Towards automated super-optimization for Taichi using Equality Saturation

Deyuan (Mike) He
Department of Computer Science, Princeton University

Background

Term rewriting is extremely common in Compilers (example *manual* rewrites from **alg_simp.cpp**):

a * 2	⇒	a << 1
a / pot	⇒	a >> log2(pot)
a / const	⇒	a * (1 / const)

Background

However, determining the order of applying rewrite rules is **HARD**!

Background

1. Common Subexpression Elimination (CSE)
2. Associativity of Matrix Multiplication (Assoc)

$$M_1 * M_2 * M_3$$

$$M_1 * M_2 * M_4$$

Cost: 2NMK + NKP + NKQ multiplications

Background

CSE Rewrite

$M_1 * M_2 * M_3$

$M_1 * M_2 * M_4$

$M_{12} = M_1 * M_2$ NMK

$M_{12} * M_3$ NKP

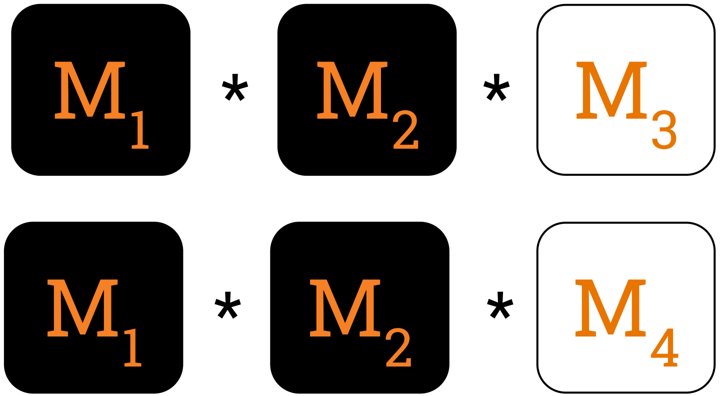
$M_{12} * M_4$ NKQ

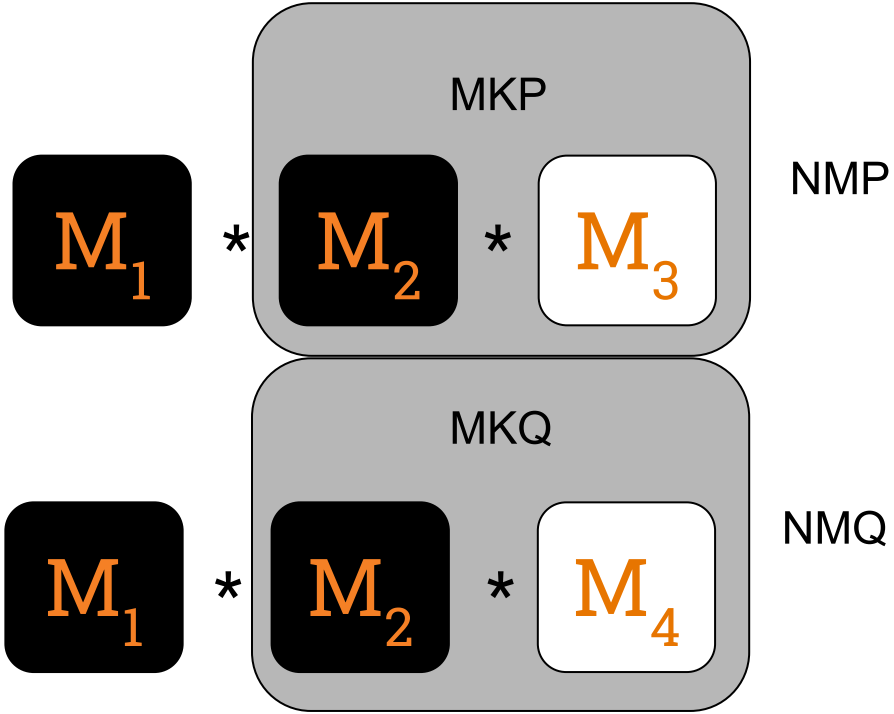
2NMK + NKP + NKQ multiplications

NMK + NKP + NKQ multiplications

Background

Assoc Rewrite





2NMK + NKP + NKQ multiplications

NMP + NMQ + MKP + MKQ multiplications

Background

Case 1 (Associativity better than CSE):
NMK + NKP + NKQ > NMP + NMQ + MKP + MKQ

e.g.
N = 2 M = 2 K = 8 P = 2 Q = 2

Before optimization: 128
⇒ CSE(96) > Assoc(80)

Background

Case 1 (CSE better than Associativity):
NMK + NKP + NKQ < NMP + NMQ + MKP + MKQ

e.g.
N = 2 M = 16 K = 4 P = 1 Q = 1

Before optimization: 544
⇒ CSE(144) < Assoc(192)

Background

Compilers may have **hundreds of passes**.

How to determine the order to ensure the product program is ***Optimal (or close)*** ?

Background

Compilers may have **hundreds of passes**.

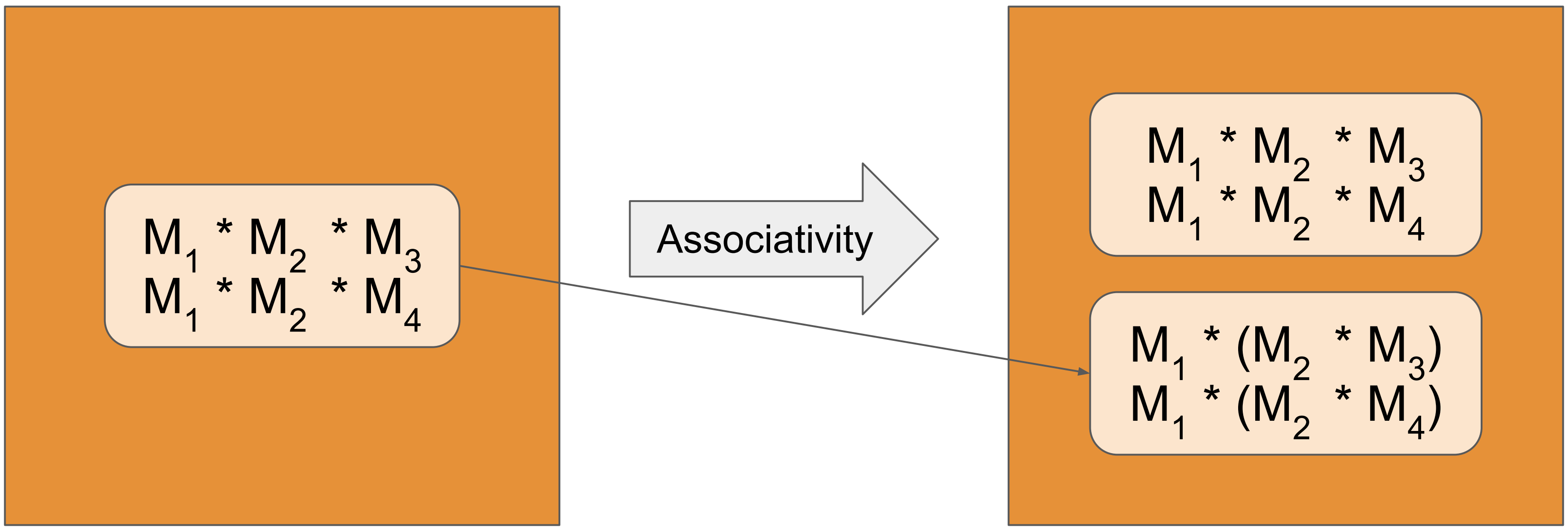
How to determine the order to ensure the product program is ***Optimal (or close)*** ?

Interleaving Passes? Phase Ordering Problem?

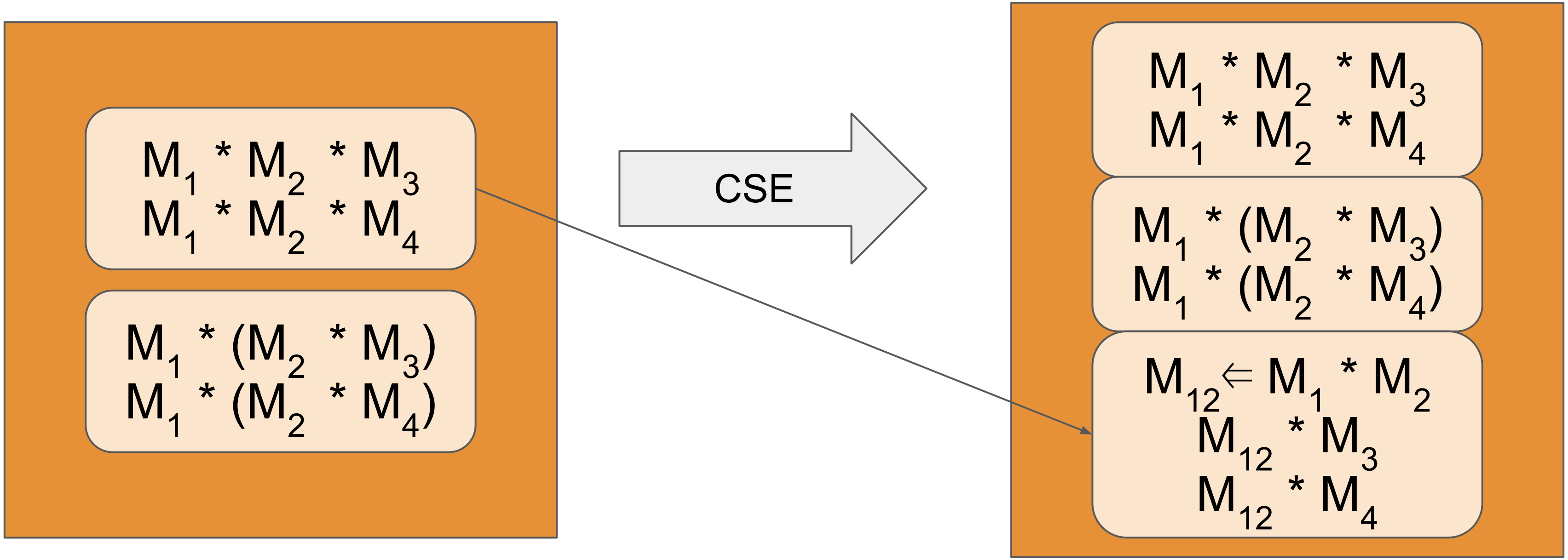
Equality Saturation

Equality Saturation (EqSat) is a technique to solve this problem by memoizing **all** the equivalences discovered by rewrite rules

Phase 1: Execute Rewrites



Phase 1: Execute Rewrites



Phase 1: Execute Rewrites

$M_1 * M_2 * M_3$
$M_1 * M_2 * M_4$

$M_1 * (M_2 * M_3)$
$M_1 * (M_2 * M_4)$

$M_{12} \Leftarrow M_1 * M_2$
$M_{12} * M_3$
$M_{12} * M_4$

Saturation: no more equivalence can be found by applying the rewrite rules (in *any* order)

This is the end of Phase 1

Phase 2: Extraction

$M_1 * M_2 * M_3$
$M_1 * M_2 * M_4$

$M_1 * (M_2 * M_3)$
$M_1 * (M_2 * M_4)$

$M_{12} \Leftarrow M_1 * M_2$
$M_{12} * M_3$
$M_{12} * M_4$

Extraction: select the optimal term from the candidate set using a cost model

E.g.: number of multiplications

Phase 2: Extraction

$$M_1 * M_2 * M_3$$
$$M_1 * M_2 * M_4$$

$$M_1 * (M_2 * M_3)$$
$$M_1 * (M_2 * M_4)$$

$$M_{12} \Leftarrow M_1 * M_2$$
$$M_{12} * M_3$$
$$M_{12} * M_4$$

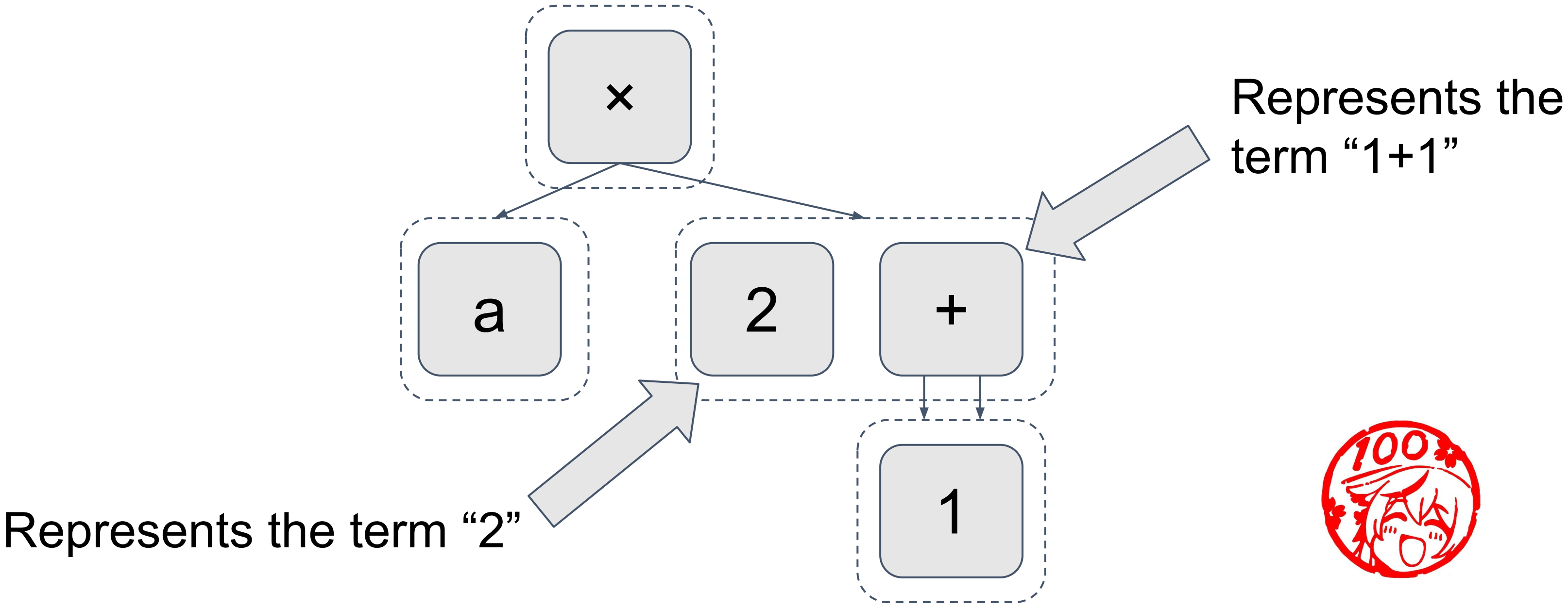
Extraction: select the optimal term from the candidate set using a cost model

E.g.: number of multiplications

Efficient Implementations?

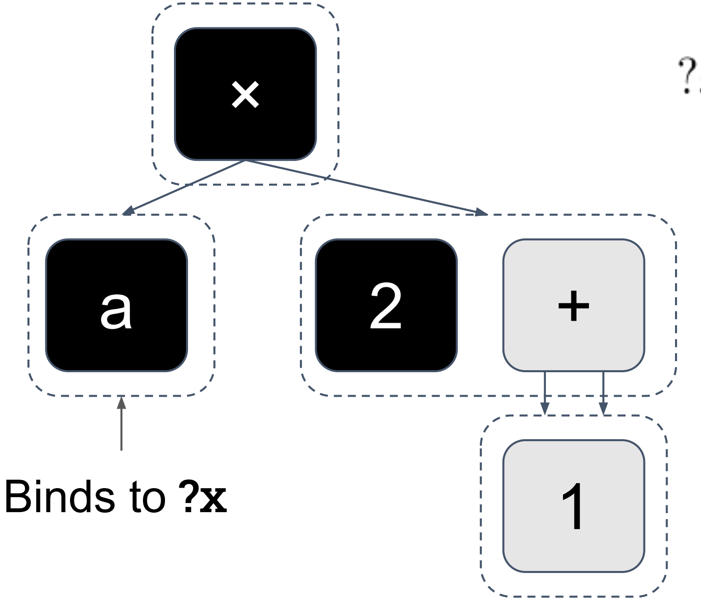
egg: Fast, Extensible Equality Saturation on EGraphs

1. E-Classes (dashed boxes): A set of equivalent terms
2. E-Nodes (solid boxes): Operators, variables or literals



Rewrite Rules

Syntactic Rewrites: an initial pattern and a target pattern

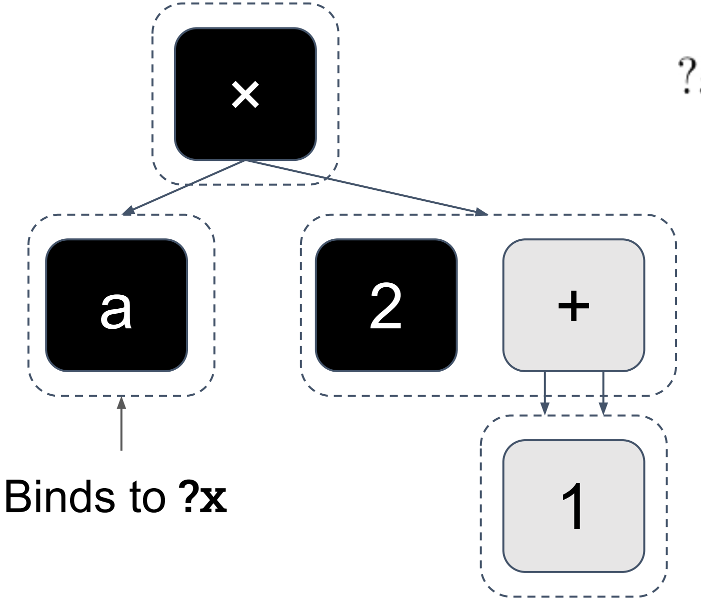


$$?x \times 2 \Rightarrow ?x << 1$$

Apply

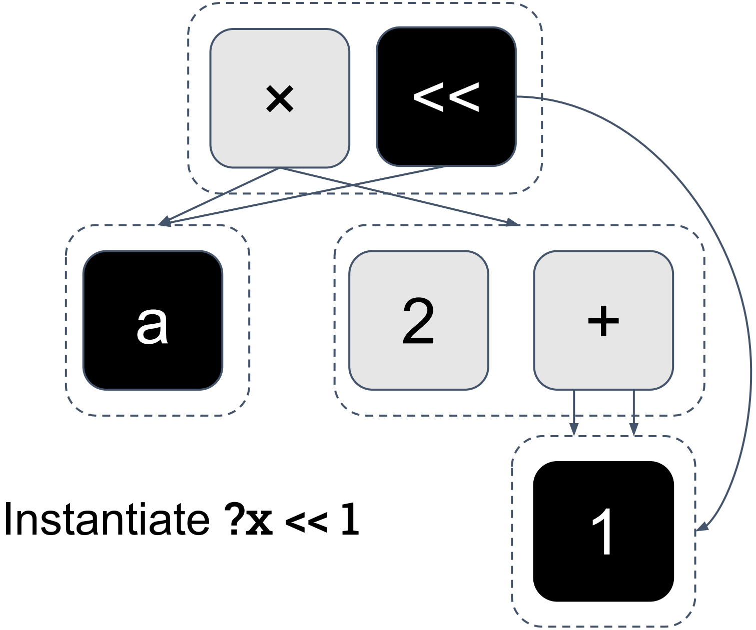
Rewrite Rules

Syntactic Rewrites: an initial pattern and a target pattern



$$?x \times 2 \Rightarrow ?x << 1$$

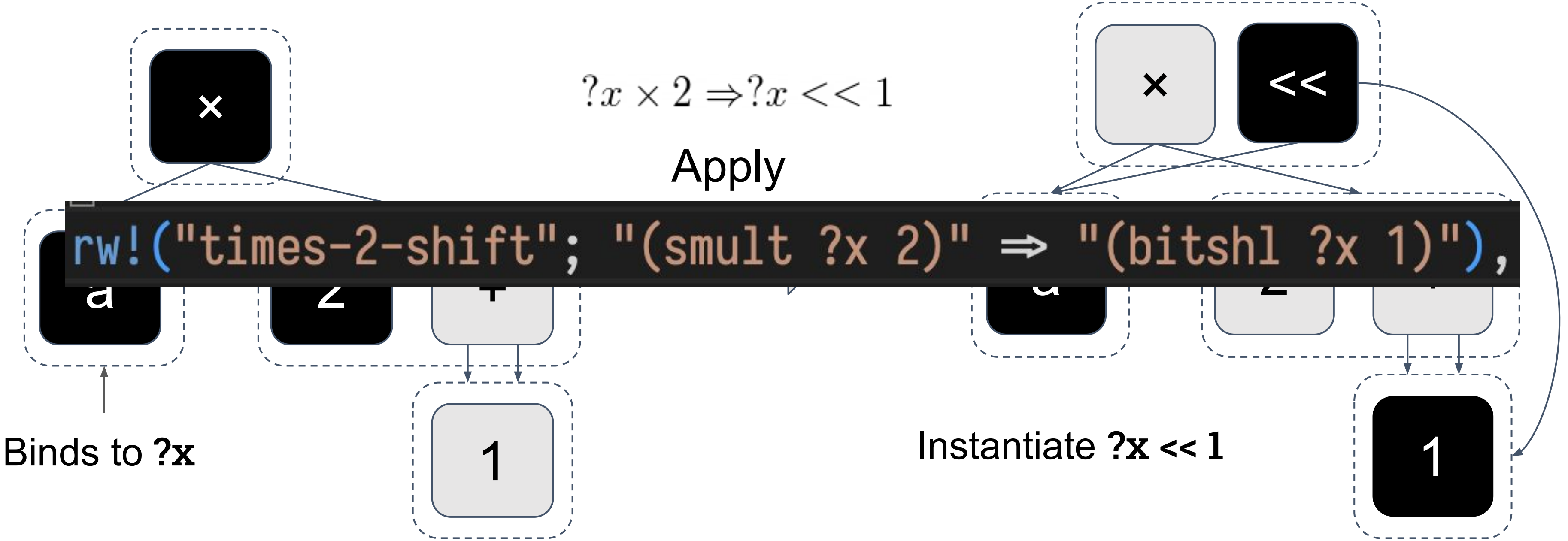
Apply



Rewrite Rules

Syntactic Rewrites: an initial pattern and a target pattern

$$?x \times 2 \Rightarrow ?x << 1$$




```
rw!("times-2-shift"; "(smult ?x 2)" => "(bitshl ?x 1)"),
```

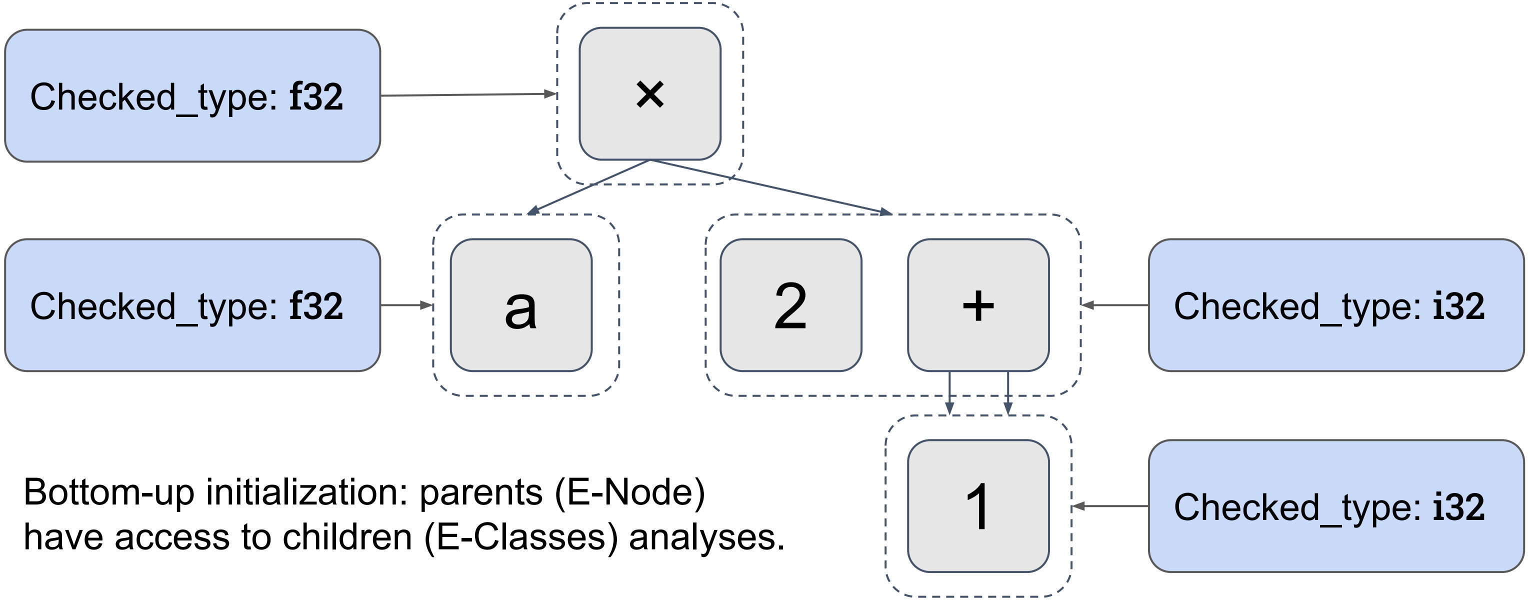
E-Class Analysis

Rewrite rules are **syntactic**, meaning that it is not always valid in terms of **semantics**

$?x\ /\ ?x \Rightarrow 1$	if **?x** does not evaluate to 0
$pow(2, ?x) \Rightarrow 1 \ll ?x$	if **?x** is an integer
$d(?c) \Rightarrow 0$	if **?c** is a constant

E-Class Analysis

E-Class Analysis: fully-customizable program analysis data attached to EClasses. E.g. Type checking / inference



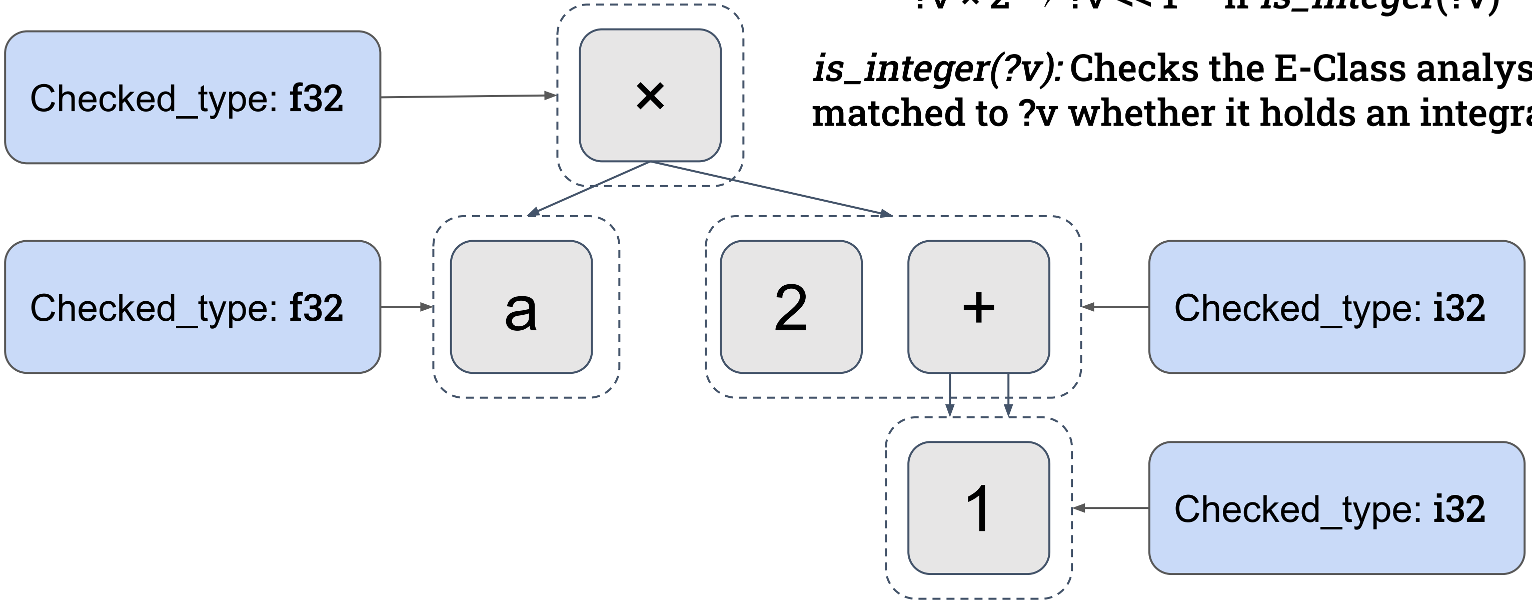
Bottom-up initialization: parents (E-Node) have access to children (E-Classes) analyses.

E-Class Analysis

E-Class Analysis enables conditional rewrites

$?v \times 2 \Rightarrow ?v \ll 1$ if *is_integer*(?v)

is_integer(?v): **Checks the E-Class analysis matched to ?v whether it holds an integral value.**

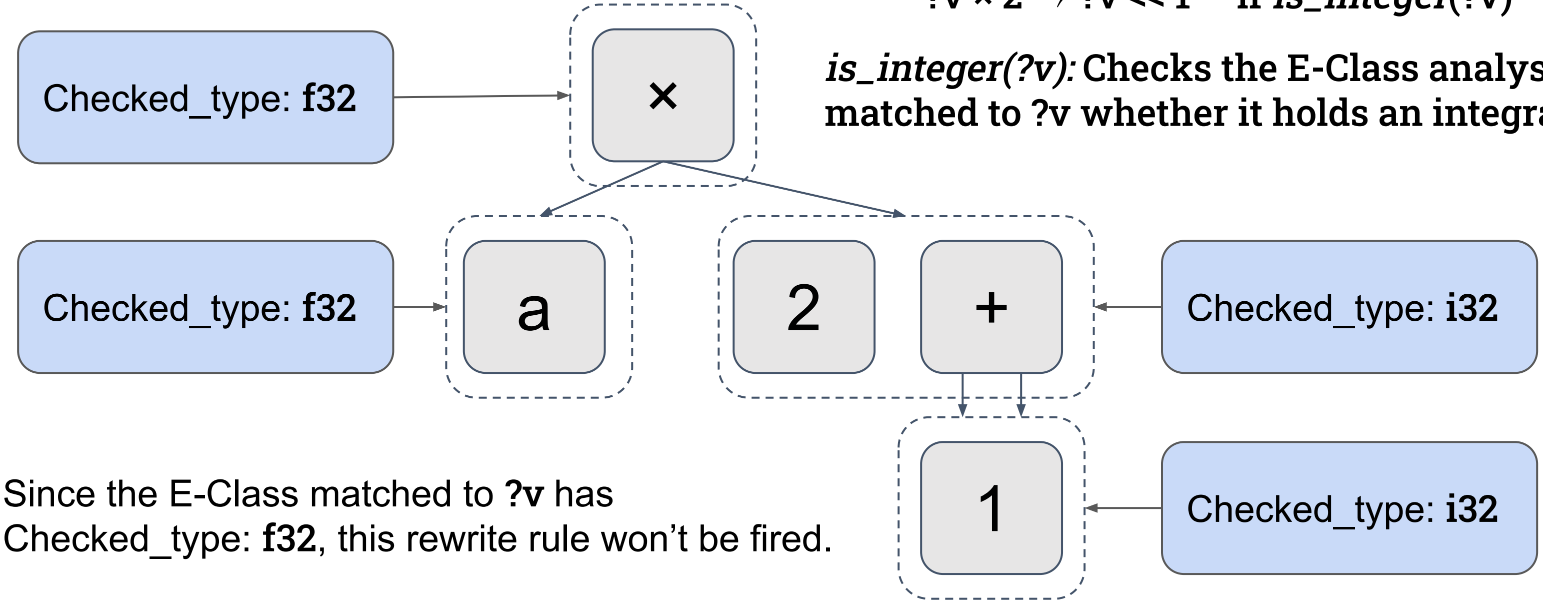


E-Class Analysis

E-Class Analysis enables conditional rewrites

$?v \times 2 \Rightarrow ?v << 1$ if *is_integer*(?v)

is_integer(?v): **Checks the E-Class analysis matched to ?v whether it holds an integral value.**



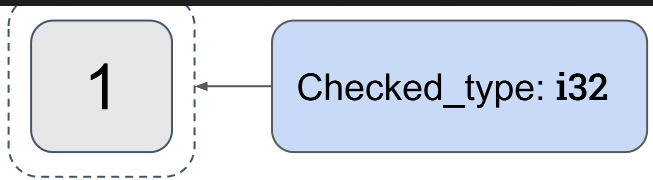
Since the E-Class matched to **?v** has Checked_type: **f32**, this rewrite rule won't be fired.

E-Class Analysis

E-Class Analysis enables conditional rewrites

```rust
fn is_integer(x: Var) -> impl Fn(&mut EGraph, egg::Id, &egg::Subst) -> bool {
    move |egraph: &mut EGraph<ChiIR, ChiAnalysis>, _id: Id, subst: &Subst| match &egraph[subst[x]].data.analysis_info {
        AnalysisInfo::DType(dt: &DataType) => match dt {
            DataType::Int(_) | DataType::UInt(_) => true,
            _ => false,
        },
        _ => false,
    }
}
rw!("times-2-shift"; "(smult ?x 2)" => "(bitshl ?x 1)" if is_integer("?x".parse().unwrap())),
```

Since the E-Class matched to **?v** has Checked_type: **f32**, this rewrite rule won't be fired.



Benefits behind

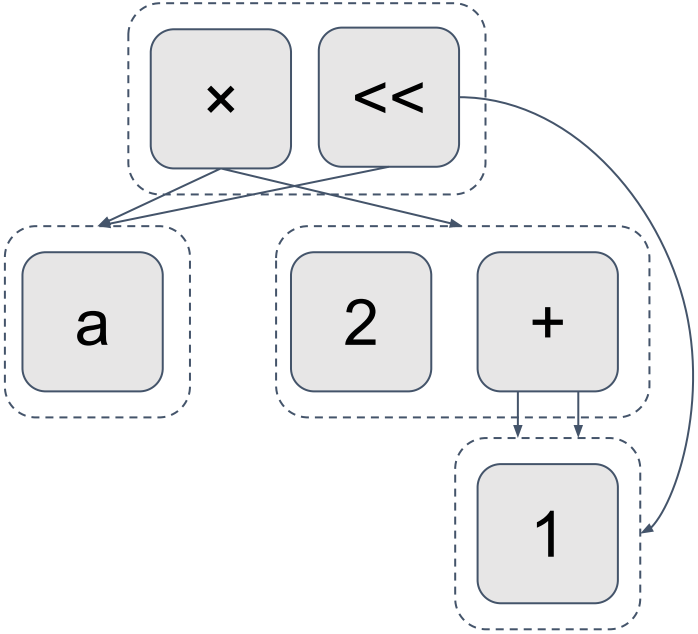
1. Observation: even there are rules that keep the EGraph from saturating[1], we are able to explore a large space of equivalences efficiently and automatically

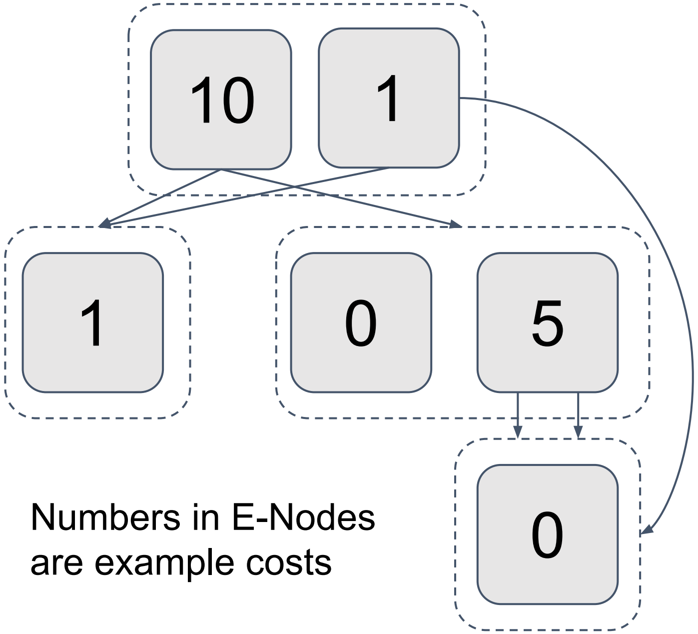
2. Verifying individual rule guarantees soundness of their compositions[2]
3. Lower the difficulty of contributing optimization rewrites
4. Enable facilitating new backend by adding tiling / offloading rewrites

1: This is the case for most applications because of expansive rules, e.g. $?\mathrm{x} \Rightarrow \mathtt{transpose}(\mathtt{transpose}(?\mathrm{x}))$
2: we are focusing on *functional* rewrites so far

Extraction in egg

Extraction: Given a root E-Class, pick the “best” term
(***minimizing*** the sum of costs of E-Nodes given by a cost model)

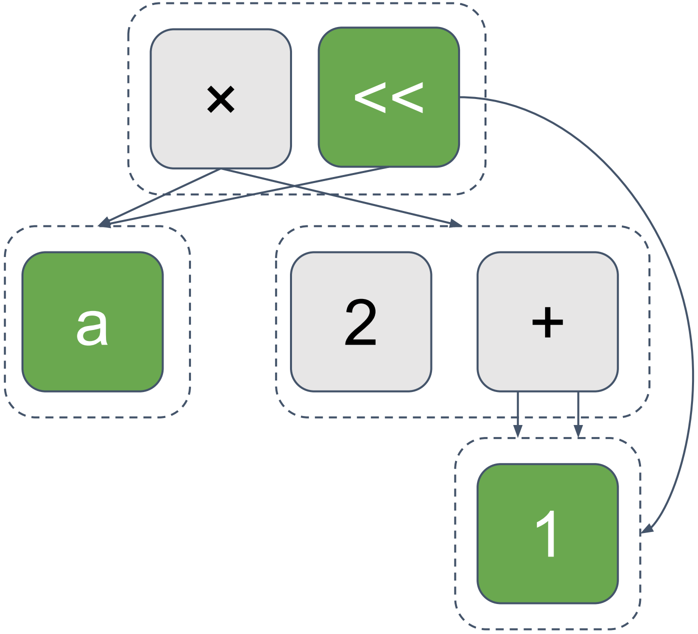


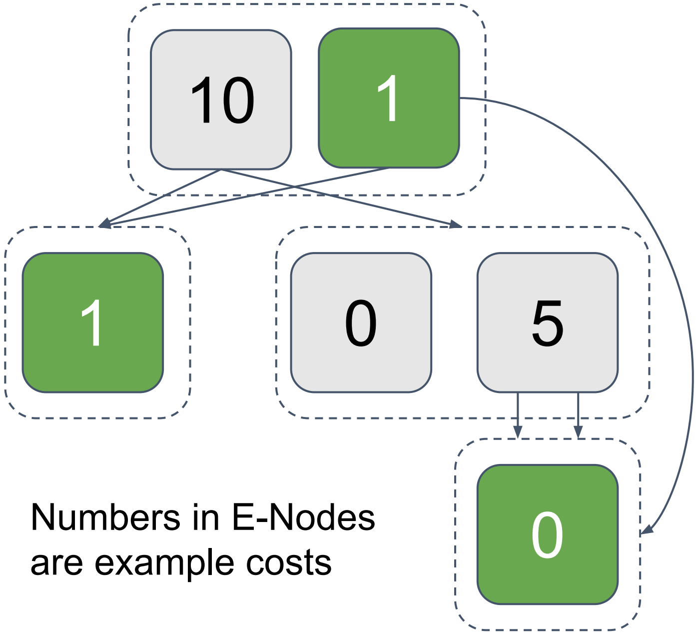


Numbers in E-Nodes are example costs

Extraction in egg

Extraction: Given a root E-Class, pick the “best” term
(***minimizing*** the sum of costs of E-Nodes given by a cost model)





Numbers in E-Nodes
are example costs

Extraction in egg

Implementations

- Greedy: Pick the minimum one at each level
  - 😁 Easy to implement
  - 😅 Don’t know about CSE (sharing)
- Integer Linear Programming (ILP)
  - 😁 Sound minimum
  - 😅 Timeout; does not work well with cycles

A CHI IR subset in egg

Thanks to egg's extensibility, we are able to encode a (functional) *subset* of CHI IR in egg

```
"sadd" = SAdd([Id; 2]),
"sminus" = SMinus([Id; 2]),
"smult" = SMult([Id; 2]),
"sdiv" = SDiv([Id; 2]),
"smod" = SMod([Id; 2]),
```

```
"≥" = Gte([Id; 2]),
"≤" = Lte([Id; 2]),
">" = Gt([Id; 2]),
"<" = Lt([Id; 2]),
"==" = Equals([Id; 2]),
```

```
"land" = LAnd([Id; 2]),
"lor" = LOr([Id; 2]),
"lnot" = LNot([Id; 1]),
"lxor" = LXor([Id; 2]),
```

```
// Vector init
// (vector <list> <data-type>)
"vector" = Vector(Vec<Id>),
// Matrix init
// (matrix <list of vectors> <data-type>)
"matrix" = Matrix(Vec<Id>),
"index" = Index([Id; 2]),
"matmul" = MatMul([Id; 2]),
// element-wise addition
// (ewadd mat/vec mat/vec)
"ewadd" = EWAdd([Id; 2]),
"ewmult" = EWMult([Id; 2]),
// Matrix transpose
"transpose" = Transpose([Id; 1]),
```

We mostly focus on matrices: major workload

Full language definition is available here: https://github.com/AD1024/egg-taichi/blob/main/src/language.rs

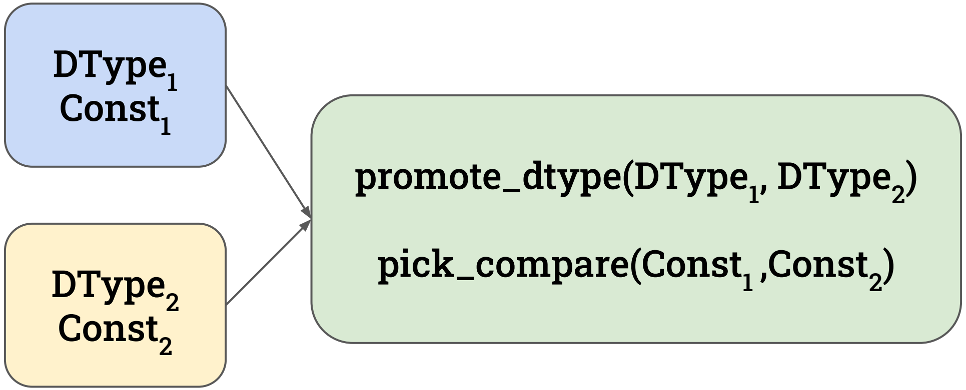
CHIAnalysis

Representation

DataType Analysis:
DType of the expression

Constant Info:
Option<ConstData>;
whether the expression yields a constant

Merging



promote_dtype follows taichi's typing rule
pick_compare chooses a **Some** value; if both are **Some** variant, then compare them

Rewrites examples

Scalar Rewrites

(sadd ?x ?y) ⇒ (sadd ?y ?x)

(smult (sadd ?x ?y) ?z) ⇒ (sadd (smult ?x ?z) (smult ?y ?z))

(pow 2 ?x) ⇒ (bitshl 1 ?x)   if *is_integer(?x)*

Matrix/Vector Rewrites

(transpose (transpose ?x)) ⇒ ?x

(transpose (add ?x ?y)) ⇒ (add (transpose ?x) (transpose ?y))

(matmul ?x (matmul ?y ?z)) ⇒ (matmul (matmul ?x ?y) ?z))

More Rewrites

Customized rewrites: Constant folding
(bear me with not using a macro for these :p)

```
rw!("const-fold-add"; "(sadd ?x ?y)" => { BinopConstFoldApplier { lhs: "?x".parse().unwrap(), rhs: "?y".parse().unwrap(), op: "+".to_string() }}),
rw!("const-fold-mult"; "(smult ?x ?y)" => { BinopConstFoldApplier { lhs: "?x".parse().unwrap(), rhs: "?y".parse().unwrap(), op: "*".to_string() }}),
rw!("const-fold-div"; "(sdiv ?x ?y)" => { BinopConstFoldApplier { lhs: "?x".parse().unwrap(), rhs: "?y".parse().unwrap(), op: "/".to_string() }}),
rw!("const-fold-sub"; "(sminus ?x ?y)" => { BinopConstFoldApplier { lhs: "?x".parse().unwrap(), rhs: "?y".parse().unwrap(), op: "-".to_string() }}),
```

More Rewrites

```rust
impl Applier<ChiIR, ChiAnalysis> for BinopConstFoldApplier {
    fn apply_one(
        &self,
        egraph: &mut egg::EGraph<ChiIR, ChiAnalysis>,
        eclass: egg::Id,
        subst: &egg::Subst,
        _: Option<&egg::PatternAst<ChiIR>>,
        _: egg::Symbol,
    ) -> Vec<egg::Id> {
        if let (Some(c1: ConstData), Some(c2: ConstData)) = (
            ChiAnalysis::get_constant(egraph, id: &subst[self.lhs]),
            ChiAnalysis::get_constant(egraph, id: &subst[self.rhs]),
        ) {
```

Enables us to check & use analysis data, and then fire a customized rewritten term.

E.g.: if we are folding +, then the resulted term is a constant equal to the sum of two constant data in the E-Class analysis.

Full implementation:
https://github.com/AD1024/egg-taichi/blob/dd5c370395662c55b8d77c3ab601a365219835ce/src/rewrites.rs#L34-L85

Cost Model

For proof-of-concept prototype, we implement a simple cost model

For scalar operations, we use an “estimated” CPU cycle count;
For matrix operations, we use the number of vector dots.

In the future, we will take vectorized instruction into consideration.
Probably use a more precise approach: profiling on the machine running the optimizer.

Implementation:
https://github.com/AD1024/egg-taichi/blob/main/src/extraction.rs

Preliminary Results

```python
@ti.kernel
def init_mesh():
    for i, j in ti.ndrange(N, N):
        k = (i * N + j) * 2
        a = i * (N + 1) + j
        b = a + 1
        c = a + N + 2
        d = a + N + 1
        f2v[k + 0] = [a, b, c]
        f2v[k + 1] = [c, d, a]
```

We set the constant N to 16

```
(cons
   (smult (sadd (smult i N) j) 2)
   (cons
     (sadd j (smult i (sadd N 1)))
     (cons
       (sadd 1 (sadd j (smult i (sadd N 1))))
       (sadd N (sadd 2 (sadd j (smult i (sadd N 1))))))))
```

Cost before optimization : 92

Preliminary Results

```python
@ti.kernel
def init_mesh():
    for i, j in ti.ndrange(N, N):
        k = (i * N + j) * 2
        a = i * (N + 1) + j
        b = a + 1
        c = a + N + 2
        d = a + N + 1
        f2v[k + 0] = [a, b, c]
        f2v[k + 1] = [c, d, a]
```

We set the constant N to 16

```
(cons
  (smult (sadd (smult i N) j) 2)
  (cons
    (sadd j (smult i (sadd N 1)))
    (cons
      (sadd 1 (sadd j (smult i (sadd N 1))))
      (sadd N (sadd 2 (sadd j (smult i (sadd N 1))))))))
```

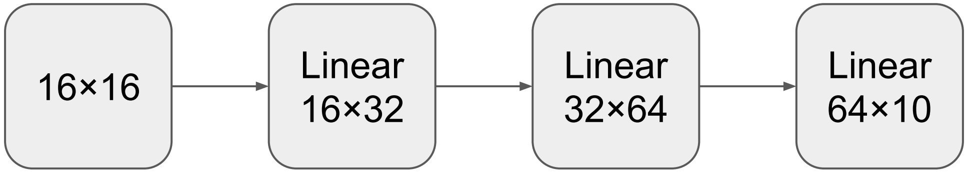
Cost before optimization : 92

```
(cons
  (sadd (bitshl i 5) (bitshl j 1))
  (cons
    (sadd (bitshl i 4) (sadd i j))
    (cons
      (sadd (bitshl i 4) (sadd i (sadd j 1)))
      (sadd i (sadd (bitshl i 4) (sadd j 18))))))
```

Cost after optimization: 53

Preliminary Results

A Simple matrix multiplications / element-wise additions (MLP)



Preliminary Results

A Simple matrix multiplications / element-wise additions (MLP)

```
(ewadd
	(matmul
		(ewadd
			(matmul
				(ewadd
					(matmul input W1)
					b1)
			W2)
			b2)
	W3)
	b3)
```

```
(ewadd
	(ewadd
		(matmul
			(ewadd
				(matmul input W1)
			b1)
			(matmul W2 W3))
		(matmul b2 W3))
	b3)
```

Cost before optimization: 51361

Cost after optimization: 44140

Discussion

1. egg only works well with data-flow based IR, but CHI IR has control flow operators
    a. Encode Loops in terms of mathematical functions (Tate et al.)
    b. Conversion from/to CFG
2. Global effects are hard to handle in egg’s representation
    a. Focus on pure functions / procedures first
    b. Proper effect handling transformations before converting into egg
3. Matrix operations representations in CHI IR

Q & A