A tutorial on the imaginary Gogi language

Yihong Zhang

Contents

Welcome to the tutorial on Gogi (short for eg**glogi**sh), a made-up language that attempts to generalize both Datalog and egg. This blog stems from a trick I learned from Nicholas Matsakis at PLMW 2022: To write a tutorial for a non-existing language. By doing this, I can get a sense of what I want from this new language as well as early feedbacks from others.

Why Gogi? The motivation behind Gogi is to find a good model for relational e-graphs that can take full advantage of (1) performance of relational e-matching and (2) expressiveness of Datalog, while (3) being compatiable with egg as well as (4) efficient. This is the first approach I described at the beginning of the previous post. I'm actually more excited about this approach, because I believe this is *the right way* in long term.

Gogi is Datalog, so it supports various reasoning expressible in Datalog. A rule has the form `head1, ..., headn :- body1, ..., bodyn`. For example, below is a valid Gogi program:

```
rel link(string, string) from "./link.csv".
rel tc(string, string).

tc(a, b) :- link(a, b).
tc(a, b) :- link(a, c), tc(c, b).
```

TODO: However, Datalog by itself is not that interesting. So for the first part of the post, I will instead focus on the extensions that make Gogi interesting. Next, I'll give some examples and show why Gogi generalizes egg I will also try to develop the operational and model semantics of Gogi.

Introduction to Gogi

Ext 1: User-defined sorts and lattices

In Gogi, every value is either a (semi)lattice value or a sort value. Lattices in Gogi are algebraic structures with a binary join operator ($\vee$) that is associative, commutative, and idempotent and a default top $\top$ where $\top \vee e = \top$ for all e. For example, standard types like `string`, `i64`, and `u64` in Gogi are in fact trivial lattices with $s_1 \vee s_2 = \top$ for all $s_1 \neq s_2$. In Gogi, $\top$ means unresolvable errors. Users can define their own lattices by providing a definition for lattice join.

Similarly, users can define sorts. Unlike lattices, sorts are uninterpreted. As a result, sort values can only be created implicitly via functional dependency. We will go back to this point later.

Ext 2: Relations and Functional Dependencies

Declaring a relation with functional dependency

Relations can be declared using the `rel` keyword. Moreover, it is possible to specify a functional dependency between columns in Gogi. For example,

```
sort expr.
rel num(i64) -> expr.
```

declares a sort called `expr` and a `num` relation with two columns `(i64, expr)`. In the `num` relation, each `i64` uniquely determines the remaining column (i.e., `num(x, e1)` and `num(x, e2)` implies `e1 = e2`). The `num` relation can be read as a function from `i64` to values in `expr`. Similar declarations are ubiquitous in Gogi to represent sort constructors.

As another example,

```
rel add(expr, expr) -> expr.
```

declares a relation with three columns, and the first two columns together uniquely determines the third column. This represents a constructor with two `expr` arguments.

Users can introduce new sort values with functional dependencies. Example:

```
num(1, c). % equivalently, num(1, _).
num(2, d).
add(c, d, e) :- num(1, c), num(2, d).
```

This program is interesting and its semantics deviates from the one in standard Datalog. In standard Datalog, this program will not compile because variable `c` in the first rule, `d` in the second rule, and `e` in the third rule are not bound. However, this is a valid program in Gogi. Thanks to functional dependency, variables in the head do not necessarily have to be bound in the bodies. Variables can be unbound as long as they can be inferred from the functional dependency. The above Gogi program is roughly equivalent to the following Datalog program:

```
num(1, c) :- !num(1, _), c = new_expr().
num(2, d) :- !num(2, _), d = new_expr().
add(c, d, e) :- num(1, c), num(2, d), !add(c, d, _), e = new_expr()
```

Negated atoms like `!num(1, _)` is necessary here because otherwise it will inserts more than one atoms matching `num(1, _)`, which violates the functional dependency associated to the relation.

The above example Gogi program can also be written into one single rule with multiple heads:

```
add(c, d, e), num(1, c), num(2, d).
% roughly equivalent to
% add(c, d, e), num(1, c), num(2, d) :- !num(1, _), !num(2, _),
%                                       c = new_expr(),
%                                       d = new_expr(),
%                                       !add(c, d, _),
%                                       e = new_expr().
```

The bracket syntax

Gogi also supports the bracket syntax, so the last program can be further simplified to:

```
add[num[1], num[2]].
```

The bracket syntax will implicitly fill the omitted column(s) with newly generated variable(s). If the atom is nested within another term, the nested atom will be lifted to the top-level, and the generated variable(s) will take the original position of the atom. Another silly example of the bracket syntax:

```
ans(x) :- xor[xor[x]].
% expands to
% ans(x) :- xor[y, z], xor(x, y, z)
% which expands to
% ans(x) :- xor(y, z, _), xor(x, y, z)
% this rule can be thought as
%   for any expr x, y where `y xor (x xor y)`
%   is present in the database, collect x as the result.
```

Finally, in equational reasoning a la egg, it is common to write rules like "for every `(a + b) + c`, populate `a + (b + c)` on the right and make them equivalent". This rule will look like the following:

```
add(a, add[b, c], id) :- add(add[a, b], c, id).
```

Gogi further has a syntactic sugar for these equational rules: `head := body if body1 ... bodyn` where both `head` and `body` should use the bracket syntax and omit the same number of columns. The if clause can be omitted. Gogi will expand this syntactic sugar by unfolding the top-level bracket in `head` and `body` with the same variable(s):

```
add[b, a] := add[a, b].
% unfolds to add(b, a, id) :- add(a, b, id).
add[a, add[b, c]] := add[add[a, b], c].
% unfolds to add(a, add[b, c], id) :- add(add[a, b], c, id).
num[1] := div[a, a] if num(x, a), x != 0.
% unfolds to num(1, id) :- div(a, a, id), num(x, a), x != 0.
```

Note the equational rules may introduce functional dependency violation; for instance, last rule may cause multiple tuples to match `num(1, _)`, yet the first column should uniquely determines the tuple. We will discuss more about how we resolve this kind of violations in the section on Functional Dependency Repair. The essential idea is that, if two sort values are present with the same primary key, then the two sort values must be equivalent, whereas if two lattice values are present with the same primary key, the new, unique lattice value should generalize the two values, i.e., it will be the least-upper bound of those lattice values.

Relations with lattices

The example relations we see so far mostly center around sort values. However, it is also possible and indeed very useful to define relations with lattices:

```
rel hi(expr) -> lmax(-2147483648).
rel lo(expr) -> lmin(2147482647).
```

To define a lattice column, a default value need to be provided in the relation definition. The default value is not a lattice bottom: the bottom means do not exist. Meanwhile, the lattice top means there are conflicts. It is also possible for default value to refer to the determinant columns:

```
rel add1(i: i64) -> i64(i + 1).
```

The column initialization syntax should be reminiscent of C++'s member initializer lists.

In the above example, `lo` and `hi` together define a range analysis for the `expr` sort. This in facts generalizes the e-class analyses in egg. Here are some rules for `hi` and `lo` (stolen from Zach):

```
hi(x, n.into()) :- num(n, x).
lo(x, n.into()) :- num(n, x).
lo(nx, n.negated()) :- hi(x, n), neg(x, nx).
hi(nx, n.negated()) :- lo(x, n), neg(x, nx).
lo(absx, 0) :- abs(x, absx).
lo[absx] := lo[x] if abs(x, absx), lo[x] >= 0.
hi[absx] := hi[x] if abs(x, absx), lo[x] >= 0.
lo(xy, lox + loy) :- lo(x, lox), hi(y, loy), add(x, y, xy).
% can be further simplified to
%   lo[xy] := lo[x] + lo[y] if add(x, y, xy)
```

Note here instead of `lo(neg[x], n.negated()) :- hi(x, n).`, we put the `neg` atom to the right-hand side and write `lo(nx, n.negated()) :- hi(x, n), neg(x, nx).` There are some nuanced differences between the two rules. This rule, besides doing what the second rule does, always populates a `neg` tuple for each `hi` tuple even when it does not exist, so the first rule can be viewed as an "annotation-only" version of the first rule, which is usually what we want.

The last example shows e-class analyses in Gogi is composable (i.e., each analysis can freely refer to each other). This is one of the reason why we believe Gogi generalizes e-class analyses. Moreover, they can also interact with other non-lattice relations in a meaningful way: [1]

```
rel geq(expr, expr).

% ... some arithmetic rules ...

% need to convert to int because they are from different lattices
geq(a, b) :- lo[a].to_int() < hi[b].to_int().
% TODO: geq is quadratic in size.
% Gogi should support inlined relations
% to avoid materialize relations like geq
```

[1]The last rule in this example has a single variable on the left-hand side, but the above mentioned syntactic expansion for `:=` does not apply to this case. The rule is indeed equivalent to `abs(x, x) :- abs[x] if geq(x, num[0])`.

```
% ... other user-defined knowledge about geq

% x and abs[x] are equivalent when x > 0
x := abs[x] if geq(x, num[0])
```

Diverging a little bit, it is even possible to write the above rules without using analyses / relations with lattices:

```
sort bool.
rel true() -> bool.
rel false() -> bool.
rel geq(expr, expr) -> bool.

% for each abs[x] exists, populate geq[x, 0],
% in the hope that later
% it will be "in the same e-class" as true[].
geq[x, 0] :- abs[x]

geq[numx, 0] := true[] if num(x, numx), x > 0.
geq[xx, 0] := true[] if mul(x, x, xx).
% if x > 0 and y > 0 are both equivalent to true,
% then x + y > 0 is also equivalent to true.
geq[xy, 0] := true[]
           if add(x, y, xy),
              geq(x, 0, true[]),
              geq(y, 0, true[])
geq[xxyy, 0] := true[] if add(mul[x, x], mul[y, y], xxyy).

% ... other reasoning rules...

% if x>0 is equivalent to true,
% every abs[x] in the database should be equivalent to x
x := abs[x] if geq(x, 0, true[])
```

The above program can be seen as implementing a small theorem prover in Gogi. Whenever it sees abs[x], a query about `x >= 0` will be issued to the database. If later `x >= 0` is proven to be equivalent to true, a distinguished sort value, `abs[x]` will be put in the same e-class as `x`.

All these rewrite will be very hard to express in egg.

Ext 3: Functional Dependency Repair

FDs can be violated: what if the user introduced two values for the same set of determinant columns? In this case, we need to repair the FDs. We have seen such examples many times in previous sections. For example, rules like `R[x1, ..., xk] := ...` will add new values to `R` indexed by `x1, ..., xk`, and it is likely that there are already other tuples with the same prefix `x1, ...,` `xk`. These rules may potentially cause violation of functional dependencies. In general, there are two kinds of violations:

Case 1. If the dependent column is a sort value, Gogi will unify the two sort values later in the iteration. We can think of a term of a sort in Gogi as a constant in some theories, which refers to some element in the model. But we don't know which element it refers to. However, by repairing functional dependencies, we can get some clues about what the structure will look like. Consider the following program

```
rel add(expr, expr) -> expr.
rel num(i64) -> expr.

% add the fact 2 + 1, where the last column is auto-generated.
add[num[2], num[1]].

% add the fact 2+1, but the last column is add[num[1], num[2]]
% (add[num[1], num[2]] is created on the fly because
% it occurs at the left hand side.)
add(num[2], num[1],
    add[num[1], num[2]]).
```

Because now (without repairing) `add[num[1], num[2]]` will contain two rows. The functional dependency is violated. If we think of rewriting under functional dependency as a process of finding a model for the sort, then what do we learn from this violation? We learned that, to respect the functional dependency, the two sort values must be the same thing! Therefore the expr originally referred by `add[num[2], num[1]]` and by `add[num[1], num[2]]` will be treated as the same expr and no longer be distinguishable in Gogi! As we will show later, when a Gogi program reaches the fixpoint, it produces a valid, minimal model for the relations and the sorts such that the rewrite rules and the functional dependencies are both respected.

TODO: mention no global union-find

Case 2. What if the dependent column is a regular type as a Rust struct or an integer? Well, we also need to unify them, but in a different way. The idea here is to describe these values with a algebraic structure, which in this case is a lattice. A lattice has a bottom (means does not exist) and a top (means conflicts). Similar to Flix, lattice values will grow by taking the least upper bound of all the violating tuples. In that sense, Gogi also generalizes Flix (as is described in the PLDI '16 paper).

Ext 4: Seamless Interop with Rust

This proposed extension takes inspiration from recent work on Ascent, an expressive Datalog engine that has seamless integration with the Rust ecosystem.

One interesting feature of Ascent is that it allows first-class introspection of the column values. Ascent use this feature to support features like first-class environment (this and the next example are both from page 4 of the Ascent paper; comments are mine):

```
sigma(v, rho2, a, tick(e, t, k)) <--
  sigma(?e@Ref(x), rho, a, t), // the environment rho is enumerated here
  store(rho[x], ?Value(v, rho2)), // rho[x] is used as an index for store
  store(a, ?Kont(k));
```

One thing though is that Ascent allows enumerating structs as a relation with the `for` keyword. For example:

```
sigma(v, rho2, store, a, tick(v, t,k)) <--
  sigma(?Ref(x), rho, store, a, t),
  // enumerating store[&rho[x]]
  for xv in store[&rho[x]].iter(), if let Value(v,rho2) = xv,
  // enumerating store[a]
  for av in store[a].iter(), if let Kont(k) = av;
```

This makes Ascent have a more macro-y vibe, which makes sense since the whole Ascent frontend is based on Rust's procedural macros. However, I think the similar can be easily achieved inside the relational land, so in a full-fledged relational language like Gogi, the `for` syntax may not be necessary.

Seamless interop with Rust is in general very powerful. In fact, we have already used this feature a lot. For example, lattices in Gogi are structs defined in Rust that implements certain traits. So rules like `hi(x, n.into()) :- num(n, x).`, will call methods in the corresponding struct (e.g., `n.into()`).

In general, these user-defined functions introduced functional dependencies from domains of functions to their range. For example, rule `hi(x, n.into()) :- num(n, x).` can be viewed as `hi(x, n_into) :- num(n, x), into_rel(n, n_into)` with functional dependency from `n` to `n_into`. Advanced join algorithms like worst-case optimal joins can leverage these functional dependencies to optimize the query.

The Model Semantics of Gogi and its Evaluation

In this section, we will focus on the problem of how to formalize Gogi and how to evaluate Gogi programs. This section will first give the model semantics of Gogi. Then, It will describe rebuilding, an essential procedure for evaluating and maintaining e-graphs, namely rebuilding, in the Gogi setting. Finally, we will discuss how Gogi's matching procedure can benefit from semi-naive evaluation, a classic evaluation algorithm in Datalog ## The Model Semantics

For simplicity, we assume that in our core language there will only be one (interpreted) lattice L and one (uninterpreted) sort S.

A relation declaration in core Gogi has the following shape:

$$\text{rel } R(c_1, \dots, c_p) \to (s_1, \dots, s_m, l_1, \dots, l_n)$$

where s_j is a sort value column, l_k is a lattice value column, and c_i can be either a lattice or sort value column. Such declaration specifies a relation with schema $(c_1, \dots c_p, s_1, \dots, s_m, l_1, \dots, l_n)$ and implies the following logical constraint:

$$\forall \vec{c}, \vec{s}, \vec{l}, \vec{s'}, \vec{l'}.R(\vec{c}, \vec{s}, \vec{l}) \wedge R(\vec{c'}, \vec{s'}, \vec{l'}) \to \vec{s} = \vec{s'} \wedge \vec{l} = \vec{l'}$$

where $\vec{x}$ denotes a vector of variables $x_1, \dots, x_k$.

Note in the core formalization, we don't assign default values to lattices and we assume the bottom is the default value, and each default value is specified as a rewrite rule.

A rewrite rule in the core looks like follows:

$$\exists \vec{z}.R_1(\vec{x_1}), \dots, R_n(\vec{x_n}) \leftarrow S_1(\vec{y_1}), \dots, S_m(\vec{y_m}).$$

All variables x_{ij} in the head are either bound in the body or existentially quantified. Importantly, existentially quantified variables in the core quantifies over sort values and must be "inferrable" from the functional dependency, meaning that they must be a dependent variable within some relation atoms. For example, the following Gogi program is not valid, because rule `R(1, c)` can be triggered arbitrarily many times, each with different `c`:

```
sort S.
rel R(i64, S).
R(1, c). % translated from R[1].
```

This "inferrable" constraints can be formalized as

$$\forall i.\exists j.z_i \in \text{dep}(R_j(\vec{x_j})),$$

where $\text{dep}\left(R_j\left(\vec{x_j}\right)\right)$ is the set of dependent variables in atom $R_j(\vec{x_j})$.

The rewrite rule in the core is further translated to the following logical constraint:

$$\forall \vec{y}.\left(\bigwedge_i \vec{y_i} \sqsubseteq_L S_i \to \exists \vec{z} \in S^k.\bigwedge_i \vec{x_i} \sqsubseteq_L R_i\right),$$

where $\vec{y}$ is the set of variables occurring in $\vec{y_1}, \dots, \vec{y_m}$ and

$$(\vec{c}, \vec{s}, \vec{l}) \sqsubseteq_L R \iff \exists \vec{l'}.R(\vec{c}, \vec{s}, \vec{l'}) \wedge \left(\bigwedge_i l_i \leq_L l'_i\right)$$

In English, whenever a valuation of variables satisfied right-hand side, there exists some $\vec{z}$ such that the left-hand side also "holds", in the sense that some tuples in the relation subsumes the substituted left-hand side.

The result of evaluating a (core) Gogi program is the minimal model $(S_{\text{min}}, D_{\text{min}})$ that satisfies the logical constraints derived from the program, where S_{min} is the minimal S of sort values (up to isomorphism) and D_{min} is the minimal database instance (i.e., interpretation of relations) with domain L and S_{min}.

This core formalization should look familiar to people who know the chase: Functional dependencies are equality-generating dependencies (EGD), and rewrite rules are tuple-generating dependencies (TGD). However, there are several critical distinctions between Gogi and the chase. First, the chase has both labelled nulls and constants, and unifying two constants will cause a conflict. Sort values in Gogi can be thought as labelled nulls, and there is no matching concept for constants in Gogi. Moreover, Gogi supports lattice values. This feature has its root in both egg's e-class analyses and relational languages like Flix, and is necessary for different kinds of analysis tasks that Gogi strives to support. Finally, and perhaps most importantly, Gogi has this "inferrable" constraint for existential variables. This constraint leads Gogi to have a single very efficient application algorithm of executing both EGDs (from functional dependencies) and TGDs (from rewrite rules). In contrast, EGDs and TGDs in the chase are executed independently. Without the lattice values and lifting the inferrable constraint, Gogi programs can be expressed in the chase.

The Evaluation Algorithm

The evaluation algorithm of Gogi programs consists of two parts. The core of the evaluation is the invariant-maintaining rebuilding algorithm, which is inspired both by the rebuilding algorithm of egg and by the evaluation algorithm of the chase. The second part involves matching and applying Gogi rules. Applying Gogi rules is efficient. In the chase's terminology, thanks to the above mentioned inferrable constraint, rule application in Gogi is able to utilize functional dependencies to avoid to generate unnecessary nulls. Moreover, because Gogi programs are monotonic computations over the relational database in nature, they can benefit from the semi-naive evaluation algorithm of Datalog. We call this semi-naive matching, which can be seen as a further improvement over relational e-matching.

Rebuilding

The rebuilding algorithm:

```
todo = mk_union_find()
domain = mk_set()

def union_sort(s1, s2):
  todo.union(s1, s2)
  domain.add_all([s1, s2])

def refresh_todo():
  todo = mk_union_find()
  domain = mk_set()

def on_insert(R, tup):
  # find the tuple by its determinant columns
  orig_tup = R.find_by_determinant(tup.det)
  if orig_tup is None:
    R.insert(tup)
  else:
    # enumerate each dependent column
    for c1 in tup.dep:
      col = c1.col
      c2 = orig_tup[col]
      if col.is_sort():
        s1 = todo.get_or_create(c1)
        s2 = todo.get_or_create(c2)
        union_sort(s1, s2)
      else:
        orig_tup.set_col(col, c1.lat_max(c2))

def normalize(tuple, union_find):
  return tuple.map(lambda val:
    union_find.get_or_default(val, default = val))

def rebuild(DB):
  while not todo.is_empty():
    # take todo into the local scope
    union_find = todo
    refresh_todo()

    to_remove = mk_set()
    to_insert = mk_set()

    for val in domain:
      for R in DB:
        for col in R.cols:
          for tup in R.index_by(col = col, val = val):
            new_tup = normalize(tup, union_find)
            if new_tup != tup:
              to_remove.add((R, tup))
              to_insert.add((R, new_tup))
    DB.remove_all(to_remove)
    # may trigger on_insert
    DB.insert_all(to_insert)
```

Applying rewrite rules

```
def batch_rewrite(pats, DB):
  to_insert = mk_set()
  for (lhs, rhs) in pats:
    for subst in match(DB, lhs):
      subst = chase(DB, subst, lhs, rhs)
      for (R, atom) in rhs:
        to_insert.add((R, atom.apply(subst)))
  DB.insert_all(to_insert)
  return to_insert.is_empty()

def chase(DB, subst, lhs, rhs):
  shouldContinue = True
  while shouldContinue:
    shouldContinue = False

    for atom in rhs:
      det_vars = atom.get_det_vars()
      if det_vars.is_subset_of(subst.get_domain()):
        shouldContinue = True

        R = DB.get_rel(atom.rel)
        det = det_vars.apply(subst)
        tup = R.find_by_determinant(det)
        for var in atom.get_dep_vars():
          col = var.col
          if var.is_sort():
              if tup is None: continue
              value = tup.get_by_col(col)
              sort_update(subst, var, value)
          else:
            value = tup is None ? col.lat_init(det)
                                : tup.get_by_col(col)
            lat_update(subst, var, value)

  for var in rhs.get_all_vars():
    if !subst.contains(var):
      assert var.is_sort():
      subst[var] = new_sort_value(var.sort)

def lat_update(subst, var, value):
  if subst.contains(var):
    subst[var] = subst[var].lat_max(value)
  else:
    subst[var] = value

def sort_update(subst, var, value):
  if subst.contains(var):
    union_sort(subst[var], value)
  else:
    subst[var] = value
```

The main algorithm

```
def run(pats, DB, max_iter):
  for iter in range(max_iter):
    if !batch_rewrite(pats, DB):
      return
    rebuild(DB)
```

Semi-Naive Matching

One of the bottleneck in evaluating Gogi programs is matching the left-hand side. Since we are matching over a relational representation of the e-graphs, we are already doing is already relational e-matching. However, we can go one step further: Let `DB'` be the database of tuples that are not touched in the current iteration of rewrite. `DB'` by itself will not produce any interesting new tuples; it has to join with newly generated tuples (i.e., the delta database). This is exactly the semi-naive evaluation algorithm of Datalog. We call this similar optimization in Gogi semi-naive matching. This optimization will be tricky to do over e-graph's DAG representation, yet is fairly obvious in Gogi's full-fledged relational representation.

Gogi by Example

Lambda Calculus

```
sort term.
rel false() -> term.
rel true() -> term.
rel num(i32) -> term.
rel var(string) -> term.
rel add(term, term) -> term.
rel eq(term, term) -> term.
rel lam(string, term) -> term.
rel let(string, term, term) -> term.
rel fix(term, term) -> term.
rel cond(term, term, term) -> term.

rel free(term, string).
rel const_num(term, i32).
rel const_bool(term, bool).
rel is_const(term).

% constant folding
const_num(c, i) :- num(i, c).
const_num[c] := const_num[a] + const_num[b]
             if c = add[a, b].
const_bool[true[], true].
const_bool[false[], false].
const_bool[c] := true[]
              if c = eq[a, a].
const_bool[c] := false[]
              if c = eq[a, b],
                 const_num[a] != const_num[b].
is_const(c) :- const_num[c].
is_const(c) :- const_bool[c].

% free variable analysis
free(c, v):- var(v, c).
free[c] := free[a] if c = add[a, b].
% unfolds to free(c, v) :- free(a, v), if c = add[a, b].
free[c] := free[b] if c = add[a, b].
free[c] := free[a] if c = eq[a, b].
free[c] := free[b] if c = eq[a, b].
free(c, v) :- free(body, v)
           if c = lam[x, body],
              v != x.
% fv(let(x, a, b)) = free(a) + (free(b) \ x)
free(c, v) := free(b, v)
           if c = let[x, a, b],
              v != x.
free[c] := free[a]
           if c = let[x, a, b].
free(c, v) :- free(body, v)
           if c = fix[x, body],
              v != x.
free[c] := free[pred] if c = cond[pred, a, b]
free[c] := free[a] if c = cond[pred, a, b]
free[c] := free[b] if c = cond[pred, a, b]

% if-true
then := cond[true[], then, else].
```

```
% if-false
else := cond[false[], then, else].

% if-elim
else := cond[eq[var[x], e], then, else]
     if let[x, e, then] = let[x, e, else]
let[x, e, then] :- cond[eq[var[x], e], then, else]
let[x, e, else] :- cond[eq[var[x], e], then, else]

% add-comm
add[b, a] := add[a, b]

% add-assoc
add[a, add[b, c]] := add[add[a, b], c]

% eq-comm
eq[b, a] := eq[a, b]

% fix
let[v, fix[v, e], e] := fix[v, e]

% beta
let[v, e, body] := app[lam[v, body], e]

% let-app
app[let[v, e, a], let[v, e, b]] := let[v, e, app[a, b]]

% let-add
add[let[v, e, a], let[v, e, b]] := let[v, e, add[a, b]]

% let-eq
eq[let[v, e, a], let[v, e, b]] := let[v, e, eq[a, b]]

% let-const
c:= let[v, e, c] if is_const(c)

% let-if
cond[let[v, e, pred], let[v, e, then], let[v, e, else]]
    := let[v, e, cond[pred, then, else]]

% let-var-same
e := let[v1, e, var[v1]]

% let-var-diff
var[v2] := let[v1, e, var[v2]] if v1 != v2
```

```
% let-lam-same
lam[v1, body] := let[v1, e, lam[v1, body]]

% let-lam-diff
lam[v2, let[v1, e, body]] := let[v1, e, lam[v2, body]]
                             if v1 != v2, free(e, v2)
% capture-avoiding subst
lam[fresh,
  let[v1, e,
    let[v2, var[fresh],
        body]]]
  :=
let[v1, e, lam[v2, body]]
  if v1 != v2, !free(e, v2), fresh = gensym().
```

Type Inference for HM Type System

Comparison to other languages

Comparison to Rel

Comparison to Souffle

Comparison to Flix