A tutorial on the imaginary Gogi language

Yihong Zhang

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 egg# 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).
```

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 that takes two `expr`s.

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 associated to head atoms do not necessarily have to be bound in the bodies. 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 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 it 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 in nested inside 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 introduce how we resolve this kind of violations in the section on Functional Dependency Repair.

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]

[1] The last rule in this example has a single variable on the left-hand side, but the above

```
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

% ... 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.

---

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])`.

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