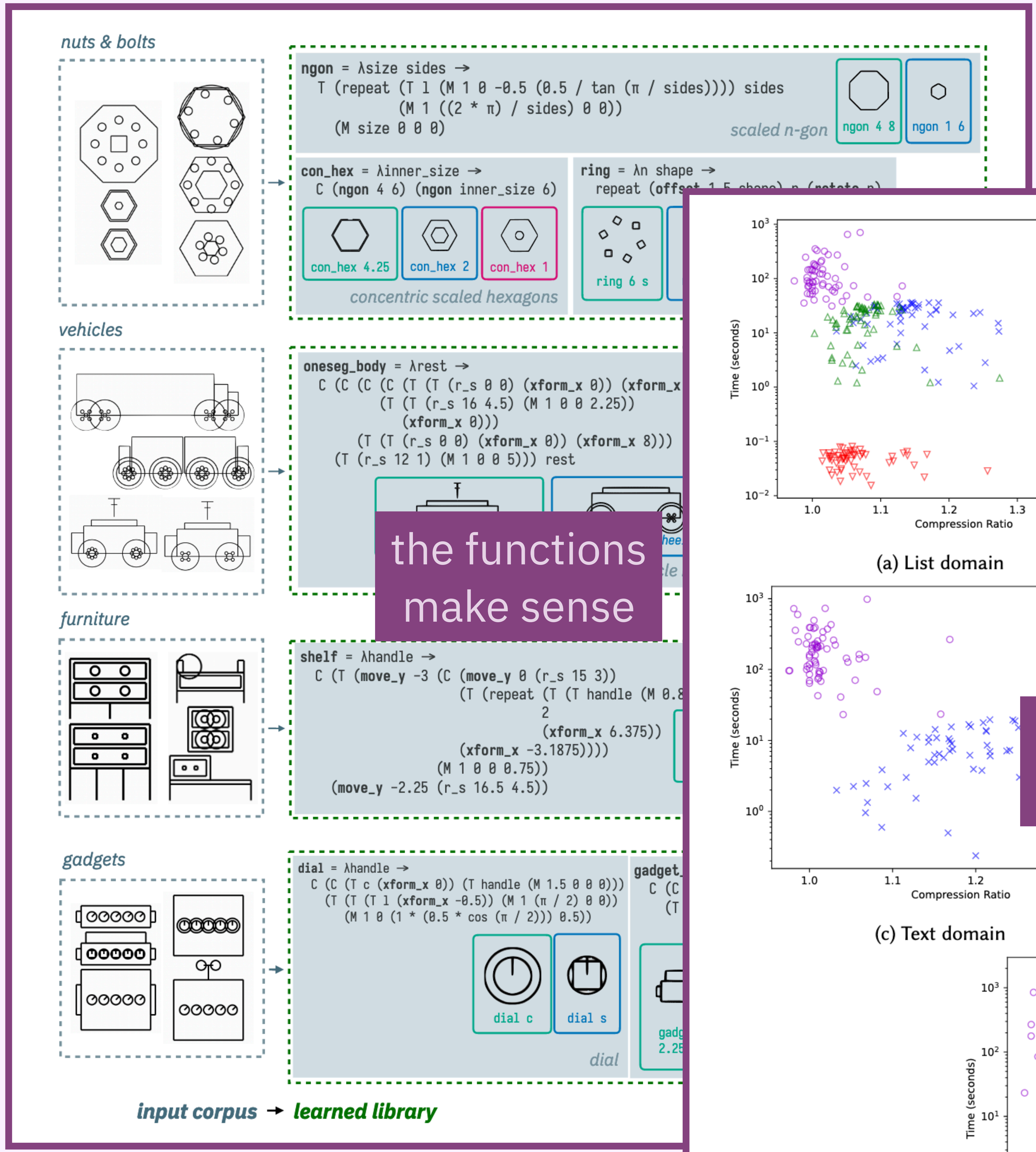
does it scale?

does it compress?

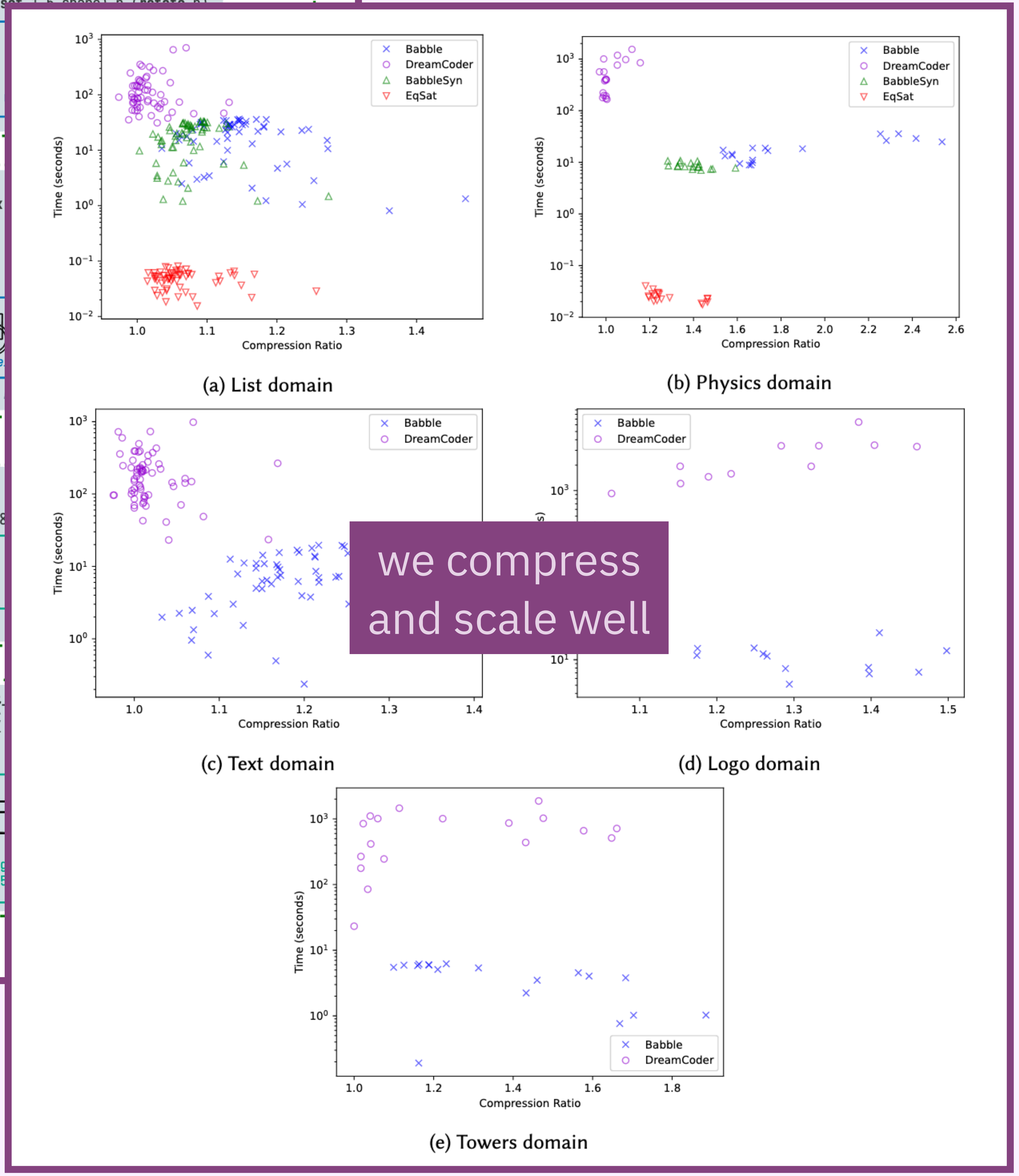
what's the challenge?

how does babble work?

how well does it work?



the functions make sense

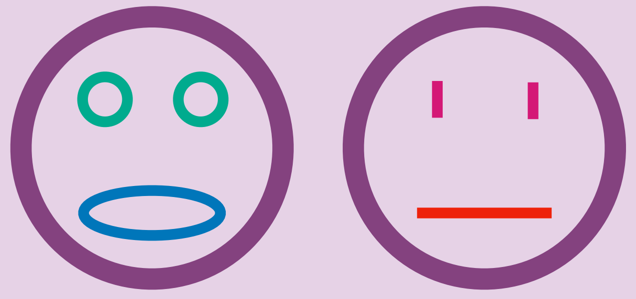


we compress and scale well

babble *learning better abstractions with e-graphs and anti-unification*

user inputs

program corpus



list of equivalences

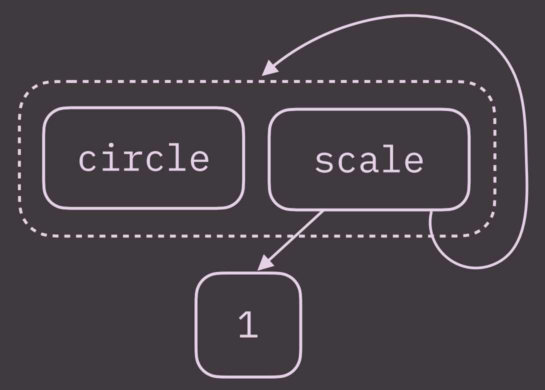
```
circle ≡ rotate 90 circle
?x ≡ scale 1 ?x
...
```

babble

insight

use e-graphs

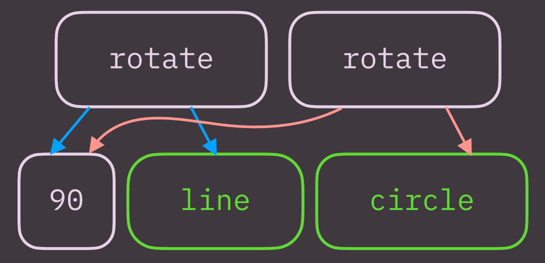
to create and store infinite equivalent input variants



contribution 2

e-graph anti-unification

to propose candidate abstractions in the presence of input variants

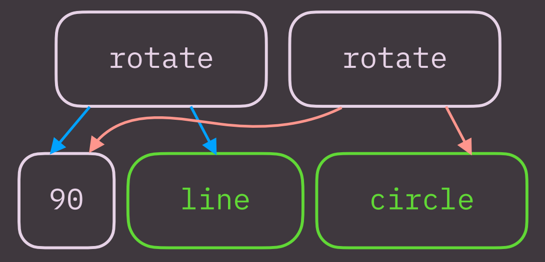


rotate 90 ?x

contribution 3

targeted CSE

to pick the best programs and abstractions



rotate 90 ?x

babble outputs

learned library

`let face = λshape →`

rewritten inputs

`face circle`

`face line`