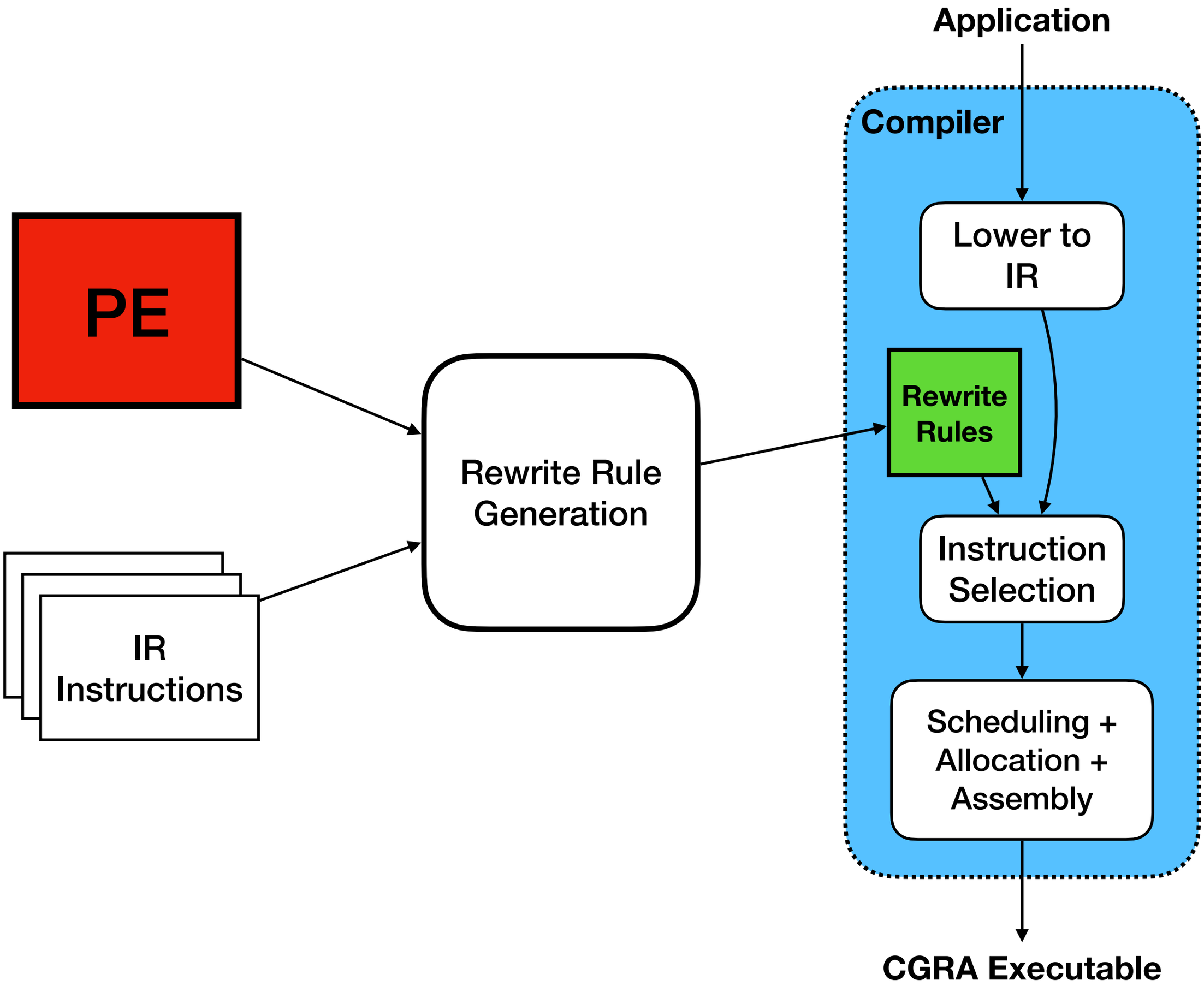
# Generalizing Rewrite Rule Synthesis

Ross Daly


Application
Compiler
Lower to IR
Rewrite Rules
Instruction Selection
Scheduling + Allocation + Assembly
CGRA Executable
PE
Rewrite Rule Generation
IR Instructions

# Large Diversity in Possible PEs

- ISA for the PE can be CISC-like compared to the IR
- PE specialization using Jack's Design Space Exploration
- Requires "Many to 1" rewrite rules
    - Each IR pattern has multiple instructions

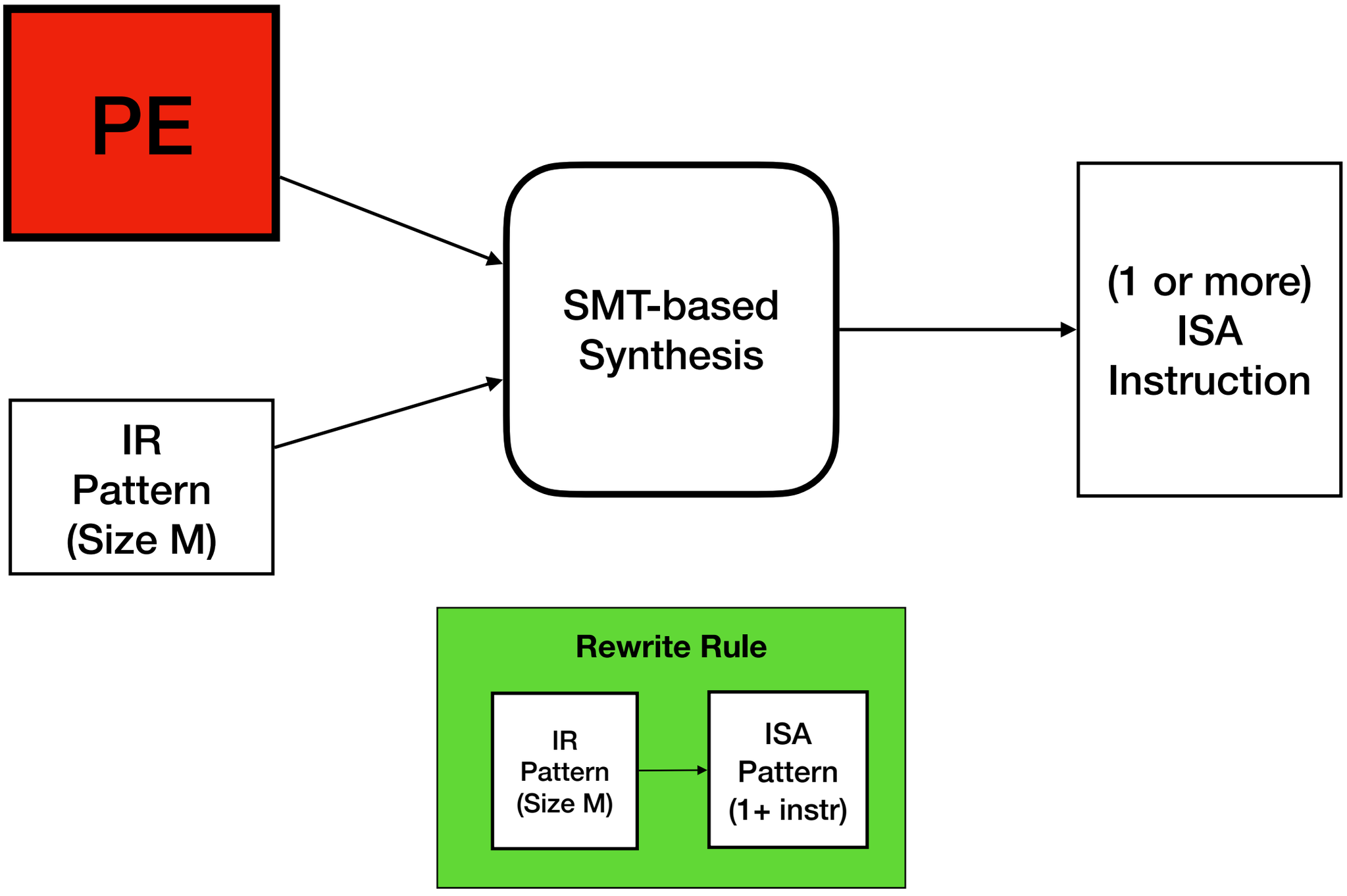
# Large Diversity in Possible PEs

- ISAs for PEs can be RISC-like compared to the IR
- Lassen when using 32-bit IR ops
- Requires “1 to many” rewrite rules
    - Each ISA pattern can has multiple instructions

# A rewrite rule generation tool ***must*** be capable of generating (N->M) rewrite rules

# Previous Synthesis Work

# Daly et al.
# FMCAD 22



# Limitations

- The set of IR patterns is a given to the tool.
  - Each IR pattern was handcrafted
- Performance of synthesizing multiple ISA instructions was bad.

# Generalized Rewrite Rule Synthesis

# IR + ISA used in rest of talk

| IR Instructions | ISA Instructions |
|---|---|
| IR.Const(C) | ISA.Neg(X) |
| IR.Add(X, Y) | ISA.Add(X, Y) |
| IR.Sub(X, Y) | ISA.MulC(X, C) |
| IR.Mul(X, Y) | ISA.Add3(X, Y, Z) |

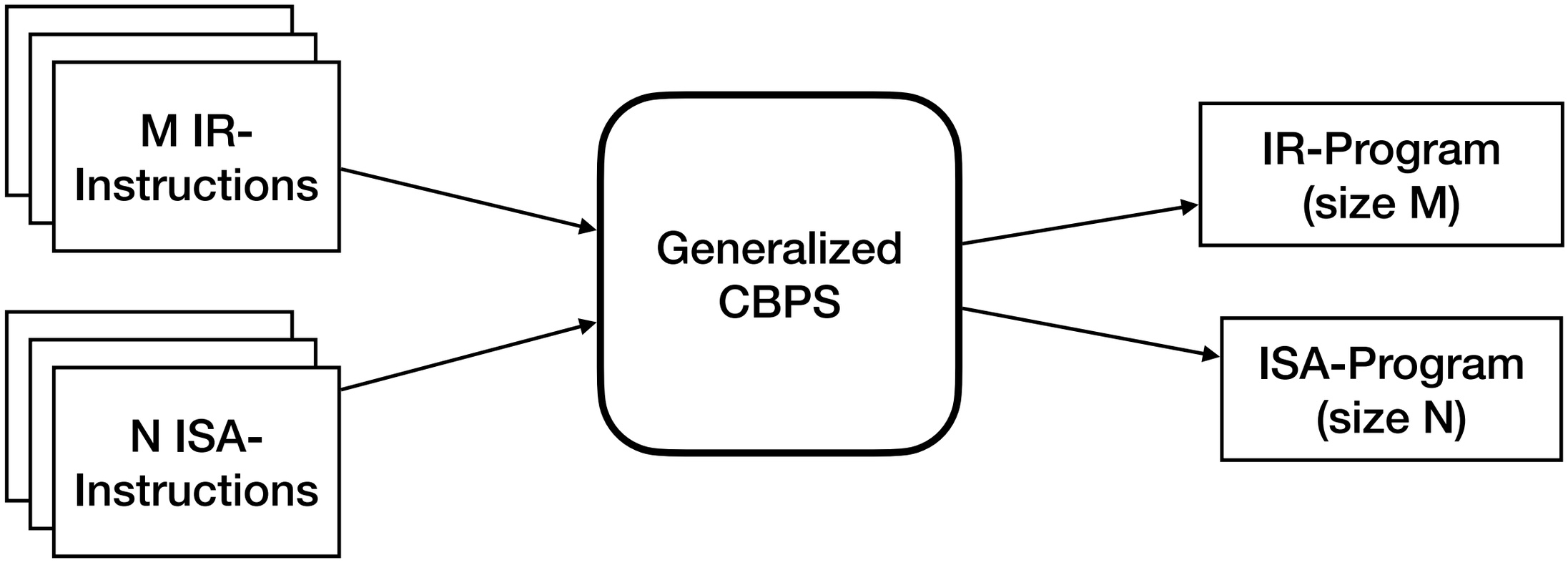
# Problem Statement

Given: Set of IR instructions and
Set of ISA instructions

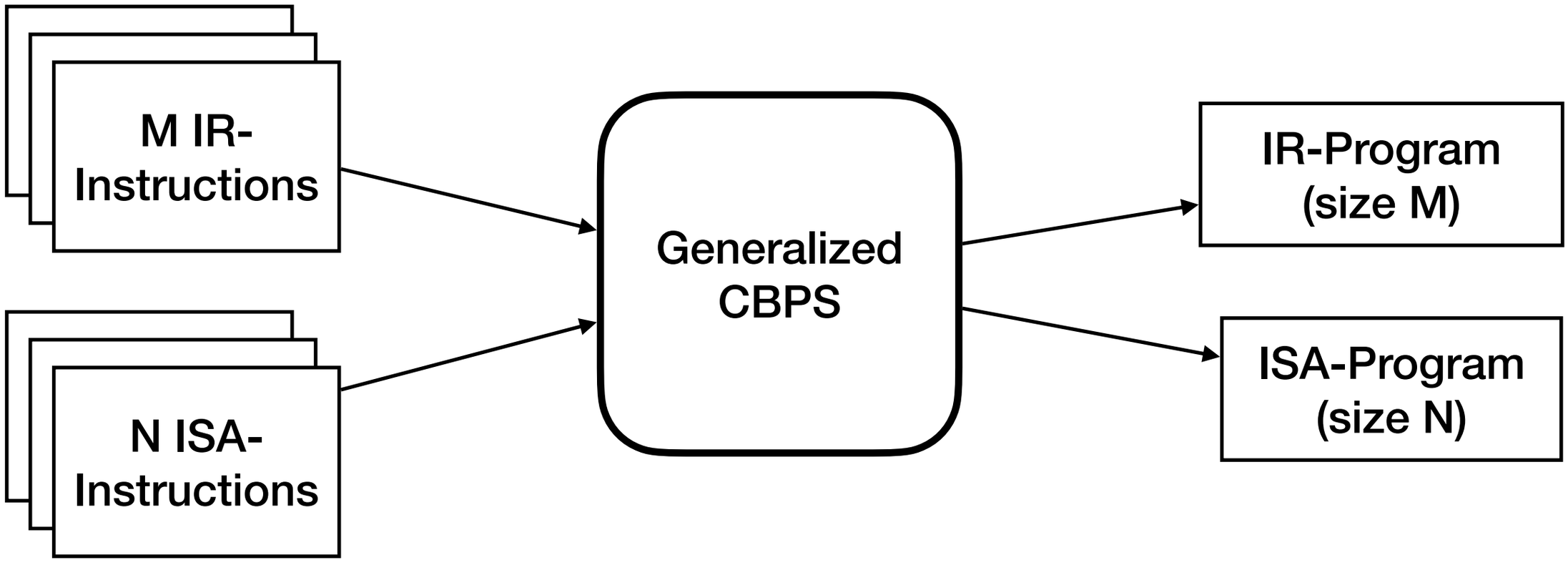
Goal: Automatically generate all* possible (M -> N) rewrite rules efficiently

***Up to a given size**

# Generalized Component-Based Program Synthesis (CBPS)

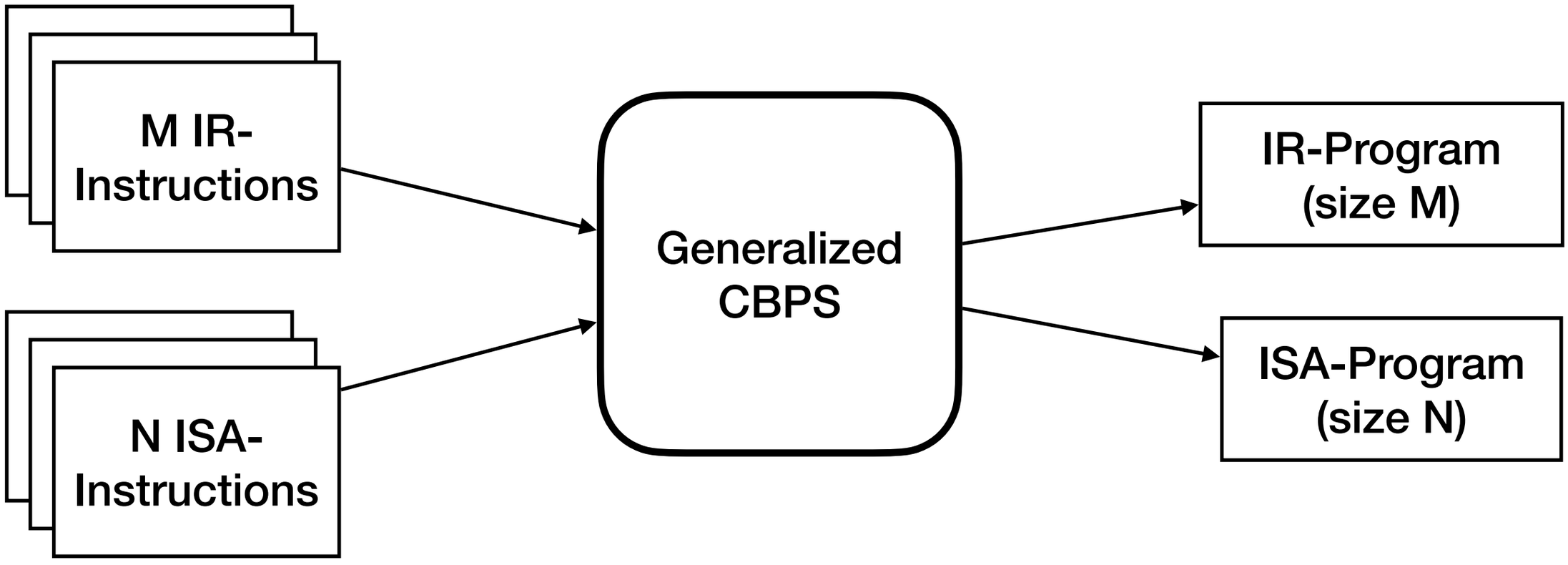


# Generalized Component-Based Program Synthesis (CBPS)




$$\exists\, \mathbf{IRProgram_m}\,,\, \mathbf{ISAProgram_n}\;\forall\, \mathbf{X}\,.$$
$$\mathbf{IRProgram_m(X) = ISAProgram_n(X)}$$

# Solved using Counter Example Guided Induction Synthesis (CEGIS)



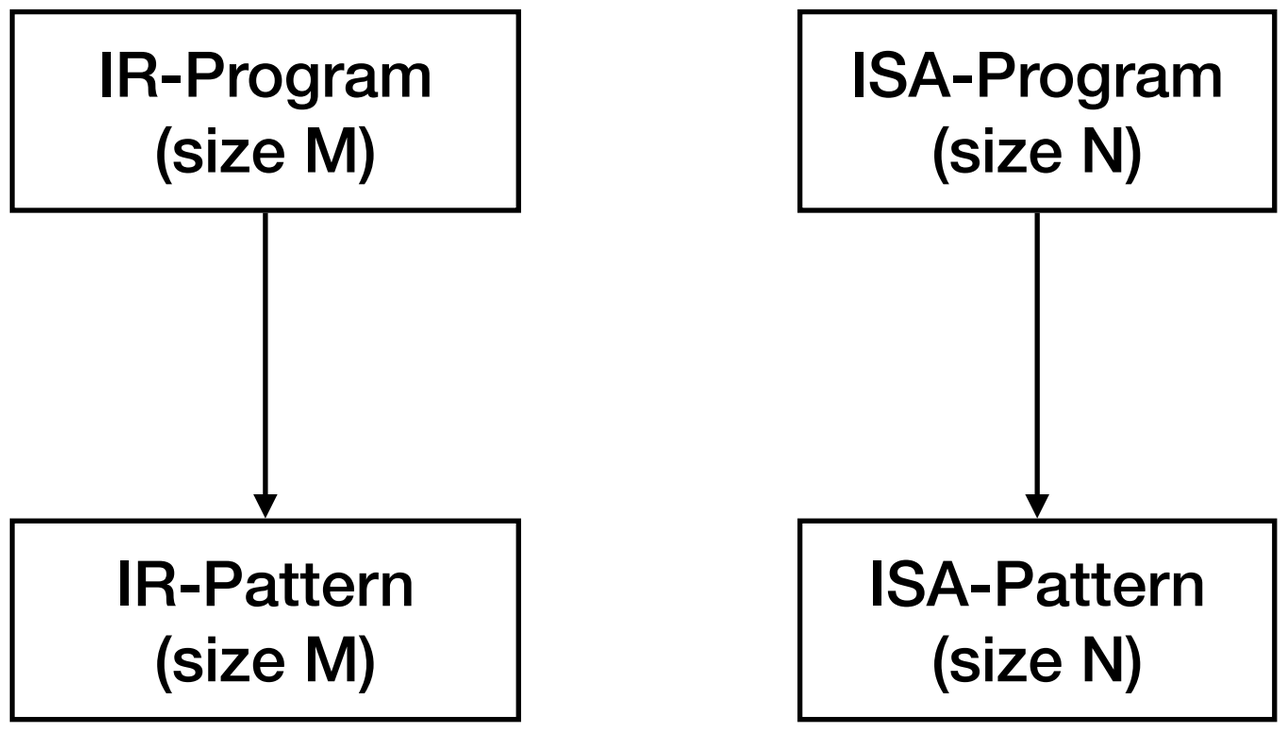

$$\exists\, \mathbf{IRProgram_m}\,,\, \mathbf{ISAProgram_n}\;\forall\, \mathbf{X}\,.$$
$$\mathbf{IRProgram_m(X) = ISAProgram_n(X)}$$

# Given a Satisfying Solution

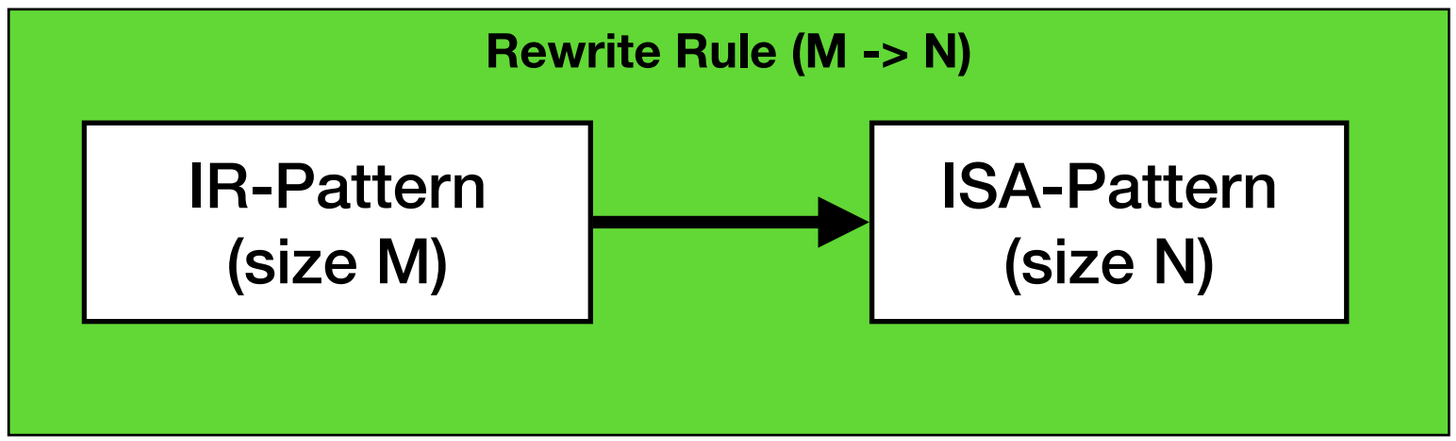
IR-Program
(size M)

ISA-Program
(size N)

# Programs are Translated to Patterns

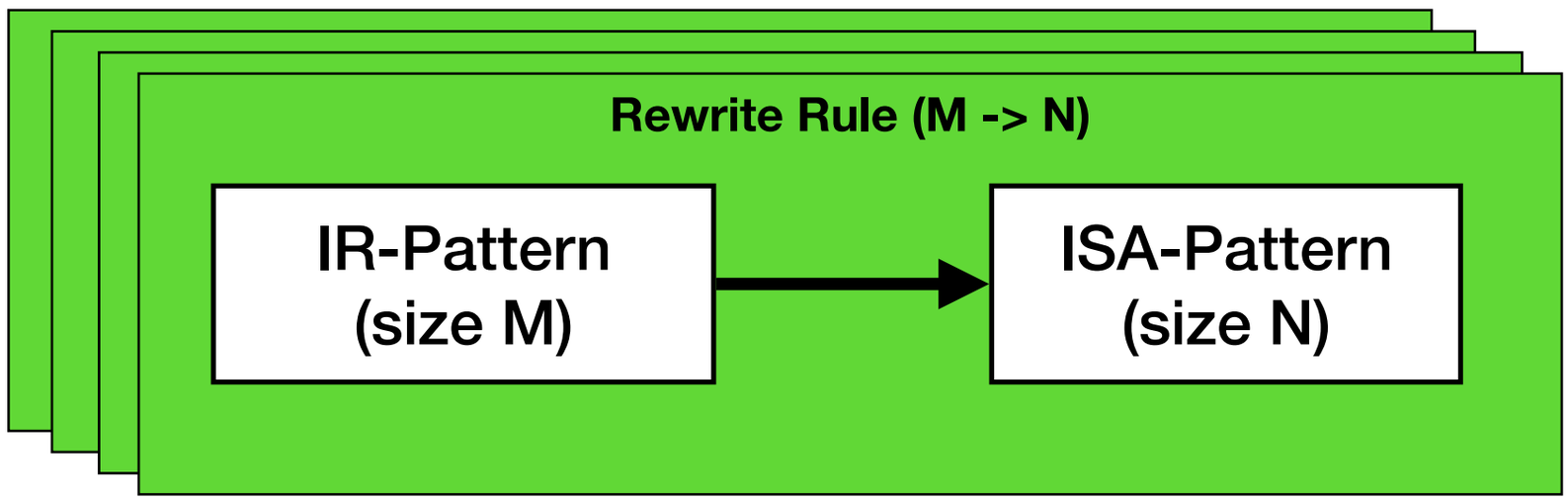


# Pair of Patterns Interpreted as Rewrite Rule

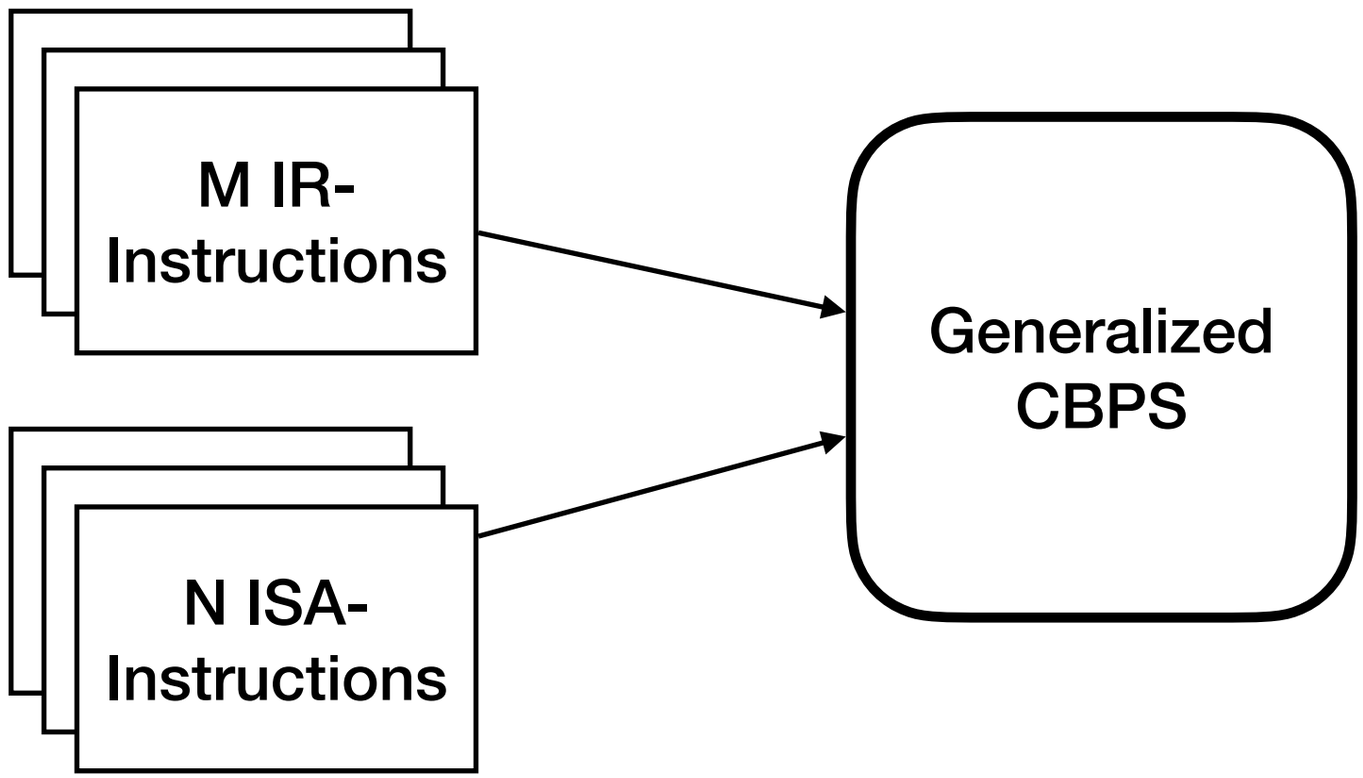


# Same query can produce many rewrite rules.

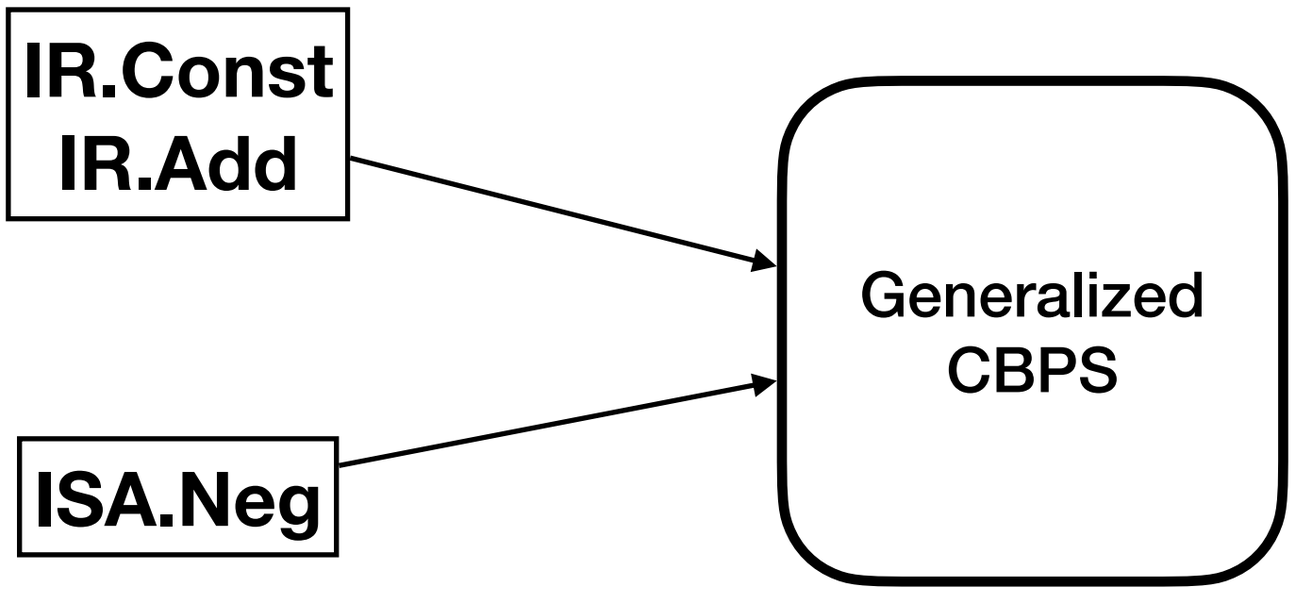
$$\exists\, \mathbf{IRProgram_m}\,, \mathbf{ISAProgram_n}\ \forall\, \mathbf{X}\,.$$
$$\mathbf{IRProgram_m(X) = ISAProgram_n(X)}$$



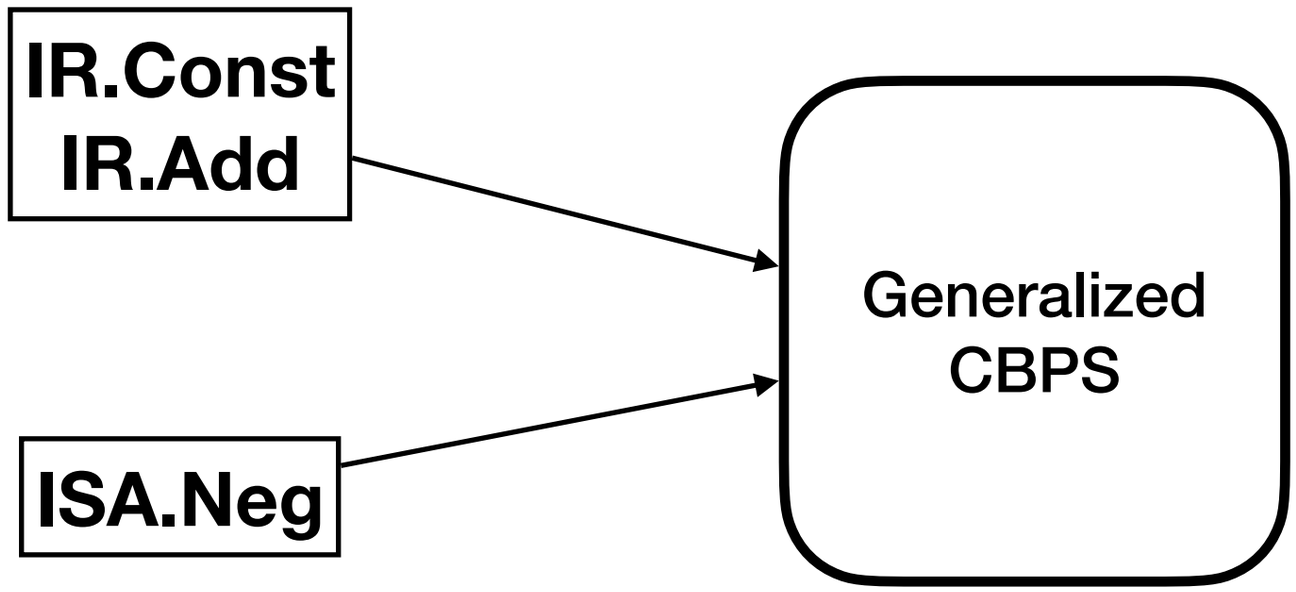
# How to Generate All Rules?
# Try all Combinations

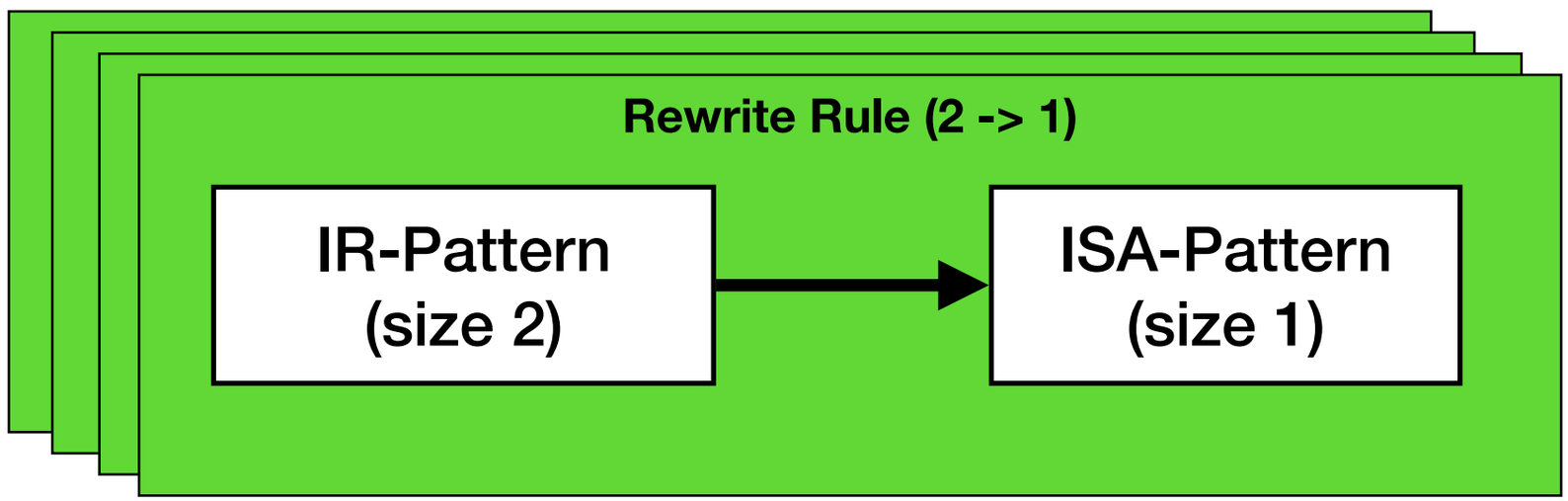


# How to Generate All Rules?
# Try all Combinations

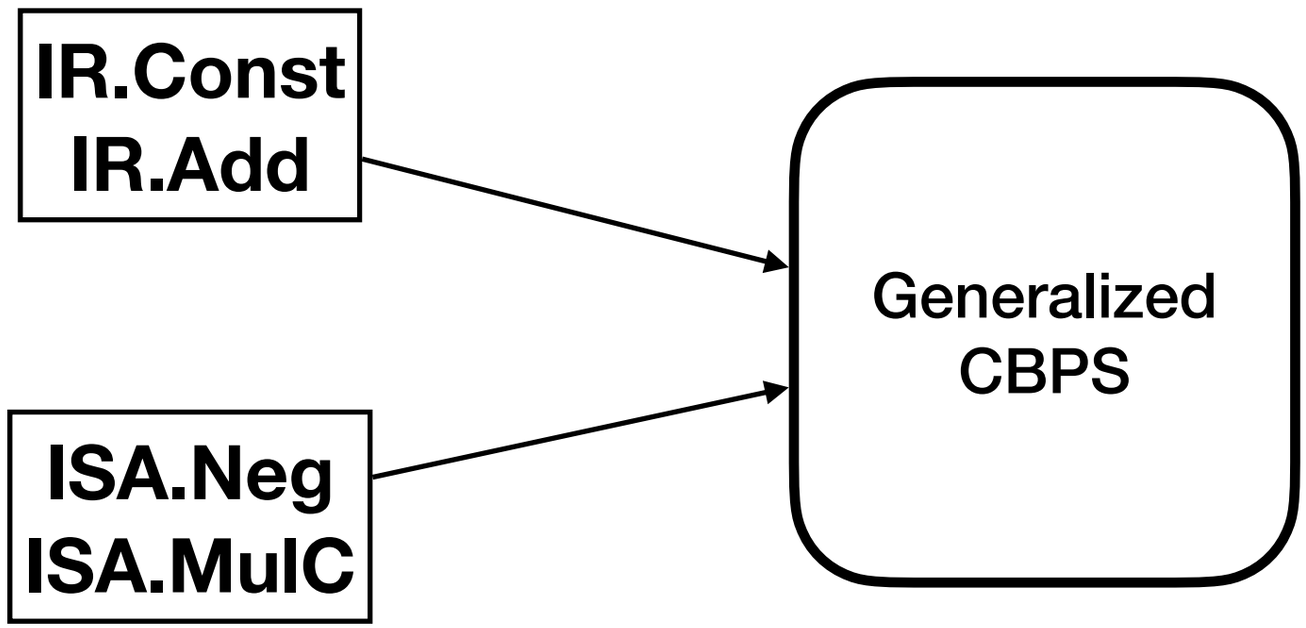


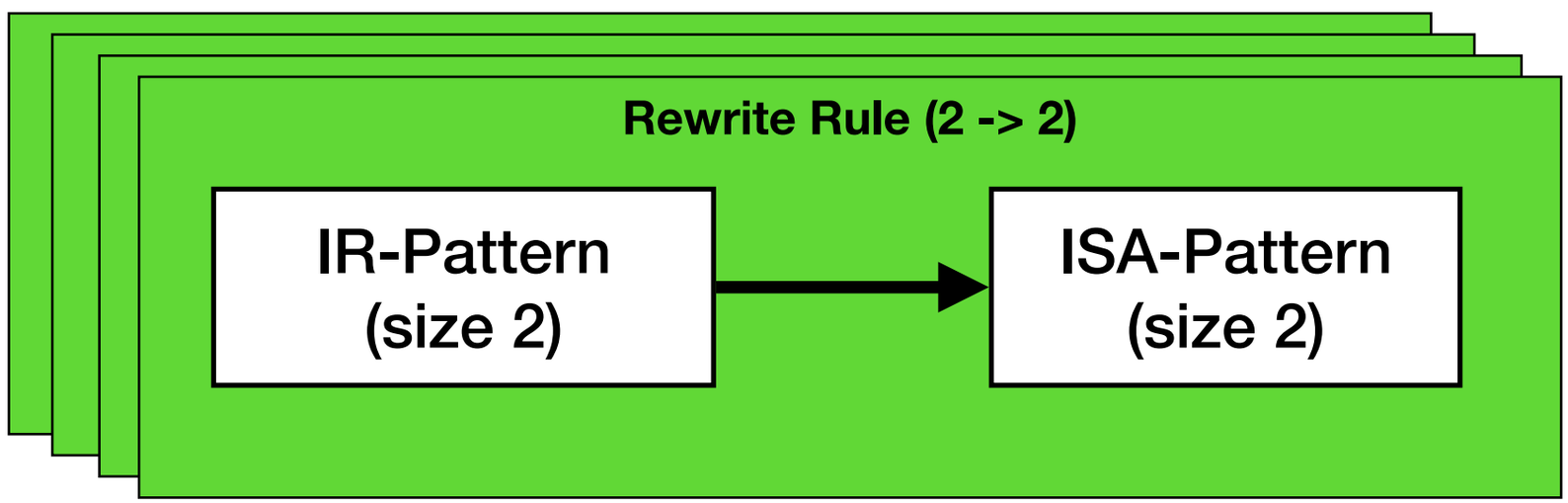
# How to Generate All Rules?
# Try all Combinations



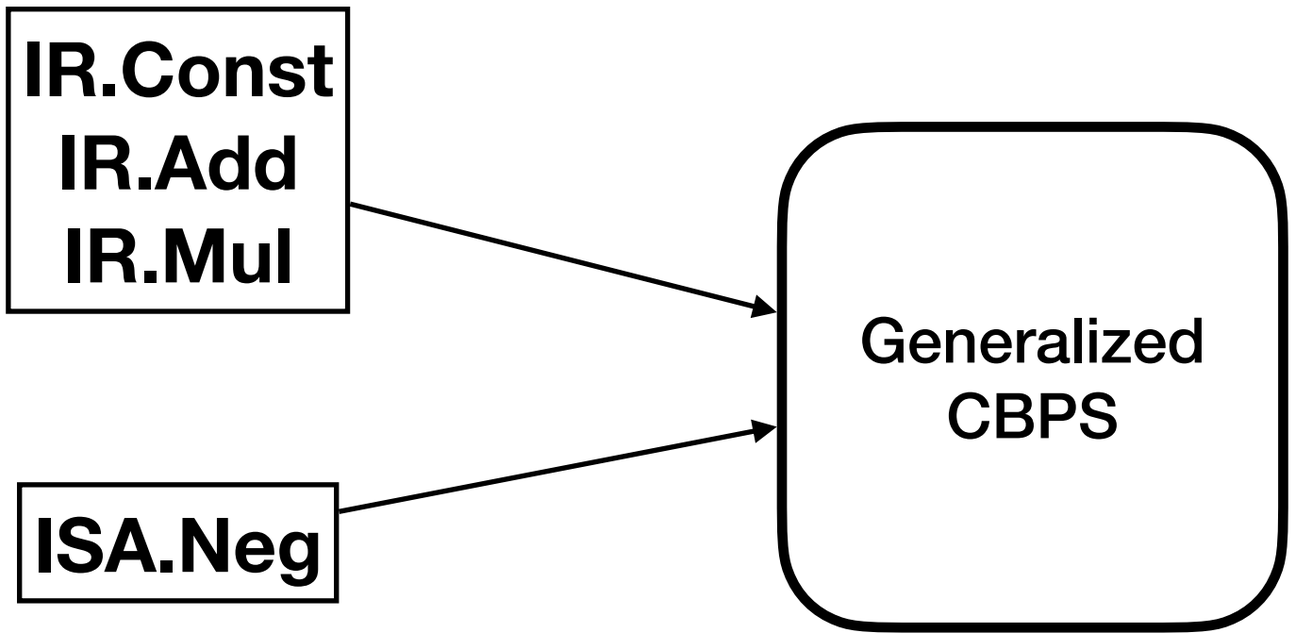



# How to Generate All Rules?
# Try all Combinations

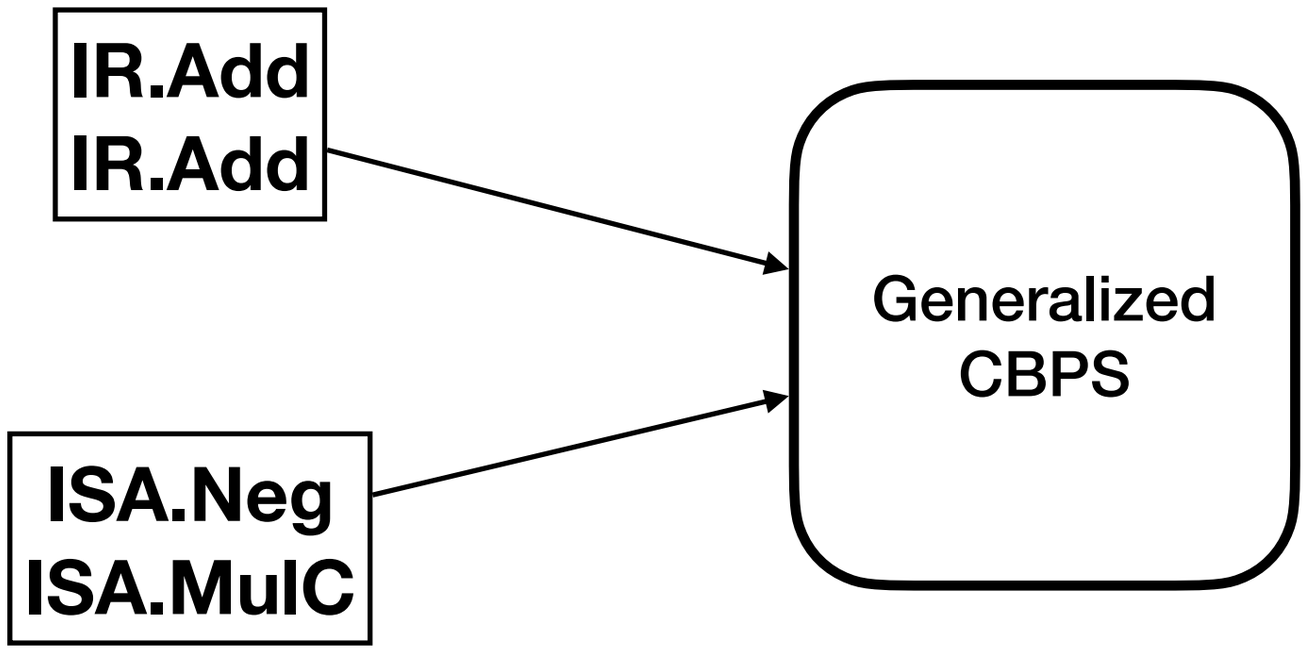


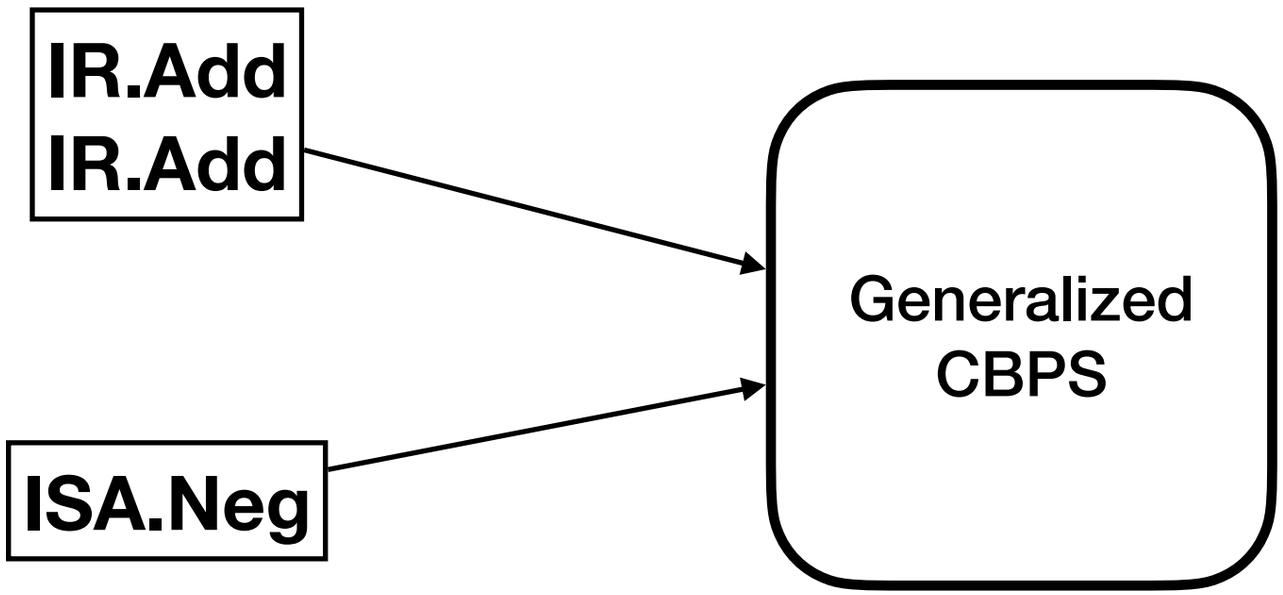
# How to Generate All Rules?
# Try all Combinations



# How to Generate All Rules?
# Try all Combinations



# How to Generate All Rules?
# Try all Combinations




▪ ▪ ▪

# Used GCBPS to generate all rewrite rules

| IR Instructions | ISA Instructions |
|---|---|
| IR.Const(C) | ISA.Neg(X) |
| IR.Add(X, Y) | ISA.Add(X, Y) |
| IR.Sub(X, Y) | ISA.MulC(X, C) |
| IR.Mul(X, Y) | ISA.Add3(X, Y, Z) |

# Number of Synthesized Rewrite Rules (Total: 1190)

| #IR \ #ISA | 1 | 2 |
|---|---|---|
| **1** | 5 | 6 |
| **2** | 223 | 284 |
| **3** | 155 | 517 |

# 1 IR -> 2 ISA

```
0| input x : BV[16]
1| input y : BV[16]
2| t0 = ir.sub(x,y)
3| return t0
```

=>

```
0| input x : BV[16]
1| input y : BV[16]
2| t0 = isa.neg(y)
3| t1 = isa.add(x,t0)
4| return t1
```

# 2 IR -> 1 ISA

```
0| input x : BV[16]
1| input C : CBV[16]
2| t0 = ir.const(C)
3| t1 = ir.mul(x,t0)
4| return t0
```

=>

```
0| input x : BV[16]
1| input C : CBV[16]
2| t0 = isa.mulC(x,C)
3| return t1
```

# 3 IR -> 2 ISA

```
0| input x : BV[16]
1| input y : BV[16]
2| input C : CBV[16]
3| t0 = ir.const(C)
4| t1 = ir.mul(x,t0)
5| t2 = ir.add(y,t1)
6| return t2
```

=>

```
0| input x : BV[16]
1| input y : BV[16]
2| input C : CBV[16]
3| t0 = isa.mulC(x,C)
4| t1 = isa.add(t0,y)
5| return t1
```

# Problem: Seems to be too many rules…

# Problem: Seems to be too many rules...

```
0|  input x : BV[16]
1|  input y : BV[16]
2|  t0 = ir.sub(x,y)
3|  return t0
```

=>

```
0|  input x : BV[16]
1|  input y : BV[16]
2|  t0 = isa.neg(y)
3|  t1 = isa.add(x,t0)
4|  return t1
```

```
0|  input x : BV[16]
1|  input y : BV[16]
2|  t0 = ir.sub(y,x)
3|  return t0
```

=>

```
0|  input x : BV[16]
1|  input y : BV[16]
2|  t0 = isa.neg(x)
3|  t1 = isa.add(y,t0)
4|  return t1
```

# X and Y are swapped

```
0|  input x : BV[16]
1|  input y : BV[16]
2|  t0 = ir.sub(x,y)
3|  return t0
```

=>

```
0|  input x : BV[16]
1|  input y : BV[16]
2|  t0 = isa.neg(y)
3|  t1 = isa.add(x,t0)
4|  return t1
```

```
0|  input x : BV[16]
1|  input y : BV[16]
2|  t0 = ir.sub(y,x)
3|  return t0
```

=>

```
0|  input x : BV[16]
1|  input y : BV[16]
2|  t0 = isa.neg(x)
3|  t1 = isa.add(y,t0)
4|  return t1
```

# X and Y are swapped

```
0|  input x : BV[16]
1|  input y : BV[16]
2|  t0 = ir.sub(x,y)
3|  return t0
```

=>

```
0|  input x : BV[16]
1|  input y : BV[16]
2|  t0 = isa.neg(y)
3|  t1 = isa.add(x,t0)
4|  return t1
```

```
0|  input x : BV[16]
1|  input y : BV[16]
2|  t0 = ir.sub(y,x)
3|  return t0
```

=>

```
0|  input x : BV[16]
1|  input y : BV[16]
2|  t0 = isa.neg(x)
3|  t1 = isa.add(y,t0)
4|  return t1
```

# Issue: All input permutations should be considered the same rule.

# Another one

```
0| input x : BV[16]
1| input y : BV[16]
2| t0 = ir.add(x,y)
3| return t0
```

=>

```
0| input x : BV[16]
1| input y : BV[16]
2| t0 = isa.add(x,y)
3| return t1
```

```
0| input x : BV[16]
1| input y : BV[16]
2| t0 = ir.add(x,y)
3| return t0
```

=>

```
0| input x : BV[16]
1| input y : BV[16]
2| t0 = isa.add(y,x)
3| return t0
```

# isa.add is Commutative

```
0| input x : BV[16]
1| input y : BV[16]
2| t0 = ir.add(x,y)
3| return t0
```

=>

```
0| input x : BV[16]
1| input y : BV[16]
2| t0 = isa.add(x,y)
3| return t1
```

```
0| input x : BV[16]
1| input y : BV[16]
2| t0 = ir.add(x,y)
3| return t0
```

=>

```
0| input x : BV[16]
1| input y : BV[16]
2| t0 = isa.add(y,x)
3| return t0
```

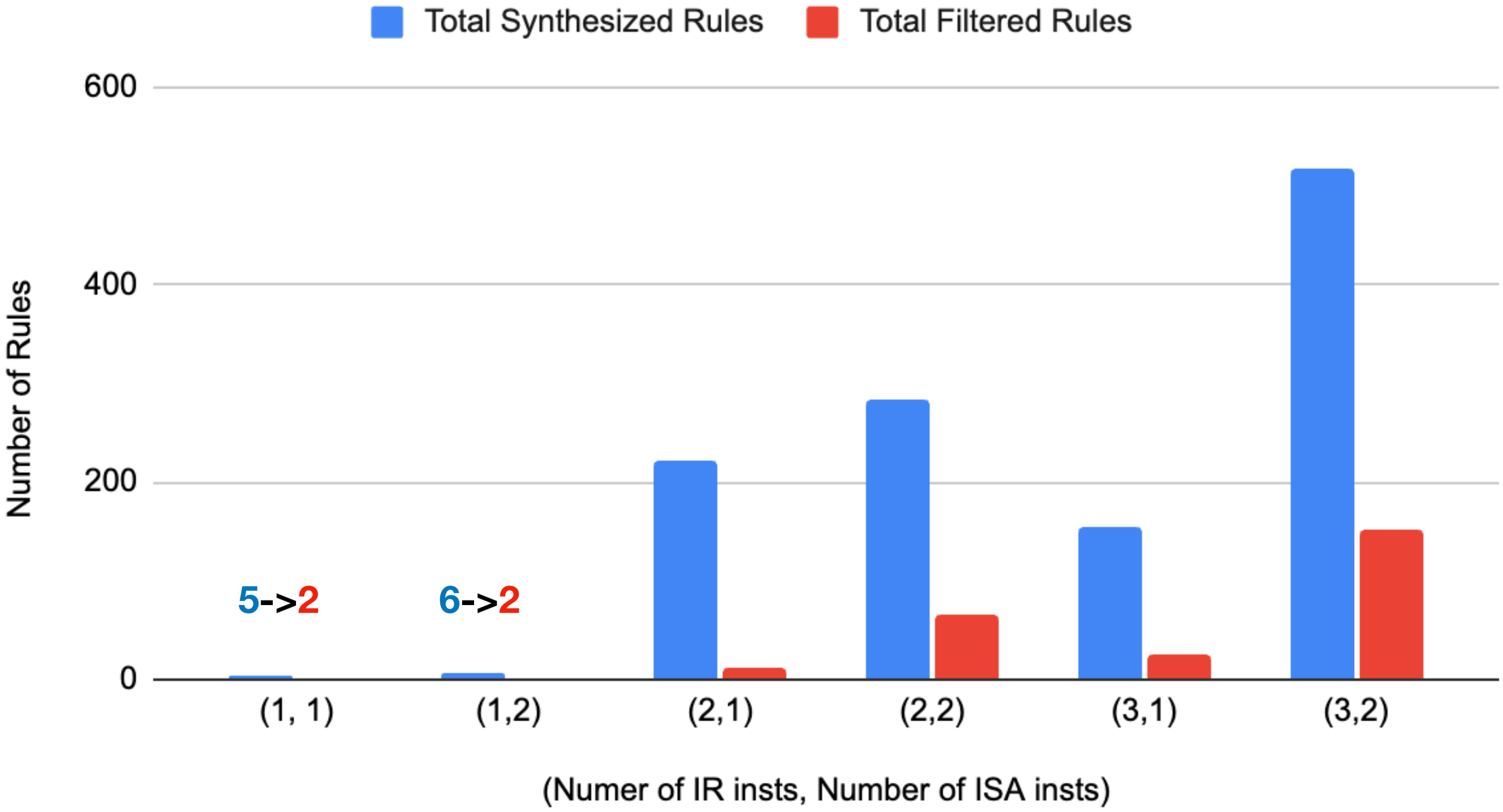
# Issue: All argument orderings of all commutative instructions should be considered the same rule.

# Post-Synthesis Filtering

- After synthesizing all rules, filter out all rules that are the same as another rule.

- Solution for related Rewrite Rule Synthesis work

# Comparison of Synthesized Rules to Filtered rules



# Post-Synthesis Filtering

- Major Performance implication!
  - Each call to CEGIS is expensive
  - Number of calls to CEGIS =

# Post-Synthesis Filtering

- Major Performance implication!
  - Each call to CEGIS is expensive
  - Number of calls to CEGIS =
    - **Number of synthesized rules** (SAT)+ number CBPS queries (UNSAT)

# Post-Synthesis Filtering

- Major Performance implication!
  - Each call to CEGIS is expensive
  - Number of calls to CEGIS =
    - **Number of synthesized rules** (SAT)+ number CBPS queries (UNSAT)
  - Better idea: Only synthesize unique rules

# Canonicalization Constraints

- I created two additional constraints
  - Canonicalize to one possible input permutation
  - Canonicalize to one possible argument ordering for each commutative instruction
- Constraints are added to the synthesis query

# Only one solution possible with additional constraint

```
0|  input x : BV[16]
1|  input y : BV[16]
2|  t0 = ir.add(x,y)
3|  return t0
```

=>

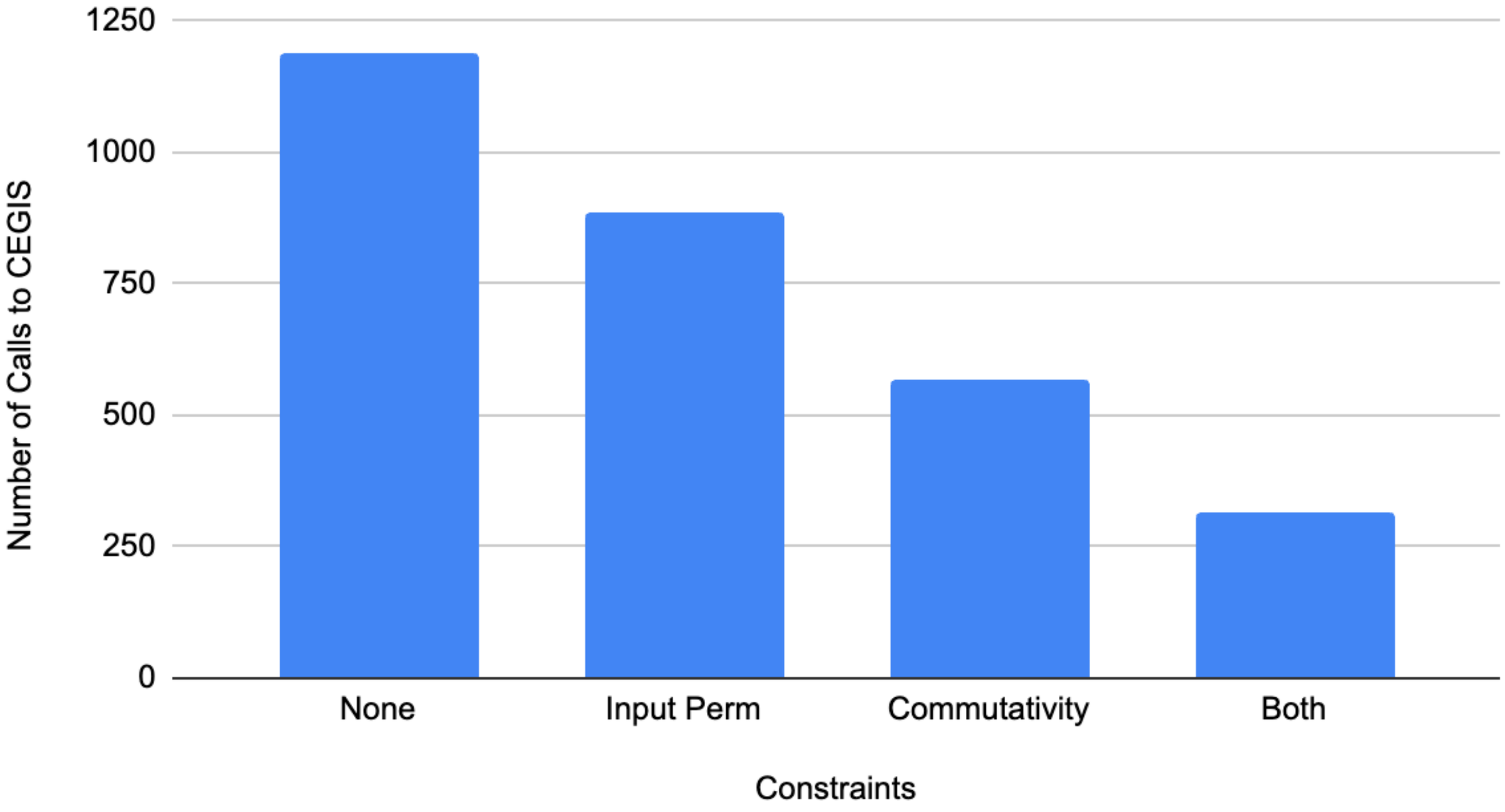
```
0|  input x : BV[16]
1|  input y : BV[16]
2|  t0 = isa.add(x,y)
3|  return t1
```

```
0|  input x : BV[16]
1|  input y : BV[16]
2|  t0 = ir.add(x,y)
3|  return t0
```

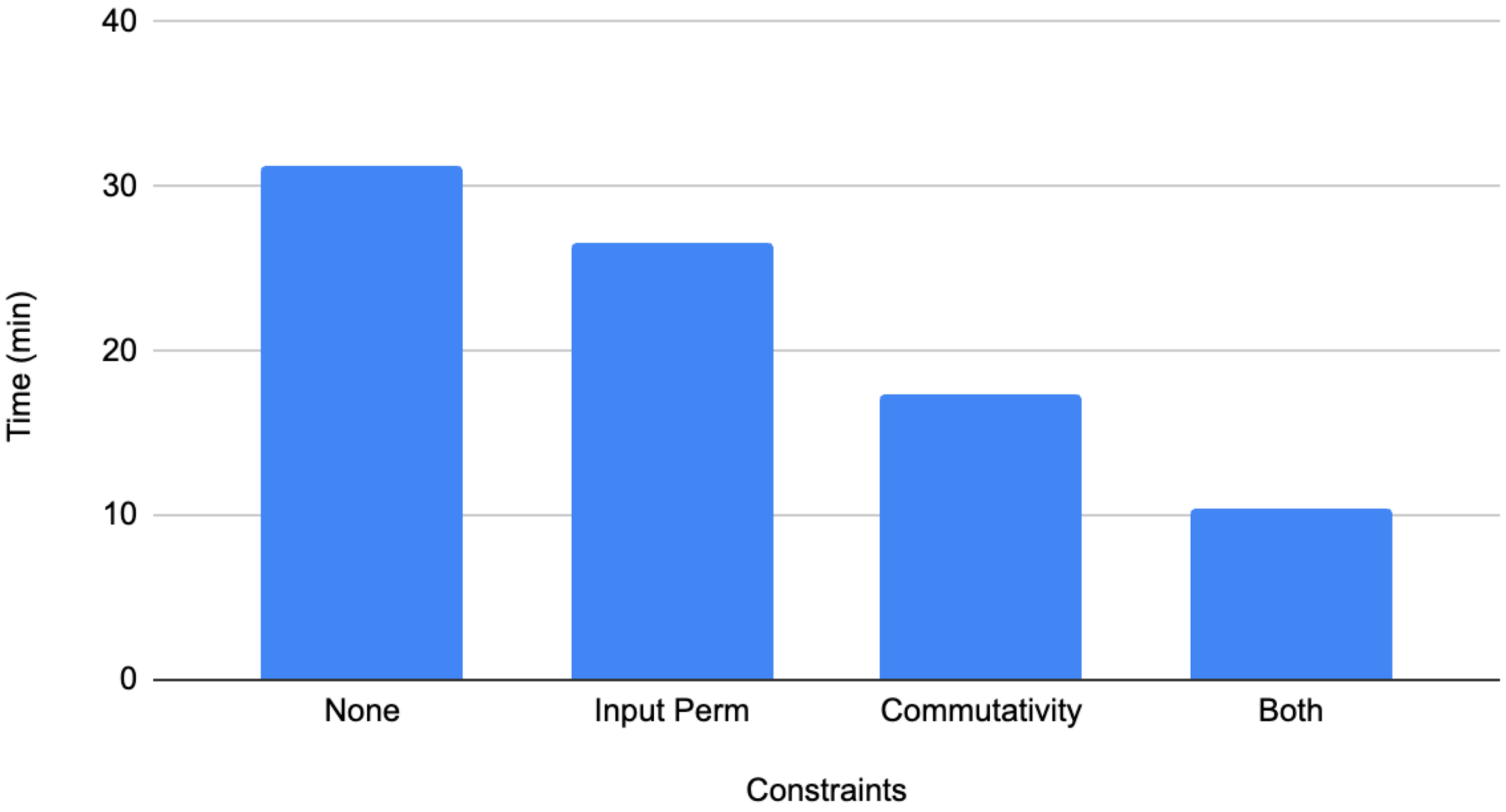
->

```
0|  input x : BV[16]
1|  input y : BV[16]
2|  t0 = isa.add(y,x)
3|  return t0
```

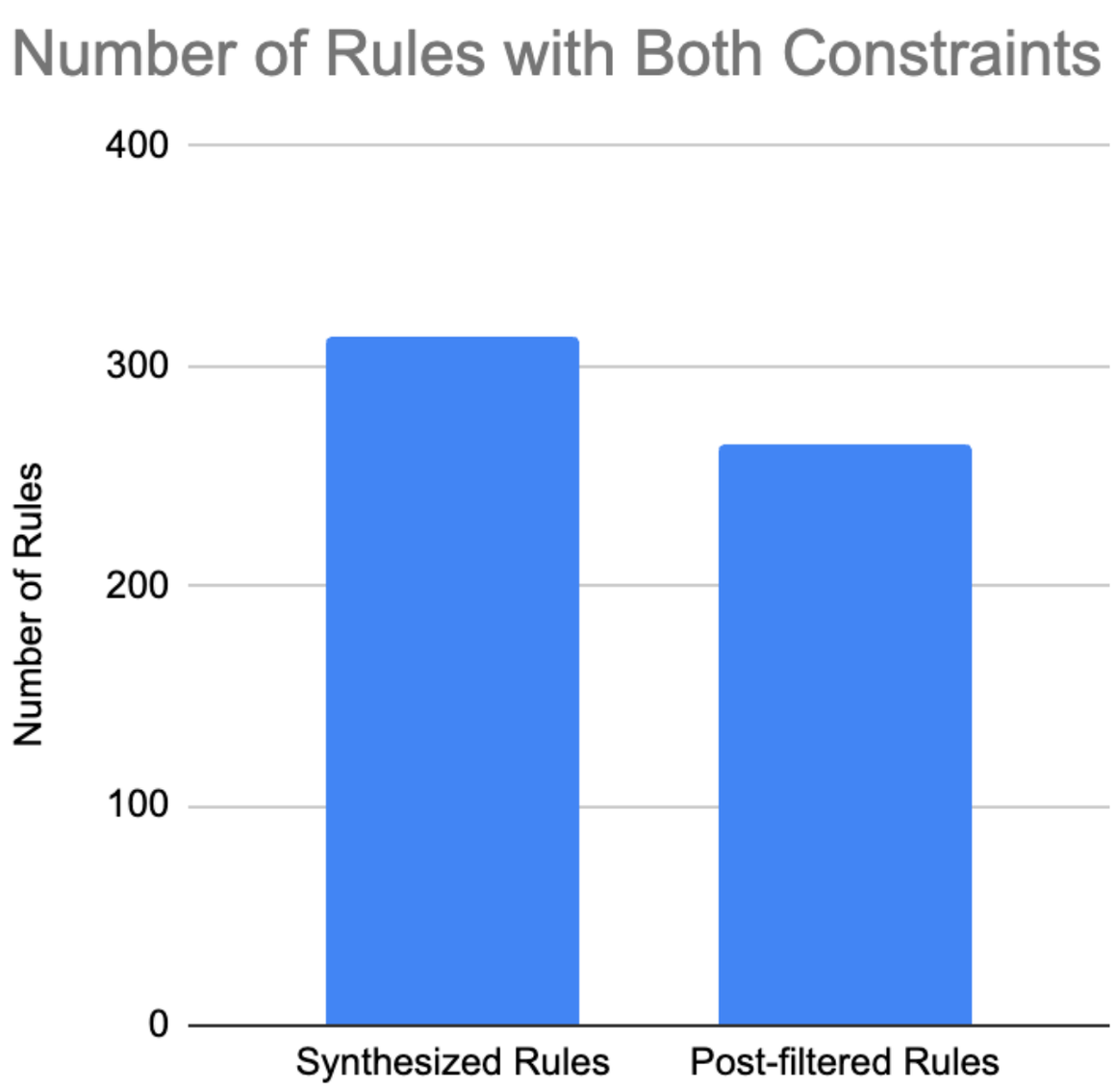
# Number of Calls to CEGIS with Constraints Enabled



# Total Synthesis Time with Constraints Enabled



# Still Missing some Rule Equivalences



```
0|  input x,y,z,w
1|  t0 = ir.add(x,y)
2|  t1 = ir.add(z,w)
3|  t2 = ir.add(t0,t1)
4|  return t2
```

=>

```
0|  input x,y,z,w
1|  t0 = isa.add(x,y)
2|  t1 = isa.add3(t0,z,w)
3|  return t1
```

.....

```
0|  input x,y,z,w
1|  t0 = ir.add(z,w)
2|  t1 = ir.add(x,y)
3|  t2 = ir.add(t0,t1)
4|  return t2
```

=>

```
0|  input x,y,z,w
1|  t0 = isa.add(x,y)
2|  t1 = isa.add3(t0,z,w)
3|  return t1
```

# Different program orders

```
0|  input x,y,z,w
1|  t0 = ir.add(x,y)
2|  t1 = ir.add(z,w)
3|  t2 = ir.add(t0,t1)
4|  return t2
```

=>

```
0|  input x,y,z,w
1|  t0 = isa.add(x,y)
2|  t1 = isa.add3(t0,z,w)
3|  return t1
```

```
0|  input x,y,z,w
1|  t0 = ir.add(z,w)
2|  t1 = ir.add(x,y)
3|  t2 = ir.add(t0,t1)
4|  return t2
```

=>

```
0|  input x,y,z,w
1|  t0 = isa.add(x,y)
2|  t1 = isa.add3(t0,z,w)
3|  return t1
```

# Different program orders

```
0|  input x,y,z,w
1|  t0 = ir.add(x,y)
2|  t1 = ir.add(z,w)
3|  t2 = ir.add(t0,t1)
4|  return t2
```

=>

```
0|  input x,y,z,w
1|  t0 = isa.add(x,y)
2|  t1 = isa.add3(t0,z,w)
3|  return t1
```

```
0|  input x,y,z,w
1|  t0 = ir.add(z,w)
2|  t1 = ir.add(x,y)
3|  t2 = ir.add(t0,t1)
4|  return t2
```

=>

```
0|  input x,y,z,w
1|  t0 = isa.add(x,y)
2|  t1 = isa.add3(t0,z,w)
3|  return t1
```

# Other Equivalencies

- Equivalencies with no canonicalization constraint:
  - Program Orderings
  - Commutative Programs

# Incremental Generative Filtering

- Tactic: Incremental Generative Filtering
  - After synthesizing one rule, programmatically enumerate all satisfying solutions equivalent to rule and exclude
  - Programatic enumeration faster than calling CEGIS. (Results in progress…)

# But wait there’s more…

```
0|  input x
1|  t0 = ir.add(x,x)
2|  return t0
```

=>

```
0|  input x
1|  t0 = isa.add(x,x)
2|  return t0
```

.....

```
0|  input x
1|  input y
2|  t0 = ir.add(x,y)
3|  return t0
```

=>

```
0|  input x
1|  input y
2|  t0 = isa.add(x,y)
3|  return t0
```

# Specialized Rewrite Rules

```
0| input x
1| t0 = ir.add(x,x)
2| return t0
```

=>

```
0| input x
1| t0 = isa.add(x,x)
2| return t0
```

```
0| input x
1| input y
2| t0 = ir.add(x,y)
3| return t0
```

=>

```
0| input x
1| input y
2| t0 = isa.add(x,y)
3| return t0
```

# Specialized Rules are Redundant

```
0| input x
1| t0 = ir.add(x,x)
2| return t0
```

```
0| input x
1| t0 = isa.add(x,x)
2| return t0
```

```
0| input x
1| input y
2| t0 = ir.add(x,y)
3| return t0
```

=>

```
0| input x
1| input y
2| t0 = isa.add(x,y)
3| return t0
```

```
0|  input x, y
1|  t0 = ir.add(x,y)
2|  return t0
```

=>

```
0|  input x, y
1|  t0 = isa.add(x,y)
2|  return t0
```

```
0|  input x, y, z
1|  t0 = ir.add(x,y)
2|  t1 = ir.add(t0,z)
3|  return t1
```

=>

```
0|  input x, y, z
2|  t0 = isa.add3(x,y,z)
3|  return t0
```

```
0|  input x,y,z,w
1|  t0 = ir.add(x,y)
2|  t1 = ir.add(z,w)
3|  t2 = ir.add(t0,t1)
4|  return t2
```

=>

```
0|  input x,y,z,w
1|  t0 = isa.add(x,y)
2|  t1 = isa.add3(t0,z,w)
3|  return t1
```

# Composite Rewrite Rules

```
0| input x, y
1| t0 = ir.add(x,y)
2| return t0
```

=>

```
0| input x, y
1| t0 = isa.add(x,y)
2| return t0
```

```
0| input x, y, z
1| t0 = ir.add(x,y)
2| t1 = ir.add(t0,z)
3| return t1
```

=>

```
0| input x, y, z
2| t0 = isa.add3(x,y,z)
3| return t0
```

```
0| input x,y,z,w
1| t0 = ir.add(x,y)
2| t1 = ir.add(z,w)
3| t2 = ir.add(t0,t1)
4| return t2
```

=>

```
0| input x,y,z,w
1| t0 = isa.add(x,y)
2| t1 = isa.add3(t0,z,w)
3| return t1
```

# Composite Rewrite Rules

```
0| input x, y
1| t0 = ir.add(x,y)
2| return t0
```

=>

```
0| input x, y
1| t0 = isa.add(x,y)
2| return t0
```

```
0| input x, y, z
1| t0 = ir.add(x,y)
2| t1 = ir.add(t0,z)
3| return t1
```

=>

```
0| input x, y, z
2| t0 = isa.add3(x,y,z)
3| return t0
```

```
0| input x,y,z,w
1| t0 = ir.add(x,y)
2| t1 = ir.add(z,w)
3| t2 = ir.add(t0,t1)
4| return t2
```

=>

```
0| input x,y,z,w
1| t0 = isa.add(x,y)
2| t1 = isa.add3(t0,z,w)
3| return t1
```

# Composite Rules are Redundant

```
0| input x, y
1| t0 = ir.add(x,y)
2| return t0
```

=>

```
0| input x, y
1| t0 = isa.add(x,y)
2| return t0
```

```
0| input x, y, z
1| t0 = ir.add(x,y)
2| t1 = ir.add(t0,z)
3| return t1
```

=>

```
0| input x, y, z
2| t0 = isa.add3(x,y,z)
3| return t0
```

```
0| input x,y,z,w
1| t0 = ir.add(x,y)
2| t1 = ir.add(z,w)
3| t2 = ir.add(t0,t1)
4| return t2
```

=>

```
0| input x,y,z,w
1| t0 = isa.add(x,y)
2| t1 = isa.add3(t0,z,w)
3| return t1
```

# Redundant Rewrite Rules

- Redundancies caused by:
  - Specialized Rules
  - Composite Rules
  - Rules guaranteed to have a higher cost.

# Preemptive Generative Filtering

- Tactic: Preemptive Generative Filtering
  - Before calling CEGIS, using existing rules, programmatically enumerate and exclude all redundant ones.
  - Programatic enumeration is faster than calling CEGIS. (Results in progress…)

# High Level View

- Goal: Generate all N -> M rewrite rules
- Uses generalized Component-based Program Synthesis.
- Works, but is inefficient!
  - Improve performance by removing equivalencies and redundancies.
    - Equivalent rules filtered using canonicalization constraints and “Incremental Generative Filtering”
    - Redundant rules filtered using “Preemptive Generative Filtering”
  - All calls to CEGIS will only synthesize a new unique rule.

# Questions?