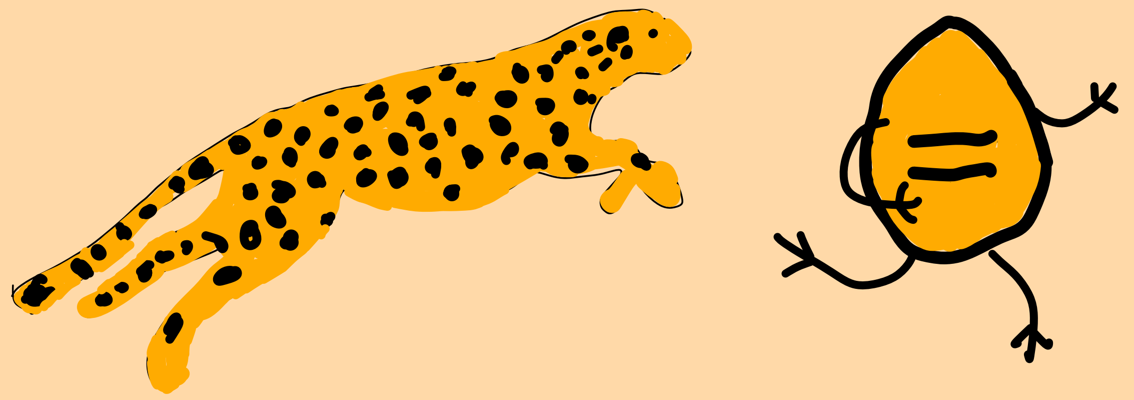
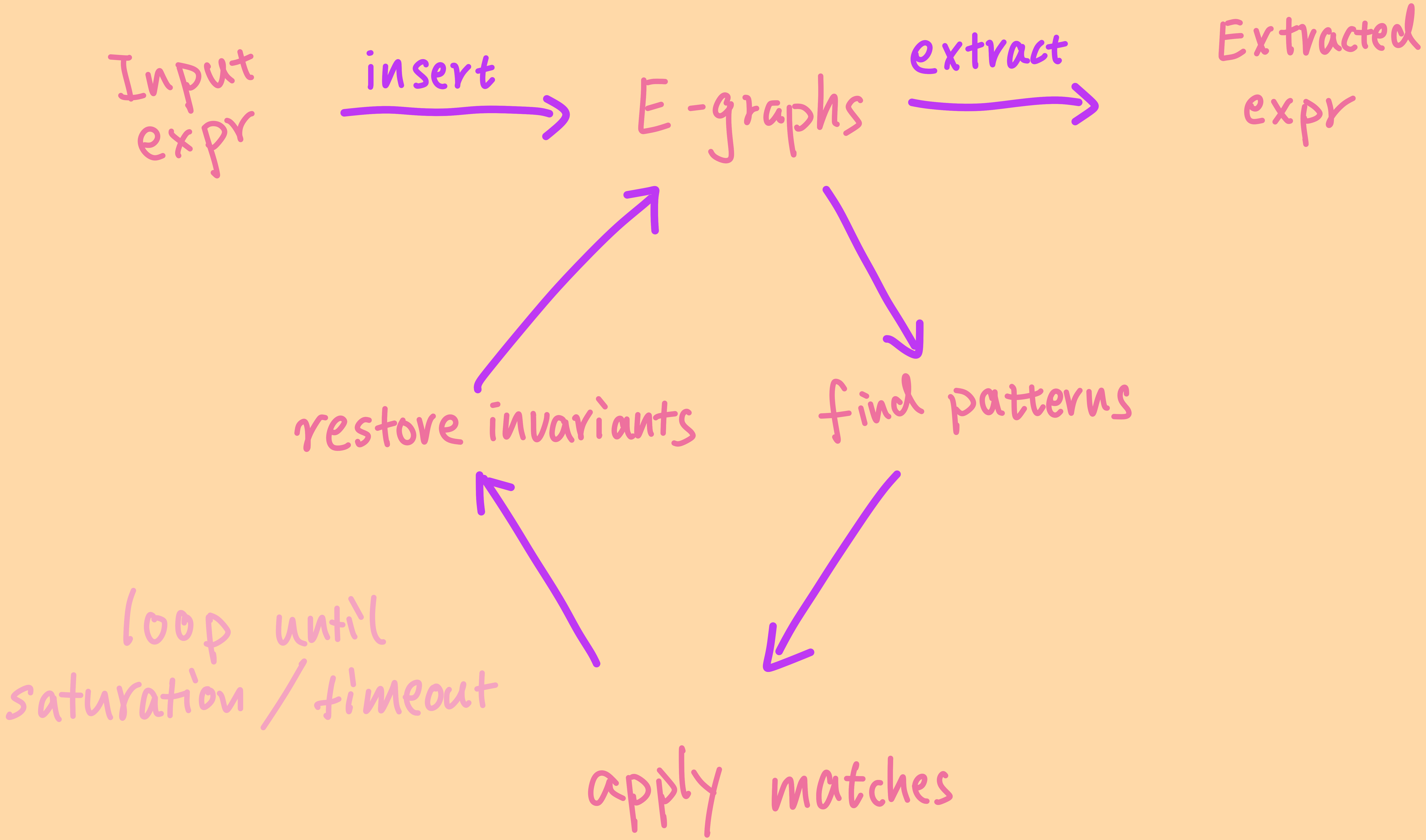
Chasing an Egg:

Towards a Relational E-graph

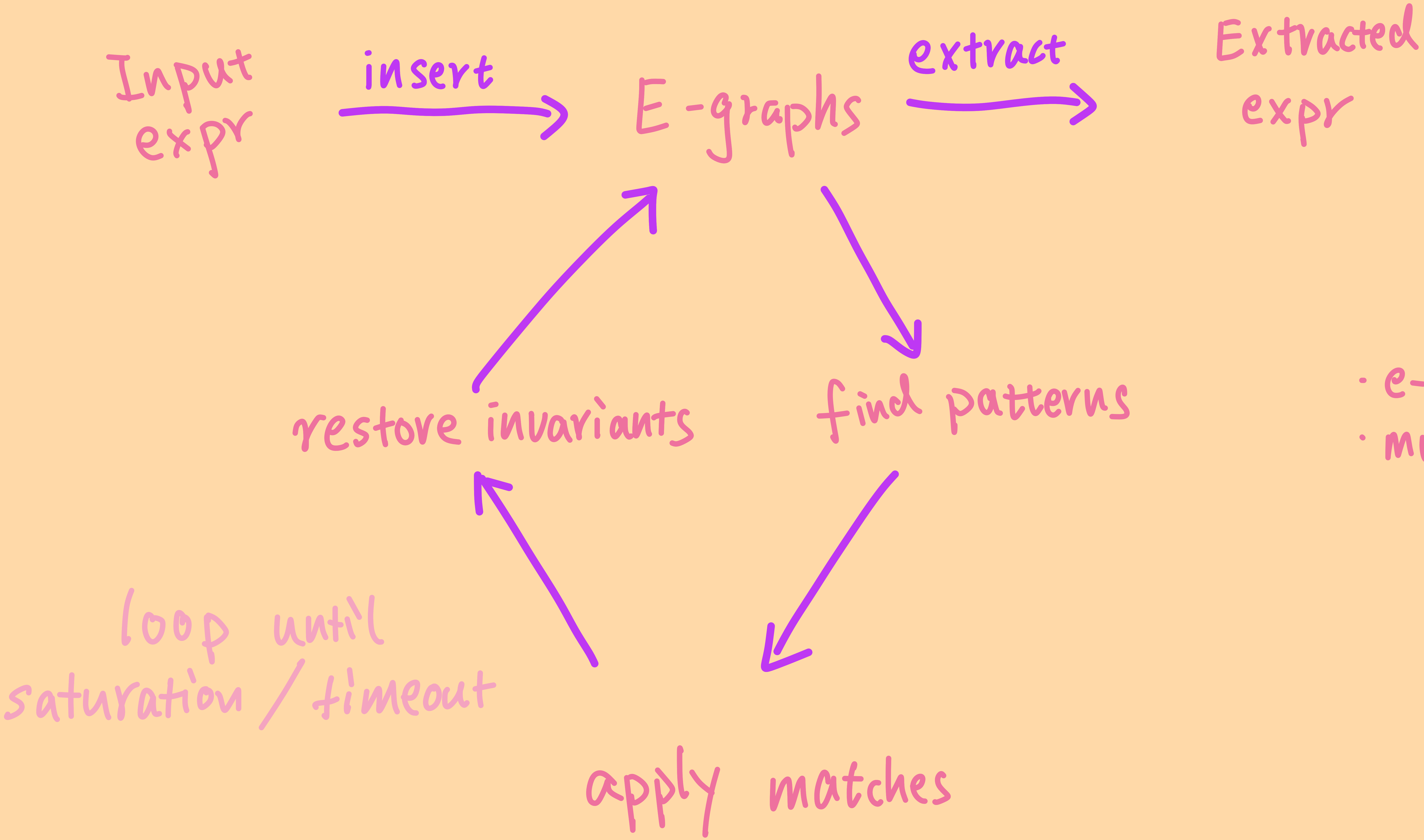
EGRAPHS 2022



Equality saturation

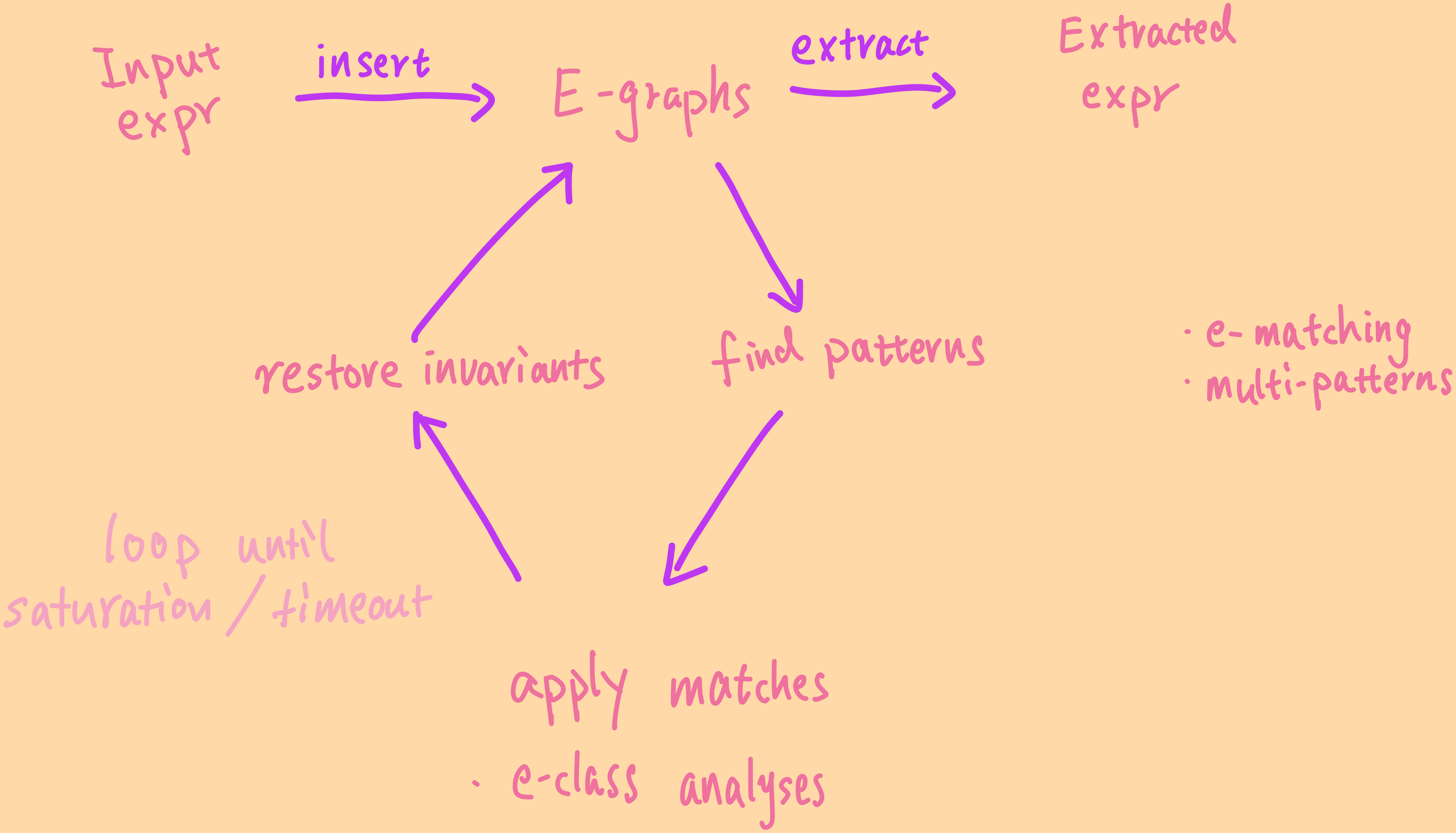


Equality saturation

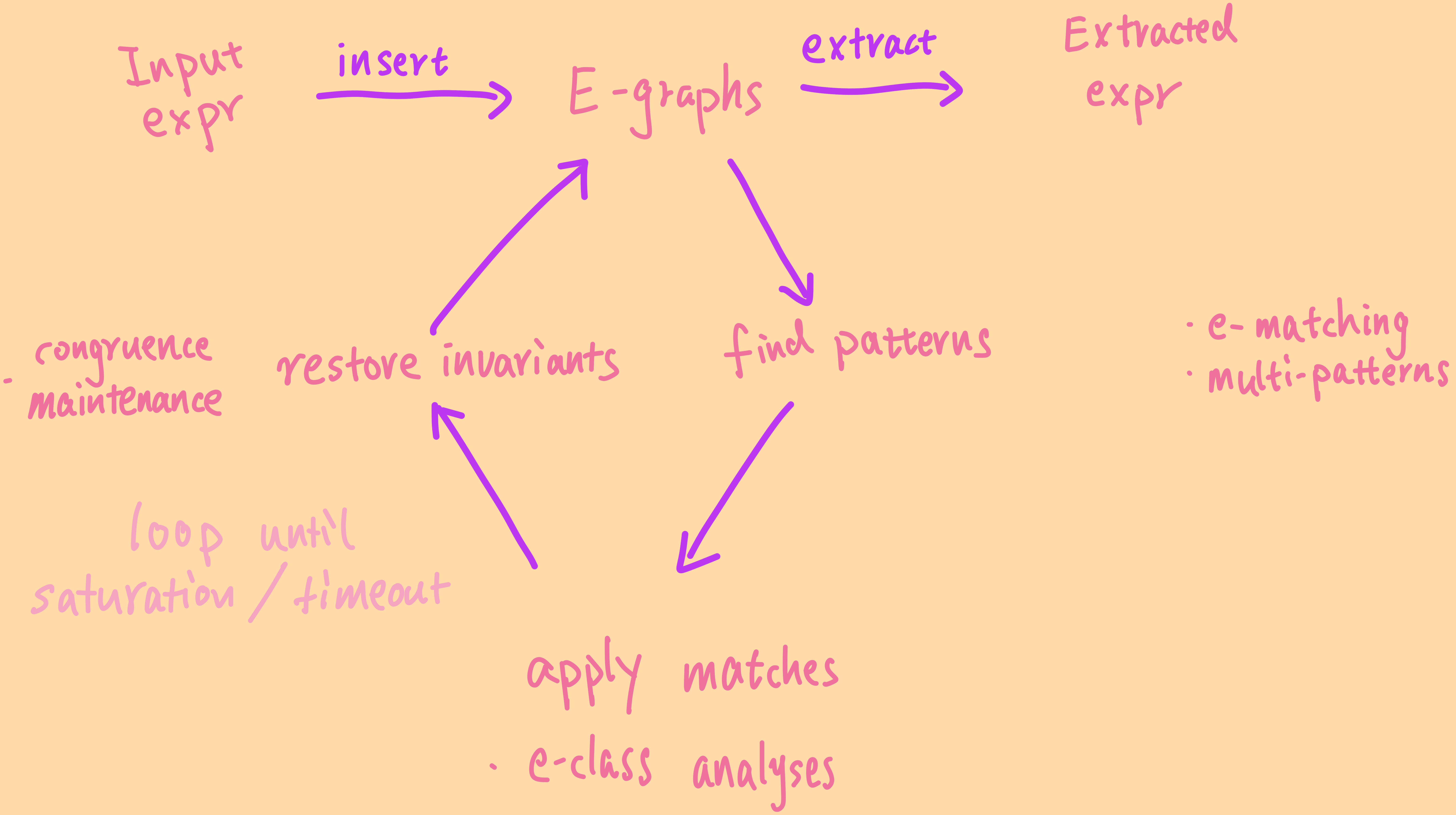


- e-matching
- multi-patterns

Equality saturation



Equality saturation



Limitations

Limitations

- Expressiveness

Limitations

- Expressiveness
  - multi-patterns are hard

Limitations

- Expressiveness
  - multi-patterns are hard
  - non-equational reasoning is hard

Limitations

- Expressiveness
  - multi-patterns are hard
  - non-equational reasoning is hard
- Performance

Limitations

- Expressiveness
  - multi-patterns are hard
  - non-equational reasoning is hard
- Performance
  - e-matching is slow

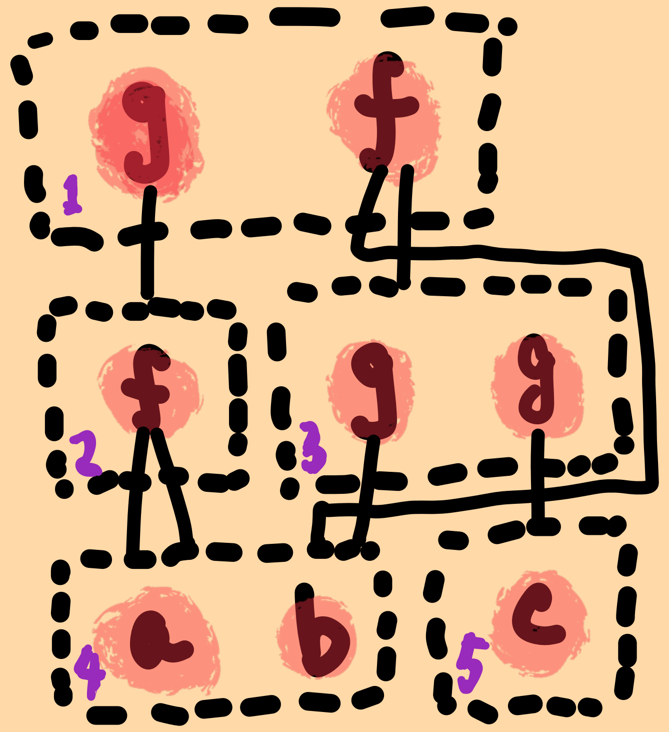
Limitations

- Expressiveness
  - multi-patterns are hard
  - non-equational reasoning is hard
- Performance
  - e-matching is slow
  - e-matching duplicates work

Limitations

- Expressiveness
  - multi-patterns are hard
  - non-equational reasoning is hard
- Performance
  - e-matching is slow
  - e-matching duplicates work
    - incremental e-matching is even harder

Relational E-matching (POPL 2022)



arg_1	arg_2	id
4	3	1
4	4	2

R_f

arg_1	id
4	3
5	3

R_g

id
4

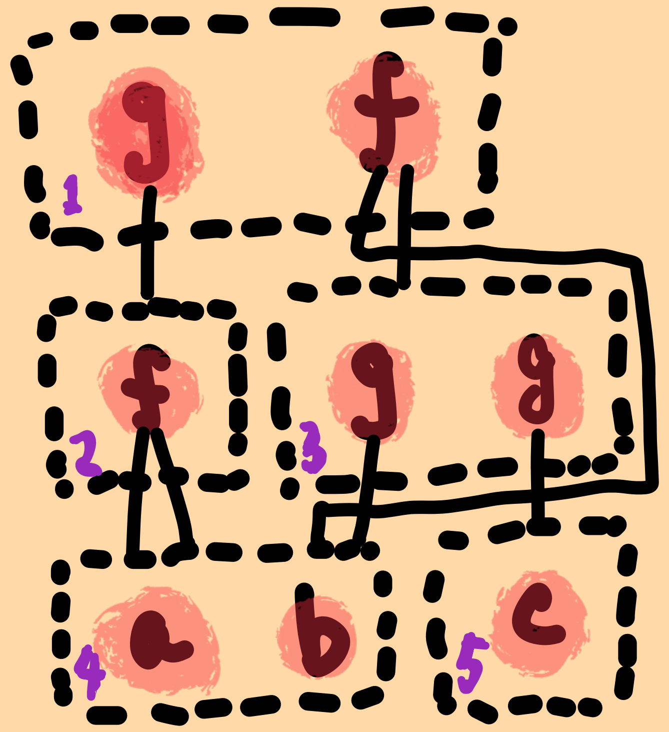
R_a, R_b

id
5

R_c

Relational E-matching (POPL 2022)

- e-graphs are relational databases
- e-matching are relational queries



arg_1	arg_2	id
4	3	1
4	4	2

R_f

arg_1	id
4	3
5	3

R_g

id
4

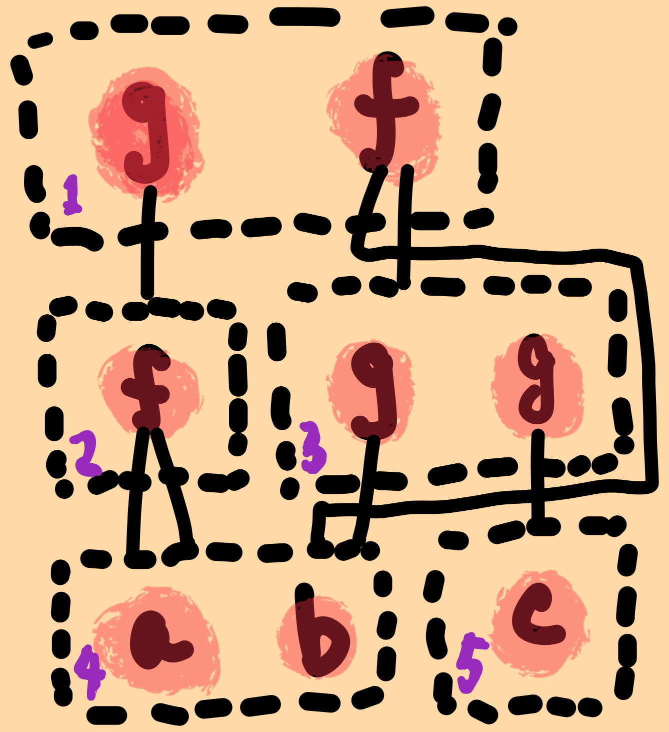
R_a, R_b

id
5

R_c

Relational E-matching (POPL 2022)

- e-graphs are relational databases
- e-matching are relational queries
- asymptotic speedup (8,000,000 x speedup)
- new complexity bound (+ optimal algorithm achieving it)



arg_1	arg_2	id
4	3	1
4	4	2

R_f

arg_1	id
4	3
5	3

R_g

id
4

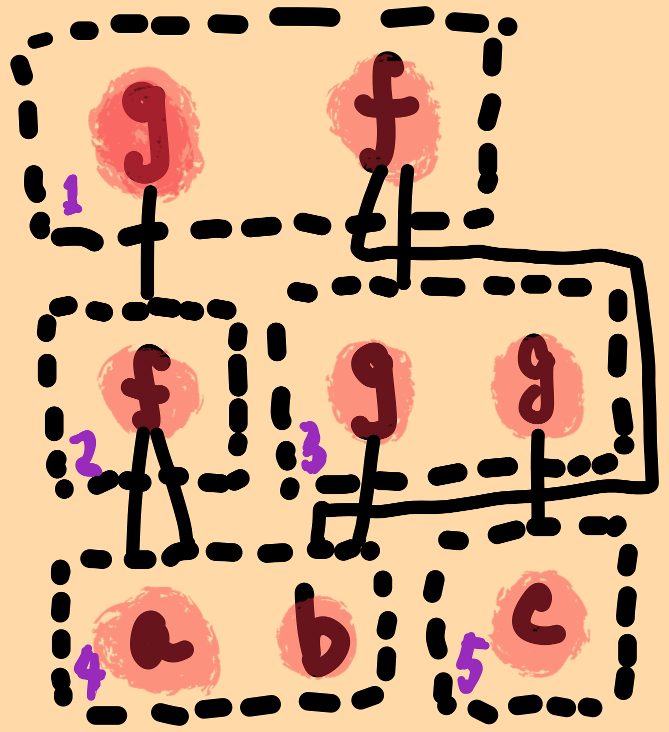
R_a, R_b

id
5

R_c

Relational E-matching (POPL 2022)

- e-graphs are relational databases
- e-matching are relational queries
- asymptotic speedup (8,000,000 x speedup)
- new complexity bound (+ optimal algorithm achieving it)
- easy support for multi-patterns



arg_1	arg_2	id
4	3	1
4	4	2

R_f

arg_1	id
4	3
5	3

R_g

id
4

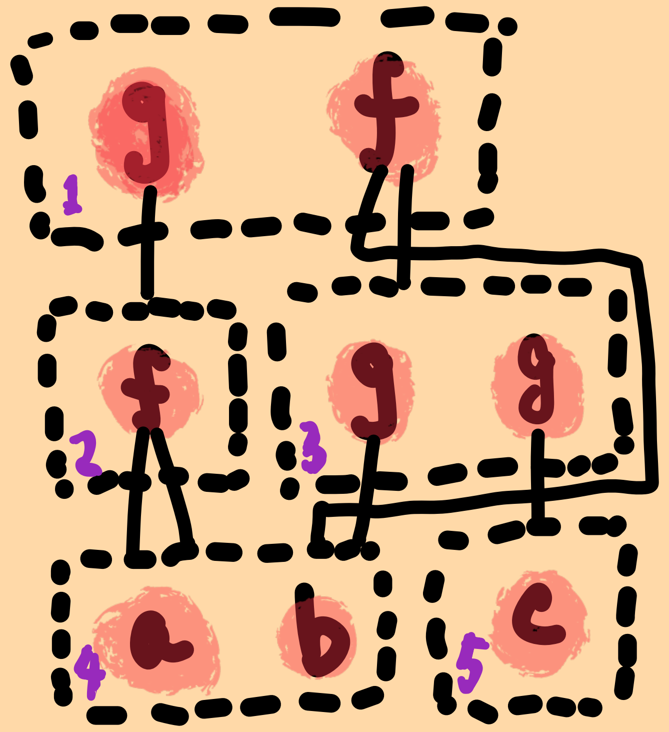
R_a, R_b

id
5

R_c

Relational E-matching (POPL 2022)

- e-graphs are relational databases
- e-matching are relational queries
- asymptotic speedup (8,000,000 x speedup)
- new complexity bound (+ optimal algorithm achieving it)
- easy support for multi-patterns
- not used in egg and other existing e-graph frameworks 🤔



arg_1	arg_2	id
4	3	1
4	4	2

R_f

arg_1	id
4	3
5	3

R_g

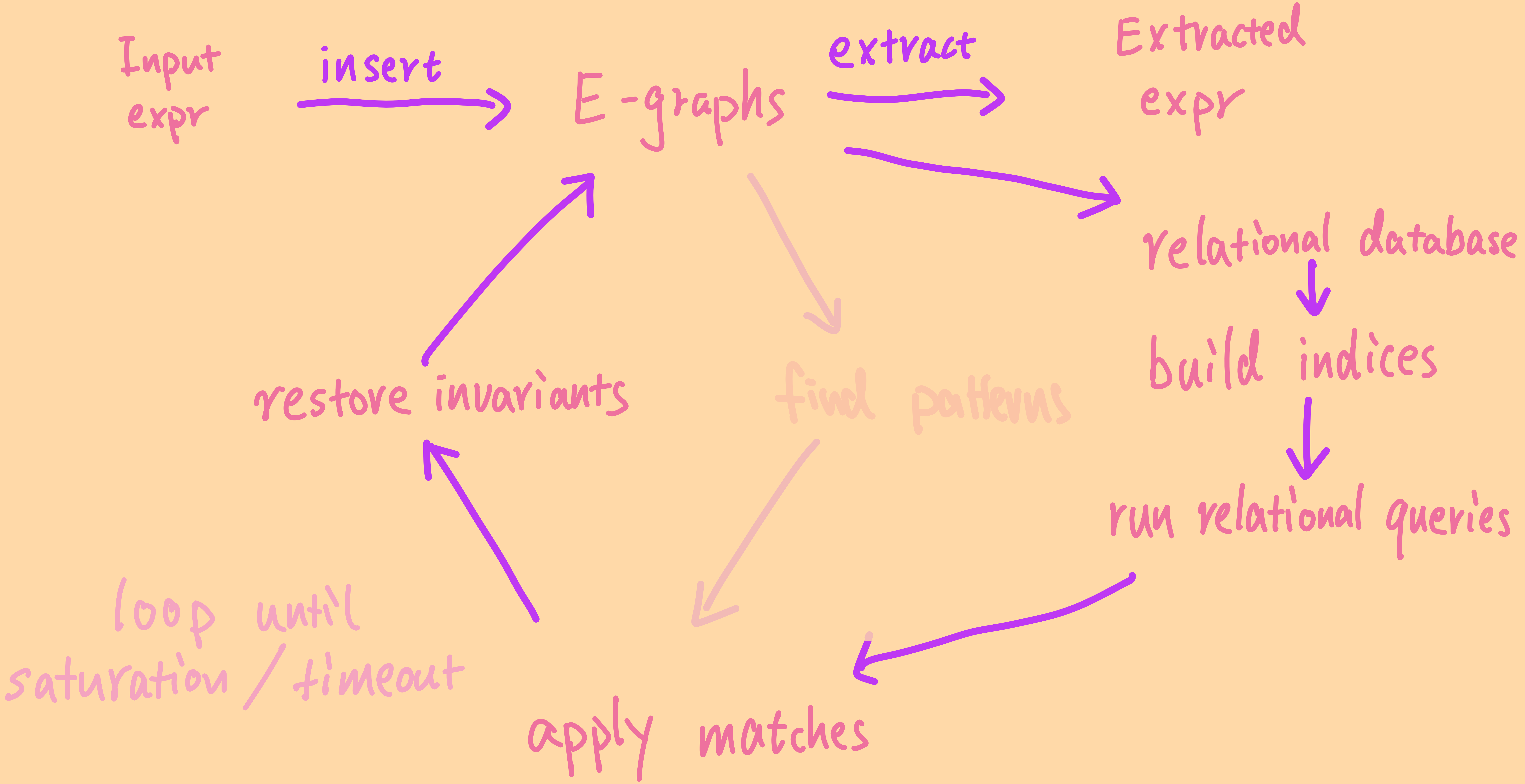
id
4

R_a, R_b

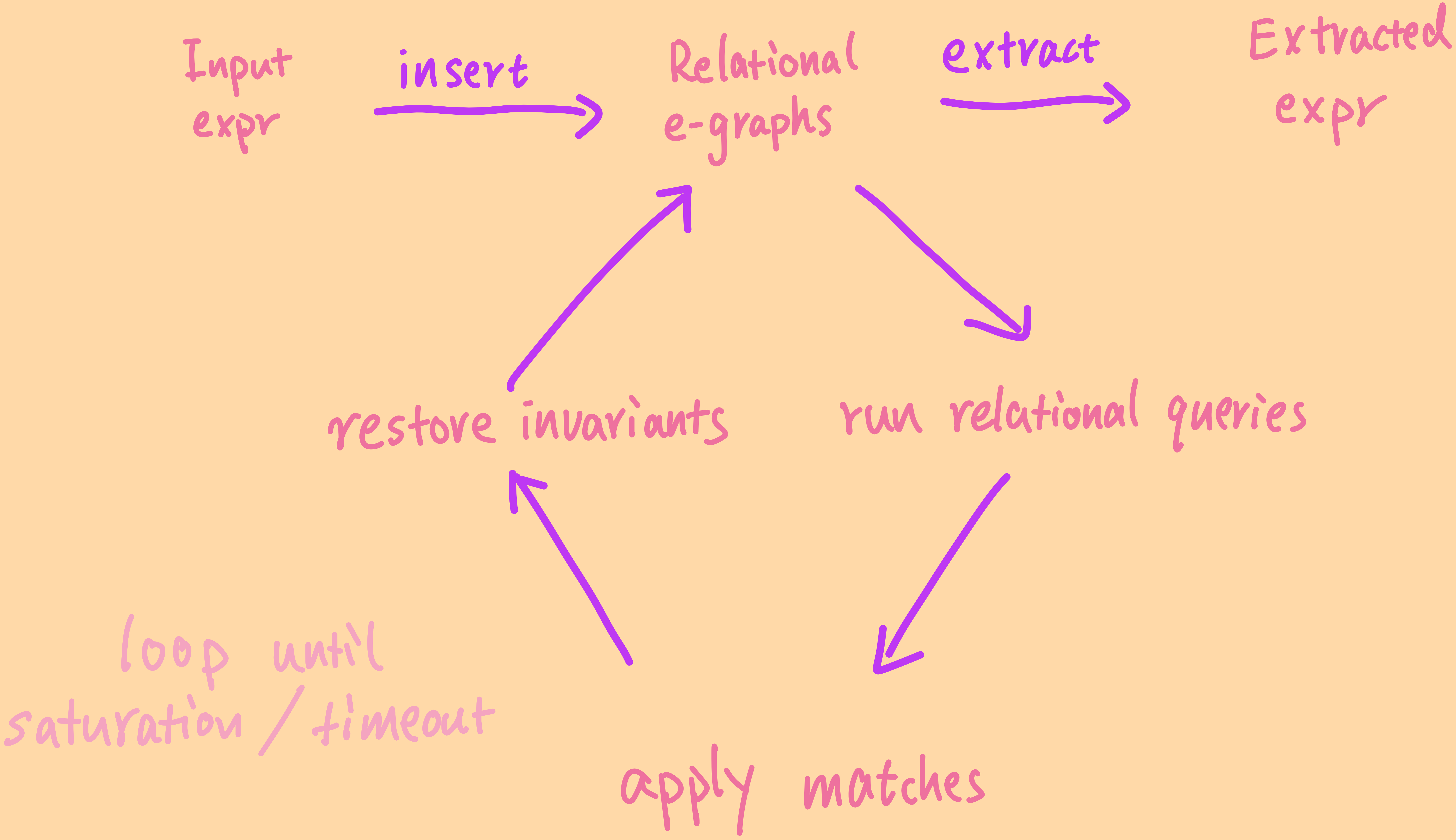
id
5

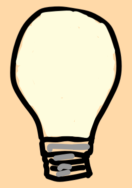
R_c

Relational e-matching

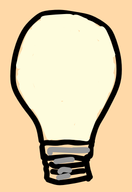


Relational ~~e-matching~~ e-graphs



egg 

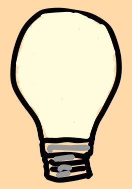
- e-graphs on top of SQLite
- e-graph operations are translated into SQL queries.
  - insertions
  - rewrites
  - rebuilding
- an order-of-magnitude slower than egg

egg: e-graphs on top of SQLite

e-graphs

examples

relational databases

egg  : e-graphs on top of SQLite

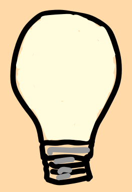
e-graphs

Congruence

add(a,b) → c

examples

relational databases

egg : e-graphs on top of SQLite

e-graphs

Congruence

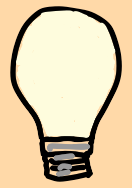
$add(a,b) \to c$

examples

$add(a,b,c_1), add(a,b,c_2) \Rightarrow$

$c_1 = c_2$

relational databases

egg: e-graphs on top of SQLite

e-graphs

Congruence

$add(a, b) \to c$

examples

$add(a, b, c_1), add(a, b, c_2) \Rightarrow$

$c_1 = c_2$

relational databases

Functional dependency

egg: e-graphs on top of SQLite

e-graphs

Congruence

$add(a,b) \rightarrow c$

Rewrite rules

$(a+b)+c \Rightarrow a+(b+c)$

examples

$add(a,b,c_1), add(a,b,c_2) \Rightarrow$

$c_1 = c_2$

relational databases

Functional dependency

egg: e-graphs on top of SQLite

e-graphs

Congruence

add(a, b) → c

Rewrite rules

$(a+b)+c \Rightarrow a+(b+c)$

examples

$add(a, b, c_1), add(a, b, c_2) \Rightarrow$

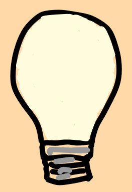
$c_1 = c_2$

$add(a, b, ab), add(ab, c, r) \Rightarrow$

$\exists bc, add(b, c, bc), add(a, bc, r)$

relational databases

Functional dependency

egg : e-graphs on top of SQLite

e-graphs	examples	relational databases
Congruence add(a, b) → c	add(a, b, c_1), add(a, b, c_2) ⇒ $c_1 = c_2$	Functional dependency
Rewrite rules $(a+b)+c \Rightarrow a+(b+c)$	add(a, b, ab), add(ab, c, r) ⇒ ∃ bc, add(b, c, bc), add(a, bc, r)	Tuple-generating dependency (generalizes Datalog)

(Relational) E-graphs are all about

data dependencies;

(Relational) E-graphs are all about data dependencies;

Equality saturation is a form of the chase.

The chase

The chase

- Family of algorithms for reasoning about data dependencies (e.g. functional dependencies, tuple-generating dependencies).

The chase

- Family of algorithms for reasoning about data dependencies (e.g. functional dependencies, tuple-generating dependencies).

```
While not fix-point:
  for each dependency τ:
    for each σ matching LHS of τ:
      apply RHS with σ if not already
```

The chase

- Family of algorithms for reasoning about data dependencies (e.g. functional dependencies, tuple-generating dependencies).
- Standard optimizations like semi-naive evaluations apply.

```
While not fix-point:
  for each dependency τ:
    for each σ matching LHS of τ:
      apply RHS with σ if not already
```

The chase

- Family of algorithms for reasoning about data dependencies (e.g. functional dependencies, tuple-generating dependencies).

- Standard optimizations like semi-naive evaluations apply.

- Equality saturation is a restricted kind of the chase, where

```
While not fix-point:
  for each dependency τ:
    for each σ matching LHS of τ:
      apply RHS with σ if not already
```

The chase

- Family of algorithms for reasoning about data dependencies (e.g. functional dependencies, tuple-generating dependencies).
- Standard optimizations like semi-naive evaluations apply.
- Equality saturation is a restricted kind of the chase, where
  - Evaluation is efficient;
  - Solutions have nice properties.

```
While not fix-point:
  for each dependency τ:
    for each σ matching LHS of τ:
      apply RHS with σ if not already
```

motivation for egg# (no full implementation yet)

motivation for egg# (no full implementation yet)

- To make relational e-graphs efficient & pratical, two problems need to be addressed
    1) Incremental e-matching
    2) E-class analysis

motivation for egg# (no full implementation yet)

- To make relational e-graphs efficient & pratical, two problems need to be addressed
  1) Incremental e-matching
  2) E-class analysis
- 1) is easy from the Datalog perspective: just semi-naive evaluation.

motivation for egg# (no full implementation yet)

- To make relational e-graphs efficient & pratical, two problems need to be addressed
  1) Incremental e-matching
  2) E-class analysis
- 1) is easy from the Datalog perspective: just semi-naive evaluation.
- 2) is sligtly harder. Thankfully people have figured it out.

motivation for egg# (no full implementation yet)

- To make relational e-graphs efficient & pratical, two problems need to be addressed
  1) Incremental e-matching
  2) E-class analysis
- 1) is easy from the Datalog perspective: just semi-naive evaluation.
- 2) is sligtly harder. Thankfully people have figured it out.

Lattice semantics of Datalog:
a tuple is a function from
D^n to L, where L is a lattice.

From Datalog to FLIX: A Declarative Language for Fixed Points on Lattices

Magnus Madsen
University of Waterloo, Canada
mmadsen@uwaterloo.ca

Ming-Ho Yee
University of Waterloo, Canada
ming-ho.yee@uwaterloo.ca

Ondřej Lhoták
University of Waterloo, Canada
olhotak@uwaterloo.ca

Abstract

We present FLIX, a declarative programming language for specifying and solving least fixed point problems, particularly static program analyses. FLIX is inspired by Datalog and extends it with lattices and monotone functions. Using FLIX, implementors of static analyses can express a broader range of analyses than is currently possible in pure Datalog, while retaining its familiar rule-based syntax.

We define a model-theoretic semantics of FLIX as a natural extension of the Datalog semantics. This semantics captures the declarative meaning of FLIX programs without imposing any specific evaluation strategy. An efficient strategy is

change its overall state at each computation step. A static analysis computes an abstract state $\hat{x}$ that over-approximates all possible concrete states that a program can reach. Every sound approximation must satisfy $\hat{F}(\hat{x}) \sqsubseteq \hat{x}$, where $\hat{F}$ is an abstraction of the concrete transformation function F, since if a state in $\hat{x}$ can be reached by a computation, then so can a state in $\hat{F}(\hat{x})$. The least $\hat{x}$ satisfying this property can be computed by starting from the least element $\bot$ and iteratively applying $\hat{F}$ until the fixed point is reached [15, 35].

Static analyzers, which involve fixed-point computations, are complex pieces of software often implemented in general-purpose languages such as C++ or Java. The many mutual de-

motivation for egg♯ (no full implementation yet)

- To make relational e-graphs efficient & pratical, two problems need to be addressed
  1) Incremental e-matching
  2) E-class
- 1) is easy
- 2) is sligth

egg	egg♯
Equational rewrites	Tuple-generating dependencies
Congruence rules	Functional dependencies (FD)
E-classes	User-defined sorts
E-class merges	FD repair through unification
E-class analyses	User-defined lattices
E-class analysis maintenance	FD repair through lattice joins

Lattice semantics of Datalog:
a tuple is a function from D^n to L, where L is a lattice.

From Datalog to FLIX: A Declarative Language for Fixed Points on Lattices

Magnus Madsen, University of Waterloo, Canada, mmadsen@uwaterloo.ca
Ming-Ho Yee, University of Waterloo, Canada, ming-ho.yee@uwaterloo.ca
Ondřej Lhoták, University of Waterloo, Canada, olhotak@uwaterloo.ca

Abstract

We present FLIX, a declarative programming language for specifying and solving least fixed point problems, particularly static program analyses. FLIX is inspired by Datalog and extends it with lattices and monotone functions. Using FLIX, implementors of static analyses can express a broader range of analyses than is currently possible in pure Datalog, while retaining its familiar rule-based syntax.

We define a model-theoretic semantics of FLIX as a natural extension of the Datalog semantics. This semantics captures the declarative meaning of FLIX programs without imposing any specific evaluation strategy. An efficient strategy is

change its overall state at each computation step. A static analysis computes an abstract state $\hat{x}$ that over-approximates all possible concrete states that a program can reach. Every sound approximation must satisfy $\hat{F}(\hat{x}) \sqsubseteq \hat{x}$, where $\hat{F}$ is an abstraction of the concrete transformation function F, since if a state in $\hat{x}$ can be reached by a computation, then so can a state in $\hat{F}(\hat{x})$. The least $\hat{x}$ satisfying this property can be computed by starting from the least element $\bot$ and iteratively applying $\hat{F}$ until the fixed point is reached [15, 35].

Static analyzers, which involve fixed-point computations, are complex pieces of software often implemented in general-purpose languages such as C++ or Java. The many mutual de-

Relational e-graphs

- E-graphs
  - congruence
  - rewrites
  - e-class analyses
- Expressiveness
  - multi-patterns are hard
  - non-equational reasoning is hard
- Performance
  - e-matching is slow
  - e-matching duplicates work

Relational e-graphs

- E-graphs
  - congruence   functional dependencies
  - rewrites
  - e-class analyses
- Expressiveness
  - multi-patterns are hard
  - non-equational reasoning is hard
- Performance
  - e-matching is slow
  - e-matching duplicates work

Relational e-graphs

- E-graphs
  - congruence — functional dependencies
  - rewrites — tuple-generating dependencies
  - e-class analyses
- Expressiveness
  - multi-patterns are hard
  - non-equational reasoning is hard
- Performance
  - e-matching is slow
  - e-matching duplicates work

Relational e-graphs

- E-graphs
  - congruence — functional dependencies
  - rewrites — tuple-generating dependencies
  - e-class analyses — lattice semantics of datalog
- Expressiveness
  - multi-patterns are hard
  - non-equational reasoning is hard
- Performance
  - e-matching is slow
  - e-matching duplicates work

Relational e-graphs

- E-graphs
  - congruence — functional dependencies
  - rewrites — tuple-generating dependencies
  - e-class analyses — lattice semantics of datalog
- Expressiveness
  - multi-patterns are hard — datalog
  - non-equational reasoning is hard
- Performance
  - e-matching is slow
  - e-matching duplicates work

Relational e-graphs

- E-graphs
  - congruence — functional dependencies
  - rewrites — tuple-generating dependencies
  - e-class analyses — lattice semantics of datalog
- Expressiveness
  - multi-patterns are hard — datalog
  - non-equational reasoning is hard — datalog
- Performance
  - e-matching is slow
  - e-matching duplicates work

Relational e-graphs

- E-graphs
  - congruence — functional dependencies
  - rewrites — tuple-generating dependencies
  - e-class analyses — lattice semantics of datalog
- Expressiveness
  - multi-patterns are hard — datalog
  - non-equational reasoning is hard — datalog
- Performance
  - e-matching is slow — relational e-matching
  - e-matching duplicates work

Relational e-graphs

- E-graphs
  - congruence — functional dependencies
  - rewrites — tuple-generating dependencies
  - e-class analyses — lattice semantics of datalog
- Expressiveness
  - multi-patterns are hard — datalog
  - non-equational reasoning is hard — datalog
- Performance
  - e-matching is slow — relational e-matching
  - e-matching duplicates work — semi-naive evaluations

Expressiveness

Beyond congruence

```
cons(Expr, Expr) -> Expr
sonc(Expr) -> (Expr, Expr)
```

```
cons(Expr, List) -> List
sonc(List) -> (Expr, List)
```

Non-equational reasoning

```
geq[x, 0] := true[] if num(n, x), n > 0.
// other arithmetic rules

geq[x, 0] :- abs[x]
x := abs[x] if geq[x, 0] = true[]
```

Composable analyses

```
lo[y] := lo[x] if abs[x] = y, lo[x] >= 0.
hi[y] := hi[x] if abs[x] = y, lo[x] >= 0.
lo[xy] := lox - hiy if lo(x, lox), hi(y, hiy), sub(x, y, xy).
hi[xy] := hix - loy if hi(x, hix), lo(y, loy), sub(x, y, xy).
```